ScienceDirect
Disponible en www.sciencedirect.com
Revista Iberoamericana de Automática e Informática industrial 12 (2015) 325–337
www.elsevier.es/RIAI

Diseño del software de control de un UUV para monitorización oceanográfica usando un modelo de componentes y framework con despliegue flexible

Francisco Ortiz[a*], Antonio Guerrero[b], Francisco Sánchez-Ledesma[a], Francisco García-Córdova[b], Diego Alonso[a], Javier Gilabert[b,c]

[a] *División de Sistemas e Ingeniería Electrónica (DSIE), Campus Muralla del Mar, Universidad Politécnica de Cartagena, 30202 Cartagena, España.*
[b] *Laboratorio de Vehículos Submarinos (LVS), Campus Muralla del Mar, Universidad Politécnica de Cartagena, 30202 Cartagena, España.*
[c] *Departamento de Ingeniería Química y Ambiental, Campus Alfonso XIII, Universidad Politécnica de Cartagena, 30203 Cartagena, España.*

Resumen

Los vehículos submarinos no tripulados (*Unmanned Underwater Vehicles*, UUVs) se diseñan para misiones de monitorización, inspección e intervención. En estudios oceanográficos y de monitorización ambiental son cada vez más demandados por las innumerables ventajas que presentan con respecto a las tecnologías tradicionales. Estos vehículos son desarrollados para superar los retos científicos y los problemas de ingeniería que aparecen en el entorno no estructurado y hostil del fondo marino en el que operan. Su desarrollo no solo conlleva las mismas dificultades que el resto de los robots de servicio (heterogeneidad en el hardware, incertidumbre de los sistemas de medida, complejidad del software, etc.), sino que además se les unen las propias del dominio de aplicación, la robótica submarina: condiciones de iluminación, incertidumbre en cuanto a posición y velocidad, restricciones energéticas, etc. Este artículo describe el UUV AEGIR, un vehículo utilizado como banco de pruebas para la implementación de estrategias de control y misiones oceanográficas. También describe el desarrollo de una cadena de herramientas que sigue un enfoque dirigido por modelos, utilizada en el diseño del software de control del vehículo, así como un framework basado en componentes que proporciona el soporte de ejecución de la aplicación y permite su despliegue flexible en nodos, procesos e hilos y pre-verificación del comportamiento concurrente. Su diseño ha permitido desarrollar, comprobar y añadir los componentes que proporcionan el comportamiento necesario para que el UUV AEGIR pudiera completar con éxito distintos tipos de misiones oceanográficas.

Palabras Clave:
UUV Unmanned Underwater Vehicle, monitorización oceanográfica, framework de componentes, modelo de componentes, configuración de despliegue, análisis de concurrencia.

1. Introducción

Los vehículos submarinos no tripulados (*Unmanned Underwater Vehicles*, UUVs) son vehículos diseñados para operar bajo el agua sin tripulantes humanos. En sus distintas formas se pueden dividir en dos categorías: ROVs (*Remotely Operated underwater Vehicles,* vehículos submarinos no autónomos teleoperados remotamente) y AUVs (*Autonomous Underwater Vehicles*, vehículos submarinos autónomos). Mientras que los AUVs operan independientemente, sin la orden directa de un humano, los ROVs son controlados desde superficie por un operador humano. Los ROVs reciben el suministro de energía por un "cordón umbilical", aunque también hay modelos donde no existe el cableado físico y se controlan remotamente.

En aplicaciones militares, los AUVs son más a menudo conocidos simplemente como UUVs, que será la terminología adoptada en este artículo. La necesidad de UUVs se ha hecho cada vez más evidente a medida que el mundo presta mayor atención a las cuestiones medio ambientales y de recursos (cartografía oceanográfica, el muestreo de océanos, inspección/estudio del suelo marino), así como las tareas científicas y militares (inteligencia, reconocimiento, vigilancia y adquisición de objetivos, neutralización de minas), también incluida la seguridad, la vigilancia y la inspección.

En el ámbito de la investigación oceanográfica costera y la gestión integrada de los recursos del litoral se requiere el desarrollo de nuevas técnicas, tecnologías y dispositivos capaces de explorar diferentes hábitats con el fin de protegerlos y gestionarlos, cumpliendo la legislación vigente, como la directiva del Marco de Agua de la Unión Europea (Directiva 2000/60/CE), estrategias marinas (Directiva 2008/56/CE), la conservación de los hábitats naturales (Directiva 92/43/CEE) y la protección del medio ambiente marino de las regiones costeras (Directiva 1999/802/CE).

* Autor en correspondencia.
Correos electrónicos: francisco.ortiz@upct.es (F. Ortiz),
URL: www.dsie.upct.es/personal/fjortiz (F. Ortiz)

http://dx.doi.org/10.1016/j.riai.2015.06.003

Muchos UUVs se han desarrollado para superar los retos científicos y los problemas de ingeniería causados por entornos submarinos no estructurada y peligrosos (Fossen-1994, Stutters et al 2008). A pesar de que los avances tecnológicos han permitido mejorar la hidrodinámica, el sistema de control y las comunicaciones, todavía hay que superar retos en el desarrollo de UUVs derivados de las características propias del dominio submarino, como restricciones energéticas, movimiento en un entorno tridimensional con perturbaciones dinámicas importantes, y ausencia de referencias para localización, entre otras. Todo ello hace que el diseño de subsistemas de navegación y control autónomos de un UUV sea una tarea ardua y compleja. En sistemas tan complejos como estos la organización arquitectónica del software es crucial (Ridao et al 2000, Carreras et al 2005, Eickstedt et al 2009). La mayoría de los robots descritos en estos artículos implementan arquitecturas híbridas, de forma similar a como sucede en los robots de servicio terrestres. Sin embargo, el diseño de los módulos que componen la arquitectura software suele ser abordado e implementado por ingenieros de sistemas, que son expertos en diseños algorítmicos pero no en la aplicación de principios de ingeniería del software (Bruyninckx 2008). Quizá por esta razón, en muchos casos el análisis de requisitos y las técnicas de diseño y reutilización de software no se usan consistentemente. Así, los requisitos de eficiencia del software sólo se consideran para las tareas de control de bajo nivel (Schlegel 2006) y generalmente el software desarrollado no proporciona mecanismos que faciliten su reutilización en escenarios similares.

Con estas consideraciones en mente, en este artículo se presentan parte de los desarrollos obtenidos en el marco del proyecto coordinado MISSION-SICUVA: (1) la construcción de un prototipo de UUV para investigación oceanográfica y su sistema de control de inspiración neurobiológica, con aplicaciones tales como la monitorización de la calidad del agua, la batimetría y cartografía de alta resolución del fondo marino y la validación de modelos hidrodinámicos 3D, y (2) el desarrollo de su software de control utilizando las técnicas de la Ingeniería del Software que favorezcan la reutilización de código y de diseños, incorporando los requisitos específicos del dominio: eficiencia, fiabilidad, escasez de recursos computacionales y limitaciones energéticas. Además, dadas las características del entorno en que opera el robot, es muy importante asegurar y verificar el funcionamiento del sistema antes de que comience su misión. También se describe como la cadena de herramientas utilizada permite llevar a cabo una pre-verificación de los requisitos temporales del software.

El UUV AEGIR (AEGIR-*Dios de los mares en la mitología nórdica*), es una re-construcción y modificación de un ROV®*Gaymarine* cedido por la Armada Española y desarrollado en el Laboratorio de Vehículos Submarinos (LVS[1]) de la Universidad Politécnica de Cartagena. Los autores pertenecientes al DSIE[2] propusieron un marco genérico de desarrollo software para este tipo de sistemas basado en componentes, modelos y frameworks, descrito en (Alonso et al 2012). El presente trabajo amplía el trabajo anterior con el desarrollo de una cadena de herramientas en Eclipse denominada C-Forge que soporta el proceso completo de desarrollo dirigido por modelos, así como un framework basado en componentes, denominado FraCC, que proporciona el soporte de ejecución para estas aplicaciones. Entre las características más sobresalientes de FraCC destaca la flexibilidad en la configuración del despliegue de las aplicaciones en nodos, procesos e hilos y la posibilidad de llevar a cabo el análisis temporal (planificabilidad) de las mismas.

El resto del artículo se organiza en seis secciones. La segunda sección resume el estado de la técnica actual en el desarrollo de software para robots de servicio y estrategias de control para robots submarinos. En la tercera sección se describen los dos vehículos que constituyen el UUV AEGIR. En la sección número cuatro se describe la cadena de herramientas C-Forge, las principales características de FraCC y el proceso de generación de modelos de análisis. La sección seis ilustra el uso de C-Forge para el desarrollo del software de control de AEGIR. Los dos últimos puntos describen algunos resultados obtenidos en los test realizados en el Mar Menor y las conclusiones del artículo así como las futuras líneas de investigación.

2. Estado de la Técnica

El desarrollo e integración de software está ampliamente reconocido como un elemento crucial para reducir la separación existente entre la elaboración de prototipos robóticos y su fabricación en serie (Schlegel 2006). La reutilización de soluciones probadas es crucial para poder elaborar rápidamente prototipos, no sólo para prevenir la reescritura del código sino para garantizar su calidad, estabilidad y robustez. Existen varias alternativas para el diseño de software, focalizadas en la reutilización. Sin embargo, no es sencillo elegir una o realizar una comparativa, a pesar de los esfuerzos realizados en proyectos como RoSta (Rosta 2013), puesto que cada una de ellas proporciona soluciones ligeramente diferentes y emplean diversos paradigmas de desarrollo. Una posible clasificación de estas herramientas es la siguiente:

- **Librerías de algoritmos:** OpenSLAM, Intempora, etc (Rosta 2013). Estas soluciones se caracterizan por que el código de usuario invoca directamente los algoritmos proporcionados por la librería. Este tipo de soluciones requieren que el usuario proporcione el software de soporte necesario para el resto del sistema, como drivers, comunicaciones, gestión de la concurrencia, etc.
- **Toolkits y middleware robóticos:** ROS, Player/Stage, MRPT, YARP, etc. (Rosta 2013, Schlegel et al 2011). Estas soluciones amplían el soporte proporcionado por la categoría anterior con middleware de comunicación para conseguir distribución y modularidad. Las aplicaciones se construyen como un conjunto de binarios heterogéneos, puesto que las herramientas no fuerzan habitualmente la estructura interna de los módulos que forman la aplicación. No existe ningún mecanismo, salvo la disciplina del programador, que prevenga que cualquier objeto que forma parte de un módulo envíe mensajes a otros módulos.
- **Frameworks basados en componentes:** OROCOS, ORCA2, ROBOCOMP, GenoM, SmartSoft, (Schlegel et al. 2011) etc. Estas soluciones proporcionan el mismo nivel de soporte que los middleware robóticos, pero intentan superar las limitaciones de estos mediante el empleo de componentes software: artefactos que especifican claramente qué servicios proporcionan y requieren a través de puertos, que son los únicos canales de comunicación permitidos.

El trabajo presentado en este artículo se puede enmarcar en el tercer grupo, puesto que incluye un framework orientado a

[1] http://www.upct.es/lvs

[2] http://www.dsie.upct.es

componentes. De entre los frameworks mencionados, SmartSoft es quizás la más parecida al enfoque que se describe en este artículo, puesto que ambas diseñan los componentes a nivel de modelos, considerándolos como unidades arquitectónicas, que posteriormente son traducidas e implementadas en C++ sobre una plataforma de ejecución soportada por un framework orientado a componentes.

Por otro lado, como se mencionó en la introducción, la cadena de herramientas C-Forge[1] permite llevar a cabo una pre-verificación del cumplimiento de los requisitos temporales del software. En lugar de crear una nueva herramienta, utilizamos las ventajas inherentes del enfoque dirigido por modelos para generar transformaciones automáticas que nos permiten utilizar algunas de las herramientas existes. Entre ellas se pueden diferenciar las que utilizan formalismos como lógica temporal, por ejemplo Spin (Ben-Ari 2008), o autómatas temporizados, por ejemplo Uppal (Behrmann et al 2001), para llevar a cabo la verificación temporal a partir de la especificación de requisitos. La OMG define el estándar MARTE (Omg 2009) para modelar no solo los requisitos temporales de la aplicación, sino todos aquellos relacionados con el modelado de la calidad de servicio (*QoS*). MAST (Medina et al 2001) es una herramienta alineada con MARTE para especificar y analizar sistemas de tiempo real, que proporciona un facilidad de análisis de sensibilidad que informa al usuario de lo lejos o cerca que se encuentra el sistema de cumplir los requisitos temporales. Por último, Cheddar (Singhoff et al 2009) es una herramienta que permite analizar la planificabilidad de sistemas de tiempo real, que implementa la mayor parte de los algoritmos y pruebas que se utilizan habitualmente en este tipo de sistemas. Dado que además está integrado con herramientas de diseño de sistemas empotrados tan importantes como AADL y TOPCASED, se consideró que Cheddar era la herramienta más adecuada para llevar a cabo el análisis de las aplicaciones desarrolladas con FraCC.

Otro de los puntos tratados en este artículo es el desarrollo de un neuro-controlador del UUV en entornos no estructurados. En general, la navegación y generación de trayectoria autónoma en combinación con la evasión de obstáculos es una cuestión de importancia fundamental en la robótica (Carreras et al 2005). Tradicionalmente se han utilizado diversos enfoques al problema incluyendo distintos esquemas de control (adaptativo, difuso, servo-visual) (Antonelli et al 2001). Una alternativa a este problema son los modelos matemáticos de los sistemas neuronales que representan un vínculo entre la biología y la ingeniería (Auke 2008).

Para la generación de trayectorias en tiempo real se han propuestos varios modelos de redes neuronales a través del aprendizaje (Chang y Gaudiano 1998, Guerrero-González et al 2010). Desde algoritmos de mapeo auto-organizativos de Kohonen (Ritter et al 1989) hasta redes multicapa con aprendizaje de refuerzo para planificación de ruta (Fujii et al 1998). Sin embargo, las trayectorias generadas utilizan métodos de aprendizaje que no son óptimos, particularmente durante la fase inicial del aprendizaje

Una alternativa para resolver los aprendizajes óptimos y la generación de trayectorias autónomas son los modelos neuronales bio-inspirados en la neurociencia. Estos modelos forman su arquitectura basándose en la actividad de las áreas motoras que intervienen en los animales para llevar a cabo una tarea, y su aprendizaje se basa en el comportamiento ante distintos estímulos (García-Córdova 2007, Auke 2008). En Chang y Gaudiano (1998) se describe una red neuronal bio-inspirada para la evasión de obstáculos basada en un modelo de condicionamiento clásico y operante. Las redes neuronales basadas en las leyes de aprendizajes asociativos pueden modelar los mecanismos de condicionamiento clásico, mientras que las redes neuronales basadas en leyes de aprendizaje por refuerzo pueden modelar los mecanismos de condicionamiento operante (Ritter et al 1989, Guerrero-González et al 2010). El aprendizaje por refuerzo se utiliza para adquirir habilidades de navegación en vehículos autónomos y actualizaciones tanto en el modelo de vehículo y el comportamiento óptimo al mismo tiempo (Carreras et al 2005).

El controlador de movimiento propuesto para el AEGIR implementa una arquitectura neuronal que está compuesta por una red neuronal de mapeo de dirección auto organizativa y una red neuronal para el comportamiento de evasión de obstáculos, ambas de inspiración neurobiológicas. La red mimetiza de forma simplificada los mecanismos que permiten al cerebro recoger la información sensorial para controlar los comportamientos adaptativos de navegación autónoma y de evasión de obstáculos, aplicando leyes de aprendizajes asociativos y por refuerzo tal como lo hacen los animales (García-Córdova 2007, Guerrero-González et al 2010, Auke 2008).

3. Descripción de AEGIR-UUV

AEGIR es un UUV construido a partir de un submarino ROV®*Gaymarine* fuera de servicio y diseñado en los años 80, utilizado para tareas de caza minas por la Armada Española durante esos años. Se ha utilizado como banco de pruebas para la implementación de estrategias de control y para misiones oceanográficas en la zona del Mar Menor, que es una gran laguna costera de la Península Ibérica y una de las más grandes de Europa. Entre las misiones a realizar por el AEGIR destacan inspecciones de larga distancia, estudios de larga duración, estudios oceanográficos, monitoreo de la calidad del agua, levantamiento de batimetrías y cartografía del fondo marino, detección y localización de pecios, entre otros (Guerrero-González et al 2011).

La plataforma robótica AEGIR-UUV está compuesta por un vehículo submarino que remolca una boya de superficie (véase la Figura 1). El vehículo de superficie tiene forma de pequeña embarcación que proporciona la energía mediante un grupo electrógeno y hace de enlace de comunicaciones entre el vehículo submarino y otros nodos mediante un enlace inalámbrico, como la estación de teleoperación, un PC portátil donde se ejecuta el software de monitorización remota. El conjunto puede operar autónomamente durante 12 horas a una velocidad máxima de 4 nudos, lo que le permite cubrir amplias zonas de aguas poco profundas durante las misiones de inspección. El vehículo submarino tiene una capacidad de carga de 25 Kg., lo que le confiere una gran versatilidad para incorporar instrumentación sin limitaciones de conectividad ni de energía.

El vehículo submarino dispone de un sonar de barrido lateral y una cámara de vídeo para monitorizar el entorno y de sensores para medir parámetros físicos y químicos de la calidad del agua. Tiene dos cascos: cuerpo y cabeza. El casco del cuerpo incorpora el paquete de baterías, hélices, inclinómetro, sensor de presión y sonar de barrido lateral, mientras que el casco de la cabeza incluye dispositivos de percepción (cámara de vídeo, sonar de imagen, altímetro acústico, y la unidad de posicionamiento). Las principales características del AEGIR se muestran en la Tabla 1. El vehículo submarino dispone de tres nodos de comunicación, en

[1] http://www.dsie.upct.es/cforge

cabeza, cuerpo y motores, interconectados a través de un bus CAN (*Controller Area Network*), tal como se muestra en la Figura 2. En el nodo 1 está situada la CPU (*embedded-PC* NEO-13-I7-620M-POE, con procesador Inter Core i7-620M) donde se ejecutará el software de control. Un NI RIO Programmable Automation Controllers (sbRIO-9606), hace de puente entre Ethernet y la red CAN.

Figura 1. Plataforma submarina experimental no tripulada (AEGIR-UPCT): (a) Vehículo submarino autónomo; (b) Vehículo de superficie.

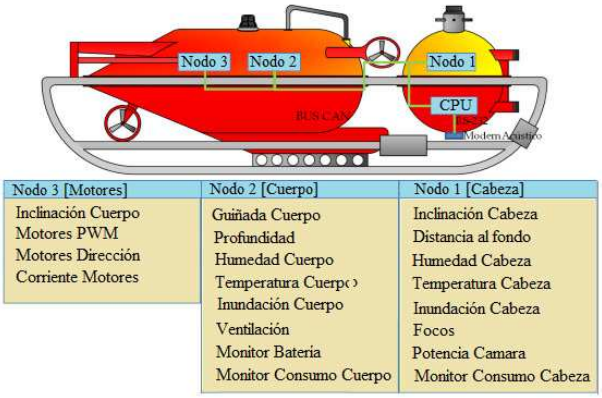

Figura 2. Nodos internos de comunicación del AEGIR-UPCT.

Tabla 1: Características principales del AEGIR-UUV.

Peso del vehículo: 160 kg. Dimensiones: 1680 x 600 x 600 mm. Max. Velocidad: 4 nudos (48V), 2 nudos (24V). Profundidad operativa: 60 m.
Datos de prueba: Lastre: 15 kg. Peso total: 163.4 kg. Desplazamiento: 163.8 dm^3. Desplazamiento del lastre: 1.92 dm^3. Empuje neto máximo: 2.32 kg
Elementos: Cámaras y luces. Sonar de barrido lateral. Sonar del perfil del subsuelo marino. Cámara de video. Sistema inercial. Módems acústicos Sensor de profundidad. Sistema de posicionamiento USBL Módulo del sistema de control de distribución. Analizador de nitrato sumergible de ultravioleta. Sonda multiparamétrica (YSI®-66000 V2-4).

4. Cadena de Herramientas C-Forge

C-Forge es una cadena de herramientas abierta desarrollada sobre la plataforma Eclipse, que emplea sus facilidades de diseño dirigido por modelos para soportar un proceso de desarrollo basado en componentes. El trabajo que aquí se describe es una mejora y ampliación del presentado anteriormente en (Alonso et al 2012), con un ejemplo de aplicación al desarrollo del software de un UUV. En dicho trabajo se proponía un marco de diseño conceptual para aplicaciones basadas en componentes que utilizan frameworks de componentes como soporte de ejecución, en lugar de una transformación de modelos que genera todo el código de implementación. Con los frameworks de componentes, el desarrollo puede centrarse en el código de la aplicación, ya que el framework proporciona todo el soporte requerido para su ejecución. C-Forge está formado por las siguientes herramientas (ver figura 3):

- Un lenguaje para modelar aplicaciones basadas en componentes, denominado WCOMM. Una versión preliminar está descrita en (Alonso et al 2010).
- Un framework denominado FraCC, que proporciona el soporte de ejecución necesario para ejecutar las aplicaciones modeladas mediante WCOMM. FraCC se describe en la siguiente sección.

Un componente WCOMM es una entidad que encapsula su estado interno, que consta de una parte estructural y una parte de comportamiento. La parte estructural viene definida por sus puertos y por los mensajes que fluyen a través de ellos, agrupados en interfaces. Estos mensajes se envían siguiendo el esquema de comunicación asíncrono sin respuesta. El comportamiento se define mediante un autómata temporizado (Bengtsson y Yi 2004), que es una máquina de estados finita, similar a la que define UML, extendidas con propiedades temporales. Es decir, el usuario modela el comportamiento del componente mediante estados, transiciones, eventos, guardas y regiones, tanto ortogonales como jerárquicas. Cada estado puede tener opcionalmente definida una actividad interna, que se asociará posteriormente en FraCC con código. En WCOMM también se modela lo que denominamos "carcasa" de la actividad, formada por los mensajes que intercambia y los eventos que genera. Estos eventos, junto con la recepción de mensajes a través de los puertos, son los responsables del cambio de estado del componente, y son por tanto, los que establecen la conexión entre estructura y comportamiento. Finalmente, una aplicación se modela como un conjunto de componentes conectados entre sí.

Es importante destacar que los autómatas temporizados modelan no solo el ciclo de vida de los compontes en WCOMM, sino que en general controlan bajo qué circunstancias se ejecutan las actividades de sus estados. Este formalismo fue escogido debido a que es especialmente adecuado para el modelado de aplicaciones reactivas con restricciones temporales, como las que pueden encontrarse normalmente en la robótica.

WCOMM utiliza tres herramientas y sigue un proceso de diseño iterativo, en el que el usuario puede ir modelando incrementalmente la aplicación con cada una de ellas (figura 3-A). Estas tres herramientas son (1) un editor textual para definir los tipos de datos, los mensajes que se intercambian los componentes y la "carcasa" de las actividades, (2) un editor gráfico para modelar compontes: puertos, mensajes enviados y recibidos por ellos y el autómata temporizado que describe su

comportamiento, y (3) un editor gráfico para modelar la arquitectura de la aplicación a partir de los componentes definidos previamente. En la siguiente sección se muestra la aplicación de estas herramientas para el modelado del software de AEGIR.

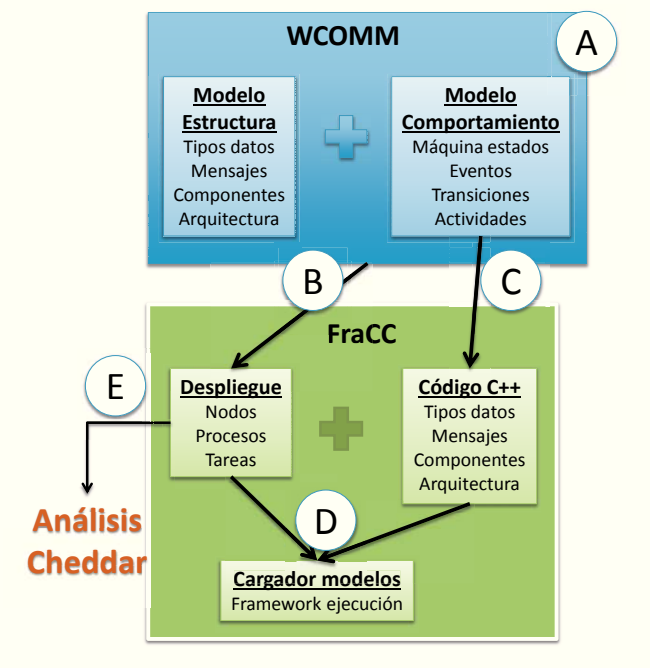

Figura 3. Cadena de herramientas C-Forge: (A) modelado de la aplicación basada en componentes con WCOMM, (B) generación del modelo de despliegue FraCC, (C) generación de clases C++ para código de usuario, (D) instanciación y ejecución de la aplicación FraCC, (E) generación del modelo de análisis para Cheddar.

Como se describe en (Alonso et al 2012), el enfoque permite la integración de otros frameworks de componentes, aunque en tal caso solo estarían disponibles las características soportadas por los mismos. Dado que la mayoría únicamente modelan la parte estructural, habría que desarrollar una transformación adicional que generase la implementación de los autómatas temporizados en el framework escogido.

4.1. Framework para componentes concurrentes FraCC

FraCC es un framework de componentes programado en C++ que fue desarrollado con el doble objetivo de proporcionar (1) soporte completo a las características del modelo de componentes WCOMM, y (2) control completo sobre las características de concurrencia de la aplicación al usuario, que es quien decide cuántos hilos se crean y en qué hilos se ejecutan los componentes, lo que permite el uso de FraCC en aplicaciones con restricciones de tiempo real. En (Alonso et al 2012) se describe una implementación preliminar de FraCC mediante la secuencia de patrones de diseño que se aplicaron en su diseño. Antes de pasar a describir las mejoras introducidas es preciso describir la parte dinámica del framework, es decir, cómo "se animan" los componentes, ya que permanece invariable.

En FraCC, la concurrencia se controla mediante la asignación de cada una de las regiones de los componentes al hilo en que va a ejecutarse, puesto que, por su propia definición, en una región solo puede haber un estado activo, y por tanto, solo se puede ejecutar una actividad por región en un momento dado. Las regiones se planifican de forma interna en cada hilo siguiendo un esquema similar al de un planificador cíclico. Cuando se activa una región en un hilo, ésta comprueba si hay algún evento pendiente de ser procesado antes de ejecutar la actividad del estado activo de la región. En caso de que dicho evento produzca un cambio de estado, realizará dicho cambio antes de ejecutar la actividad del nuevo estado. Por último, la gestión del envío de mensajes entre componentes se realiza mediante un tipo especial de región, presente solo en la parte FraCC. Embebida en esta región hay una actividad que copia los mensajes de salida de un componente a los puertos de entrada de los componentes destinatarios de dichos mensajes. Estas regiones se asignan a hilos de forma similar a como se realiza con el resto de regiones. Esta característica dota de gran regularidad a FraCC, puesto que el usuario fija la carga computacional de cada hilo siempre mediante regiones. Además, le proporciona control completo sobre la ejecución de la aplicación, ya que no hay código "oculto" en FraCC, es decir, código que se ejecute sin que el usuario lo haya asignado previamente a un hilo. Respecto a esta versión previa, se han añadido las siguientes mejoras:

- Se ha diseñado un *modelo de despliegue* que centraliza la gestión de las propiedades de concurrencia de la aplicación.
- Se ha aumentado la flexibilidad de la región de gestión del envío de mensajes.
- Se ha reimplementado la infraestructura de *buffers* de los mensajes para permitir el análisis temporal del desempeño de la aplicación.
- Se ha introducido una capa intermedia que separa el código (actividad) que se ejecuta en las estados del autómata temporizado de la propia estructura de FraCC.

FraCC aporta a C-Forge una transformación modelo-a-modelo que genera el *modelo de despliegue* por defecto para ejecutar la aplicación (figura 3-b), una herramienta para modificar dicho despliegue, una transformación modelo-a-texto que genera los esqueletos de las clases C++ para que el usuario complete el código de las actividades (figura 3-c), una transformación modelo-a-texto que genera el código de instanciación de la aplicación en FraCC y lo enlaza con las actividades (figura 3-d), así como una transformación modelo-a-texto que genera un fichero que sirve de entrada a la herramienta Cheddar para llevar a cabo el análisis de planificabilidad (figura 3-e).

La primera mejora consistió en la creación de un modelo de despliegue, mediante el cual el diseñador especifica los nodos, procesos e hilos en que se va a ejecutar la aplicación WCOMM, junto con la asignación de las regiones de los autómatas temporizados y de gestión del envío de mensajes a hilos. La restricción más importante de este modelo es que todas las regiones de un componente deben asignarse a hilos de un mismo proceso. El modelo de despliegue independiza la arquitectura de la aplicación, descrita mediante los modelos WCOMM, del despliegue de la misma en nodos e hilos, realizado con FraCC, ya que es posible utilizar varios despliegues sin necesidad de modificar la arquitectura de la aplicación.

La segunda mejora requirió la modificación de la región de envío de mensajes para que el usuario pueda elegir, en el modelo de despliegue, cuántas regiones de este tipo tiene el componente y qué puertos controla cada una de ella. De esta forma se mejora el uso de los recursos del sistema, puesto que no todos los mensajes tienen los mismos requisitos temporales.

La última mejora introduce una capa intermedia que separa el código (actividad) que se ejecuta en los estados del autómata temporizado de la propia estructura de FraCC. En el esqueleto de código C++ el usuario debe codificar un *paso del algoritmo* que quiere ejecutar. Nunca debe añadir bucles infinitos porque dejaría bloqueadas al resto de actividades asignadas al hilo en que se ejecuta. El código del usuario tiene por supuesto restricciones: no puede crear ni destruir hilos, mutex, semáforos ni ningún otro elemento que genere unidades de ejecución concurrente nuevas ni elementos de sincronización que puedan bloquear la actividad, puesto que la concurrencia es completamente gestionada en FraCC, no por el usuario. Esta mejora permite no solo el desarrollo del código de manera totalmente aislada del modelado de la aplicación con WCOMM, sino que además permite la reutilización de algoritmos ya existentes. El código de la actividad se compila como una librería dinámica de Linux, que es posteriormente referenciada desde el modelo de despliegue (figura 3-D). De esta manera el código de la actividad se *enlaza* al modelo, no está *embebido* en él, con la gran ventaja de que pueden ser desarrolladas y evolucionar independientemente.

Esta es otra de las grandes diferencias que existen entre FraCC y otros enfoques similares: no se genera todo el código de implementación de la aplicación a partir de los modelos de entrada, sino que por un lado se instancian las clases de FraCC para crear los componentes WCOMM y por otro se crean los esqueletos de las clases C++ que el usuario completará posteriormente.

4.2. Generación de modelos de análisis en Cheddar

La forma en que están implementados los componentes en FraCC, la estructura de los buffers y la organización de la ejecución de las regiones en hilos permiten la generación del modelo de análisis Cheddar mediante una transformación modelo-a-texto (figura 3-E). El modelo de despliegue FraCC define de forma explícita los hilos (*Threads*) en que se va a desplegar la aplicación. Estos hilos se corresponden directamente con las tareas (*Tasks*) de Cheddar. La transformación calcula el periodo de cada uno de estos hilos como el mínimo común divisor de todos los periodos de todas las regiones asignadas al hilo. Cada hilo se implementa en FraCC como un ejecutivo cíclico, lo que permite asignar regiones con distintos períodos al mismo hilo.

En cambio, los recursos compartidos (*Resources* en Cheddar) no están definidos explícitamente en el modelo de despliegue, pero es posible determinar cuáles pueden ser potencialmente compartidos a partir de la implementación de FraCC. Estos recursos potencialmente compartidos son buffers, que almacenan los mensajes de entrada, salida y eventos. Estos buffers son accedidos únicamente por las actividades programadas por el usuario, las actividades de gestión de las regiones y de los puertos, ambas predefinidas en FraCC. Para determinar si un buffer es finalmente compartido por más de un hilo, y por tanto tiene que ser protegido frente al acceso concurrente, es necesario recorrer el modelo de la aplicación WCOMM y el de despliegue FraCC, buscando si las regiones que acceden a un mismo recurso están asignadas al mismo hilo o no. En caso negativo, dicho recurso deberá ser protegido, y en el modelo Cheddar se añadirá un elemento de tipo *Resource*.

Para poder realizar el análisis de planificabilidad es necesario que el desarrollador estime el peor tiempo de ejecución de las actividades contenidas en los estados, así cómo los periodos de las actividades periódicas y los tiempos mínimos de activación de las esporádicas. A partir de estos datos es posible obtener los datos que faltan en el análisis: el tiempo de cómputo de una región se calcula como el peor tiempo de todas las actividades que ejecuta, mientras que el periodo del hilo se calcula como el máximo común divisor de todos los periodos de las actividades de las regiones asignadas al hilo. También se comprueba que el periodo calculado para el hilo sea mayor que el mayor tiempo de cómputo de todas las actividades de las regiones asignadas a él. Es un análisis muy pesimista y existe mucho margen para mejorar la estimación, ya que por ejemplo la región no está siempre ejecutando el estado con el tiempo de cómputo mayor.

La separación entre arquitectura (modelo WCOMM) y despliegue (modelo FraCC) permite que el desarrollador de aplicaciones genere, analice y pruebe distintos escenarios de despliegue para la misma aplicación, tanto en nodos como en hilos, sin tener que modificar su arquitectura.

5. Diseño de la arquitectura de control de AEGIR

En esta sección se muestra una visión general del proceso de diseño implícito en C-Forge que se presentó en la sección anterior, detallando a la vez los conceptos de implementación más importantes de la aplicación de control para AEGIR. Así, partiendo del modelado de la arquitectura de componentes que componen la aplicación, se detalla cómo se diseña el comportamiento interno de cada componente utilizando como ejemplo uno de los más complejos en cuanto a comportamiento concurrente. Por otra parte, para ilustrar la inclusión de algoritmos en los componentes, se explica el diseño del neuro-controlador adaptativo para evitación de obstáculos. Se finaliza con un ejemplo práctico de despliegue flexible en nodos, procesos e hilos y el análisis de planificabilidad con Cheddar.

5.1. Modelo arquitectónico de la aplicación

Teniendo en cuenta la típica configuración por capas de las arquitectura híbridas referenciadas en el estado de la técnica y los requisitos funcionales del sistema AEGIR, se proponen los componentes que integran la vista arquitectónica mostrada en la Figura 4. Se utiliza para ello el editor gráfico que se mencionaba en la Figura 3-A. De arriba abajo, los componentes se distribuyen en capas de interfaz, deliberativa, ejecutiva y de comportamiento, respectivamente. La conexión entre componentes se realiza por medio de los puertos (de entrada o salida) y conectores por los cuales se intercambiarán mensajes asíncronos sin respuesta.

Interfaz de usuario: El componente *C_UserInterface* recibe comandos de usuario y los envía a *C_Mission-Planner.* También recoge información de este último para ofrecerla al operador, incluyendo la información de más bajo nivel, como los datos recogidos por los sensores.

Capa deliberativa: En este nivel se localizan los dos componentes de planificación: *C_MissionPlanner*, que interpreta los comandos de misión para construir un plan de tareas secuencial y *C_PathPlanner,* encargado de planificar la ruta como una lista de puntos de paso desde el origen hasta el destino deseado.

Capa ejecutiva y supervisora*:* Incluye el componente *C_HealthMonitor,* que monitoriza el estado operativo del resto de subsistemas, por ejemplo, recogiendo *watchdogs* de los principales componentes. El componente ejecutivo *C_MissionSequencer* es el encargado de interpretar las misiones enviadas por *C_MissionPlanner* y secuenciarlas en tareas que

serán remitidas a las capas inferiores. En concreto, le transmite la ruta calculada por *C_PathPlanner* a *C_PathTracker* o *C_Maneuver* en la capa inferior. Por su necesidad de incluir múltiples comportamientos basados en autómatas temporizados concurrentes, el componente *C_MissionSequencer* es uno de los más interesantes para tomarlo como ejemplo de modelado de comportamiento interno en la sección 5.2.

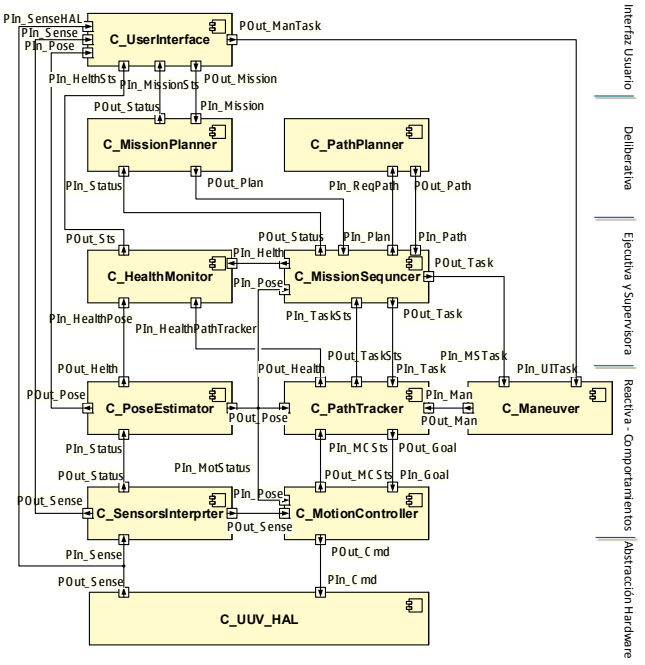

Figura 4. Componentes de una arquitectura híbrida para AEGIR.

Capa reactiva - comportamientos: En esta capa se incluyen los componentes reactivos para control de movimiento, percepción y localización. Aquí se incluirían los típicos componentes de arquitecturas basadas en comportamientos de los AUVs (Carreras et al 2005) como son navegación local (mover el vehículo a lo largo de una ruta local punto a punto); navegación autónoma (generación de rutas libres de colisión entre la posición actual y la meta), navegación reactiva (evasión de obstáculos), seguimiento (seguir un objeto de interés o una secuencia de puntos dados), deambular en un cierto margen o zona preestablecidas, y flotación (mantiene la posición del vehículo). Puede incluir fusión de mecanismos o selección/activación de comportamientos. En la implementación de estos comportamientos intervendrán varios componentes que en la arquitectura para AEGIR se han dispuesto en dos sub-niveles, el de guiado y localización como nivel superior y el de comportamientos de control, evitación de obstáculos e interpretación de datos de sensores como nivel inferior en contacto directo con la capa de abstracción del hardware.

El sub-nivel superior incluye *C_PathTracker*, que recibe una lista de puntos de paso de la ruta (*waypoints*) y realiza el seguimiento fusionando también los comandos enviados por *C_Maneuver*. Este último componente recibe comandos simples de velocidad, altitud, profundidad, mantenimiento de orientación y también las órdenes de control manual de *C_User_Interface*. *C_PathTracker* también implementa políticas de prioridad para combinar comportamientos. El componente de localización *C_PoseEstimator* es el encargado de fusionar toda la información sensorial disponible para estimar la posición del vehículo. Recibe esta información del sub-nivel inferior, donde se sitúa el *C_SensorsInterpreter* que procesa datos de sensores (e.g., registrador sonar de velocidad Doppler, sensor de presión/profundidad, unidad inercial, etc.) para poder estimar la distancia al fondo marino, altitud, etc. También ofrece datos de sonar del fondo marino y los procesa y filtra para proporcionar la mejor información posible de marcas (*landmarks*) y obstáculos. Finalmente, el componente *C_MotionController* transforma las coordenadas espaciales y de velocidad en comando para los actuadores (propulsores). Para ello implementa un algoritmo complejo, el neuro-controlador adaptativo para evitación de obstáculos, que se explicará con detalle en la sección 5.3.

Siguiendo el proceso iterativo de desarrollo implícito en C-Forge, el modelado inicial de la arquitectura de la aplicación se puede proponer tal cual se ha mostrado en la Figura 4. Es decir, se pueden proponer los componentes mostrando sólo sus puertos. Esta visión arquitectónica nos lleva a la necesidad de definir los tipos de datos y mensajes que se intercambian los componentes. Éstos a su vez se agrupan en interfaces que pueden ser reutilizados por varios componentes. Para esta tarea se utiliza la herramienta textual que incorpora C-Forge. Un ejemplo se muestra en la Figura 5, con la definición de parte de las interfaces de *C_MissionSequencer*. En este momento se está ya en condiciones de refinar el diseño arquitectónico mediante el modelado del comportamiento del componente.

```
import "basicDatatypes.wida", "userDatatypes.wida" ;
package auvb1.interfaces;

interface I_PlanCmd {
    message setPlan (int32 numSteps, plan missionPlan);
    message abort ();
    message pause ();
    message resume ();
    }

interface I_MissionSts {
    message misSts (planSts sts);
    message misEnd ();
}

interface I_PathPlan_Plug {
    message pathReq (pose start, pose end);
    }
interface I_PathPlan_Socket {
    message path (int32 numWaypoints, path waypointList);
}
```

Figura 5. Extracto de definición de interfaces.

5.2. *Modelado del comportamiento del componente*

Para el modelado de un componente de la aplicación, como el *C_MissionSequencer*, se utiliza otra de las herramienta gráficas de C-Forge mostradas en la Figura 3-A. Como se observa a la derecha de la Figura 6, el diseñador cuenta con una paleta gráfica que le permite definir los puertos del componente, junto con las regiones, estados, transiciones, eventos y relojes. El usuario debe añadir en cada puerto las interfaces, de entre las modeladas en la fase anterior, que definen los mensajes que intercambia el componente por dicho puerto.

Al ser un componente ejecutivo y de monitorización de tareas, *C_MissionSequencer* cuenta con numerosos puertos de interacción con otros componentes. A su vez contiene las tres regiones ortogonales que se muestran en la Figura 6.

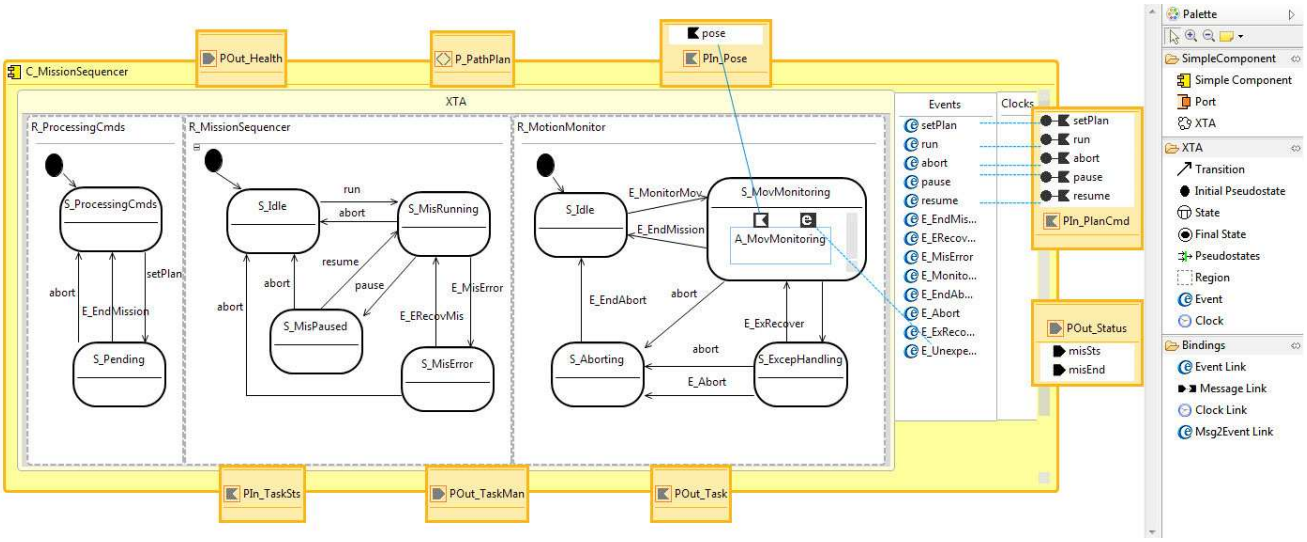

Figura 6. Componente *C_MissionSequencer* con tres regiones concurrentes. Sólo se muestran mensajes de tres puertos y actividad de un estado.

Estas regiones procesan en paralelo las tres actividades que debe realizar el componente: (1) *R_ProcessingCmds,* donde se interpretan y procesan los comandos de misión que lleguen de los niveles superiores, reportando a la vez sobre los estados de capas inferiores. (2) *R_MissionSequencer,* donde se secuencian los comandos interpretados, ordenando la ejecución de cada tarea. Esta actividad se realizará en el estado *S_Running*. Por ejemplo, ir a un punto implicará recabar la ruta de *C_PathPlanner*, que devolverá una lista de puntos de paso, que a su vez se enviarán a *C_PathTracker*. Como se observa en la región *R_MissionSequencer* de la Figura 6, una misión puede entrar en pausa o ser abortada por orden del operador, o en estado de error por la generación del evento interno *E_Error* (3) *R_MotionMonitor,* donde se monitoriza el movimiento del vehículo, en su evolución normal o en estados de excepción. Cuando se inicia un movimiento, llegará el evento *E_MotionMonitor* a esta región generada por la actividad del estado *S_Running*. Se pasa así al estado de monitorización de la ruta seguida. Si el vehículo se desvía demasiado, por ejemplo, por un obstáculo insalvable, *E_Unexpected* hace transicionar a *S_ExcepHandling*, donde se volverá a pedir recalcular la ruta a *C_PathPlanner*. Se volverá al estado anterior *S_Monitoring*, una vez se haya enviado la nueva ruta a *C_PathTracker*. Si llega el mensaje de *abort* para abortar la misión, se realizarán las órdenes necesarias en *S_Abort* para que el UUV quede en estado seguro. Una vez hecho esto, *E_Done* mandará transicionar a *S_Idle.*

El comportamiento del resto de componentes se ha completado de la misma forma. Una vez diseñados los autómatas temporizados, hay que incluir las actividades que se realizan en cada estado, aunque teniendo en cuenta que no todos los estados ejecutan una actividad. Estas actividades también se modelan utilizando el editor textual con el que se definieron los mensajes y los tipos de datos. Más concretamente se modela la "carcasa" de la actividad, que será visible en el modelo gráfico, con pines de entrada y salida de mensajes y eventos. Estos pines se enlazan con los mensajes en los puertos y los eventos que correspondan (ver Figura 6, dentro de *S_MovMonitoring*). A partir de este modelo de actividad, una transformación genera el esqueleto de código que es rellenado por el usuario (Figura 3-C) y posteriormente compilado en una librería dinámica que será cargada cuando se ejecuta la aplicación.

5.3. Implementación de algoritmos

Para ilustrar el proceso de inclusión de algoritmos en actividades, se muestra en este apartado uno de los componentes más complejos, el neuro-controlador adaptativo para control del movimiento y evitación de obstáculos embebido en el componente *C_MotionController*. El usuario completa cuatro métodos, *init(), onEntry(), onExit(), doCode(),* (ver Figura 7), que serán ejecutados por el framework en el estado al que se asocia la actividad. El método *doCode()* es el principal de la actividad, y debe codificar un *paso del algoritmo* que se ejecuta con las restricciones descritas en la sección 4.1. En la Figura 7 se muestra un extracto de dicho código, en el que se pueden observar los métodos mencionados y se detalla únicamente el *doCode()*. En él se muestra cómo se accede a los mensajes de los pines de entrada a la actividad (*inMsg*) y se envían mensajes por los de salida (*outMsg*). La definición de *AegirMsg* los genera automáticamente la herramienta. El usuario invoca los métodos *get_XX* y *set_XX* para acceder a la información que transportan los mensaje y en este caso, invoca los métodos (*updateGoal, updateMeasurements*) de la clase que encapsula el algoritmo neuro-controlador que se explica a continuación.

```
#include "A_MotionController.h"
#include "../neuronalController.h"
        // defines 'updateGoal' and 'updateMeasurements'
void A_MotionController::init(){ ... }
void A_MotionController::onEntry(){ ... }

void A_MotionController::doCode(){
    AegirMsg::Goal inMsg_goal;
    AegirMsg::Pose inMsg_pose;
    AegirMsg::Sense inMsg_sense;
    AegirMsg::Status outMsg_sts;

    if(get_Goal(inMsg_goal))
        updateGoal(inMsg_goal);

    if(get_Pose(inMsg_pose) && get_Sense(inMsg_sense))
        updateMeasurements(inMsg_pose, inMsg_sense);

    set_Status(outMsg_sts)
}

void A_MotionController::onExit(){ ... }
```

Figura 7. Extracto de código simplificado para una Actividad.

En la figura 8 se muestran el sistema de control de movimiento bio-inspirado compuesto por una red neuronal de mapeo de dirección auto organizativa (SODMN-*Self-Organization Direction Mapping Network*) y una red neuronal para el comportamiento de evasión de obstáculos (NNAB- *Neural Network for the Obstacle Avoidance Behaviour*), ambas de inspiración biológicas.

La SODMN es un neuro-controlador adaptativo cinemático y una red neuronal no supervisada en tiempo-real que aprende a controlar submarinos autónomos y vehículos de superficie en un entorno no estacionario. La SODMN combina un aprendizaje asociativo y un mapa asociativo vectorial (VAM, Vector Associative Map) (García-Córdova 2007, Guerrero-González et al 2010) para generar transformaciones entre coordinadas espaciales y coordenadas de velocidad motoras (Guerrero-González et al 2011). Las transformaciones se aprenden en una fase de entrenamiento no supervisado, durante el cual el vehículo submarino se mueve como resultado de las velocidades seleccionadas al azar de sus actuadores. El neuro-controlador aprende la relación entre estas velocidades y los movimientos incrementales resultantes.

La NNAB está basada en una forma de aprendizaje animal conocida como el *condicionamiento operante*. El aprendizaje no requiere supervisión y tiene lugar en un entorno lleno de obstáculos. La NNAB aprende a controlar esa evasión de obstáculos en un vehículo autónomo generando pequeñas desviaciones de la ruta hacia la meta para evadir el obstáculo y cuando está libre de éste se corrige la desviación por la SODMN hacia la meta. Ambas redes están fuertemente acopladas para el control del movimiento, por eso el comportamiento de evasión de obstáculos no se implementa en un componente diferente, sino que se integra en el mismo *C_MotionController*.

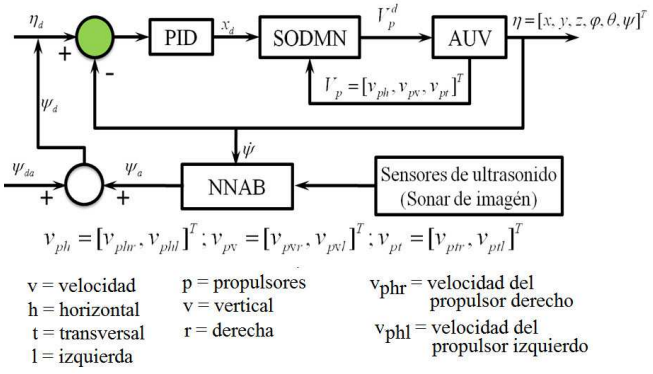

Figura 8. Esquema neuronal para el control de movimiento (SODMN) y la evasión de obstáculos reactiva (NNAB).

5.4. Despliegue flexible y análisis temporal con Cheddar

El despliegue del software de control de AEGIR se realiza en dos nodos: un PC en la estación de teleoperación que ejecuta los componentes *C_UserInterface* y *C_MissionPlanner*, y un *embedded-PC* embarcado en el UUV en el que se ejecutan el resto de componentes. A la hora de realizar la distribución en procesos de los componentes de la aplicación, en otros modelos de componentes es habitual que cada componente se ejecute en su propio proceso, sobre todo en aquellas herramientas en las que los componentes son binarios. Con el diseño de FraCC hemos querido añadir un grado más de flexibilidad al despliegue, permitiendo al usuario elegir el número de procesos e hilos en que quiere ejecutar la aplicación. En este apartado se analiza el despliegue de la aplicación de AEGIR atendiendo a los siguientes criterios:

- Todos los componentes que forman la capa reactiva se agrupan en un mismo proceso para evitar una sobrecarga excesiva en las comunicaciones.
- El componente *C_HealthMonitor* se ejecuta en su propio proceso para asegurar su integridad, y de esta manera poder supervisar la ejecución de la aplicación.
- Los componentes restantes, *C_MissionSequencer* y *C_PathPlanner*, se ejecutan en su propio proceso.

Sólo los componentes de la capa reactiva, seis en total, tienen restricciones de tiempo real. Se han agrupado en un solo proceso, para el cual se hace el análisis de planificabilidad con Cheddar. La comunicación entre procesos se realiza por actualmente por sockets, y dado que Cheddar no soporta análisis de sistemas distribuidos, sólo se realiza el análisis de los procesos con restricciones de tiempo-real.

El comportamiento de todos los componentes de esta capa se modela mediante una única región, salvo *C_PoseEstimator* y *C_UUV_HAL* que requieren dos regiones ortogonales. Además, FraCC añade una región de gestión del envío de mensajes por cada componente, lo cual arroja un total de catorce regiones, que deben ser asignadas a hilos. Atendiendo a las restricciones temporales de la aplicación, se decidió crear un total de cinco hilos: gestión del envío de mensajes de todos los componentes, ejecución de las regiones del componente *C_UUV_HAL*, ejecución de las regiones de los componentes *C_PoseEstimator* y *C_SensorsInterpreter*, ejecución de las regiones de los componentes *C_PathTracker* y *C_Maneuver*, y el quinto hilo para el componente *C_MotionController*.

Este despliegue se realiza fácilmente con la herramienta gráfica que incorpora C-Forge. La flexibilidad de la misma permite la rápida modificación de la distribución de las regiones concurrentes en diferentes *hilos*, simplemente moviendo una línea del modelo. En la Figura 9 se muestra un fragmento de este despliegue, sobre el que se realiza el análisis para pre-verificar si se cumplen los plazos de respuesta requeridos. Como muestra la figura 10, el despliegue anterior resulta ser planificable. Si no lo fuera, se podría modificar fácilmente, moviendo regiones entre hilos o incluso eliminando o creando nuevos hilos.

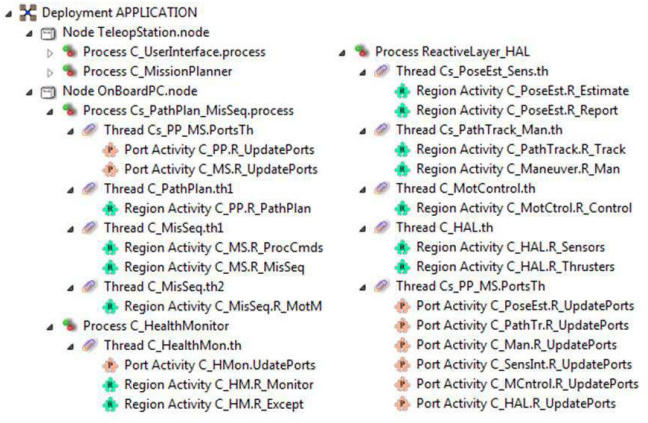

Figura 9. Modelo de despliegue de AEGIR. Se muestran dos nodos y los hilos desplegados dentro de los tres procesos en el PC embarcado.

6. Pruebas y Resultados

En esta sección se resumen algunos de los resultados obtenidos con el AEGIR-UUV, tanto en la realización de unas pruebas en la costa de la Región de Murcia como los resultados del neuro-controlador bio-inspirado.

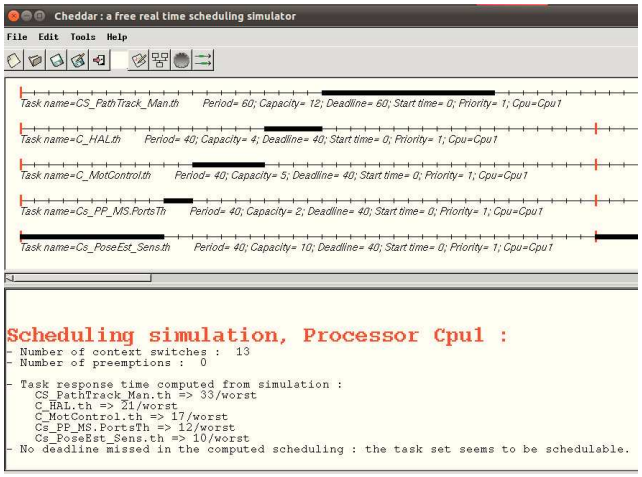

Figura 10. Análisis de planificabilidad del despliegue de la Figura 9.

6.1 Misiones oceanográficas en la costa de la Región de Murcia

Con el fin de ofrecer una amplia gama de capacidades de investigación oceanográfica, AEGIR está equipado con varios tipos de instrumentos de carácter oceanográfico (Guerrero-González et al 2011, Pérez-Ruzafa et al. 2005, González et al 2012). Se han realizado por el momento misiones de aguas poco profundas (laguna costera del Mar Menor) y aguas adyacentes del Mediterráneo cuya localización se muestra en la figura 11.

Figura 11. Mapa que representa las dos áreas de investigación. Vista aérea de la Laguna del Mar Menor y Cabo Tiñoso, en Cartagena-Murcia, España.

El Mar Menor es una laguna costera hipersalina ubicada el sureste de la Región de Murcia, en España. La más grande de la península Ibérica y una de las más grandes de Europa. Sus características ecológicas hacen de la laguna un paraje natural único. Con una profundidad máxima de 6 m (media de 4.4m), una superficie de 135 km^2, embalsa alrededor de 580 Hm^3 de agua con una salinidad entre 42-49 P.S.U.. La laguna es un lugar emblemático regional donde se solapan numerosos usos como agricultura intensiva en la cuenca hidrográfica, turismo con fuerte presencia del sector naval y, pesca (cuyas capturas adquieren un precio muy superior al equivalente del mediterráneo). Todos estos usos de la laguna han producido interferencias con los procesos naturales teniendo que tomarse medidas de gestión para asegurar la explotación sostenible. En este sentido la monitorización de diferentes parámetros ambientales juega un papel decisivo en el conocimiento de la laguna y su funcionamiento. El uso de vehículos autónomos submarinos se presenta como una herramienta altamente eficaz para los estudios científicos sobre los que basar las decisiones de gestión del medio natural. En este sentido, las misiones del AEGIR están diseñadas para obtener datos tanto de calidad de aguas como del sustrato. Se desarrollaron en este entorno tres misiones diferentes empleando distintos tipos de sensores: 1) *Monitoreo de calidad del agua*: se utilizaron diferentes sensores, bien por separado o conjuntamente, como una sonda multiparamétrica YSI® (para medida de parámetros ambientales como temperatura, salinidad, pH, turbidez, clorofila y oxígeno disuelto.), un espectrómetro de ultravioletas SUNA® para el análisis en continuo de nitratos y un fluorómetro de inducción-relajación para la medida de parámetros de fotosíntesis. La medida de la dirección y velocidad de corrientes la proporciona el *Doppler velocity logger* (DVL) utilizado para la navegación del AEGIR.

2) *Batimetría y cartografía del fondo*: mediante el altímetro y sensor de presión instalados en el AEGIR se determinó una primera aproximación a la batimetría que se completa con las medidas de alta resolución obtenidas mediante un escáner sonar de barrido lateral de TRITECH®. Para la elaboración de mapas de vegetación sumergidas se utilizaron, junto con las imágenes de sonar de barrito lateral las de vídeo ambas geo-referenciadas que permiten determinar la densidad y tipo de fondo, tanto en su vegetación como fauna asociada. 3) *Oceanografía*: un aspecto esencial en todos los estudios ecológicos y ambientales es la hidrografía que permite conocer el movimiento de agua mediante las corrientes. Para ello se realizan modelos hidrodinámicos tridimensionales que se fuerzan con parámetros atmosféricos y/o elevación del nivel del mar, además de los resultados de otros modelos de escala regional. Para la fiabilidad de los modelos resulta imprescindible una correcta validación del modelo. Los vehículos autónomos submarinos son una herramienta que permite validar espacialmente los parámetros predichos por los modelos sin necesidad de multiplicar el número de sensores. De particular interés para el Mar Menor es el conocimiento del intercambio de agua con el Mediterráneo. Para ello se realizó un experimento multivehículo con el fin de determinar el área de influencia del agua de la laguna en le Mediterráneo y viceversa (https://sites.google.com/site/auvexperiment2011/).

Figura 12. Muestras de salinidad en la ruta del AEGIR-UPCT en el Mar Mediterráneo adyacente a la Laguna del Mar menor.

La figura 12 muestra, a modo de ejemplo, algunas de las mediciones de la salinidad del Mar Mediterráneo sobre una de las trayectorias del vehículo submarino a una profundidad de 1,5 m. El rango de salinidad varió entre 37,54 y 37,60 P.S.U.

Para verificar la fiabilidad del sistema de control se llevaron a cabo pruebas de navegación que contemplaban la flotabilidad, estabilidad direccional, elevación y alabeo, inmersiones, fiabilidad de sensorización, así como el control de velocidad de los propulsores. En todos los casos hemos sido capaces de verificar la respuesta correcta a las solicitudes del operador del vehículo, que interacciona con el mismo a través de la interfaz que se muestra en la figura 13. Este sistema de monitorización permite la visualización y el registro de las imágenes de las cámaras de visión, el sonar de imagen, la cartografía de navegación, los niveles de energías de las baterías, la posición y orientación del vehículo, la profundidad operativa, la unidad inercial, la actividad y niveles de consumo de los motores propulsores, la velocidad de navegación y varias alarmas de seguridad del vehículo.

En una de las pruebas de navegación de alrededor de una hora de duración la carga de la batería no mostró signos de agotamiento. También se puede verificar la exactitud de medición del desplazamiento total del vehículo sumergido, un hecho que es esencial para ser capaz de determinar correctamente el lastre en cada operación futura.

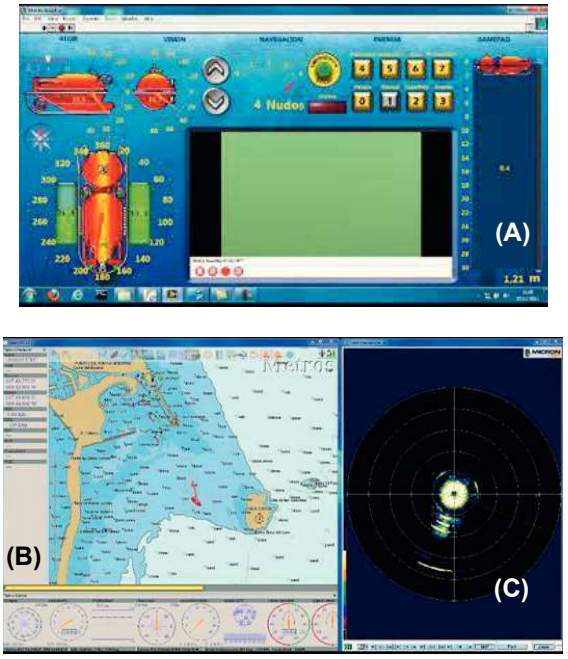

Figura 13. Sistema de monitorización del AEGIR-UPCT. (a) Sistema de control, (b) cartografía de navegación, y (c) imágenes de obstáculos.

6.2 *Comportamiento del sistema de control basado en red neuronal*

La arquitectura neuronal propuesta para el sistema de navegación autónomo del vehículo submarino descrita en la sección 4.2 es capaz de generar trayectorias óptimas para vehículos submarinos o de superficie en un entorno arbitrariamente variable. El espacio de estado con el cual se trabaja es el espacio cartesiano del robot submarino. Las Figuras 15 y 16 muestran las prestaciones del sistema de control de navegación autónomo en las pruebas realizadas en la piscina de del Parque Industrial de Fuente Álamo, en la Región de Murcia. Se realizaron en un espacio de trabajo en 3-D sin ningún obstáculo, con una posición inicial P_0 $(x, y, z) = (1, 1, 1)$ m y una orientación inicial tal como se muestran en las figuras 15 y 16 de la forma $O_0(\varphi_0, \theta_0, \psi_0)$. En la Figura 15, AEGIR genera una trayectoria de alcance que va de P0→T1→T2→T3, sin tener como referencia la trayectoria lineal que forman estos puntos. Además en esta figura se muestra la orientación y la velocidad que adquiere el vehículo al alcance de estos objetivos. El error de trayectoria, mostrado en la figura 14, es el que se obtiene de la trayectoria lineal que se forma con los puntos (P0, T1, T2, yT3) y la trayectoria generada por el propio neuro-controlador para alcanzar dichos puntos.

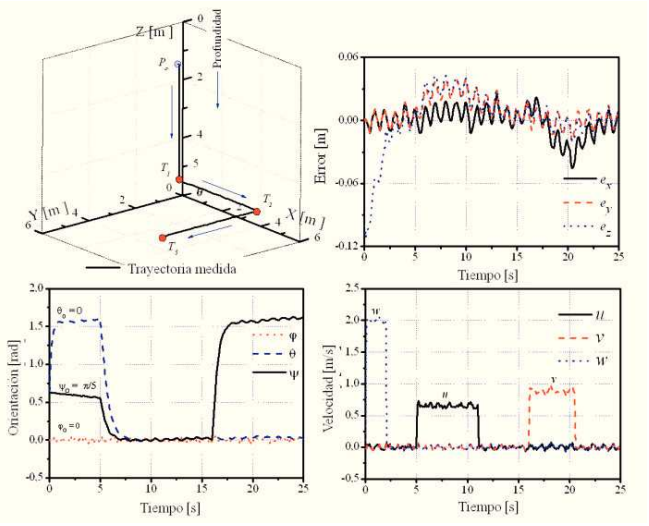

Figura 14. Alcance de objetivos de P0→T1→T2→T3, siendo P0 (1,1,1)m, T1(1,1,5)m, T2(5,1,5)m y T3(5,5,5)m.

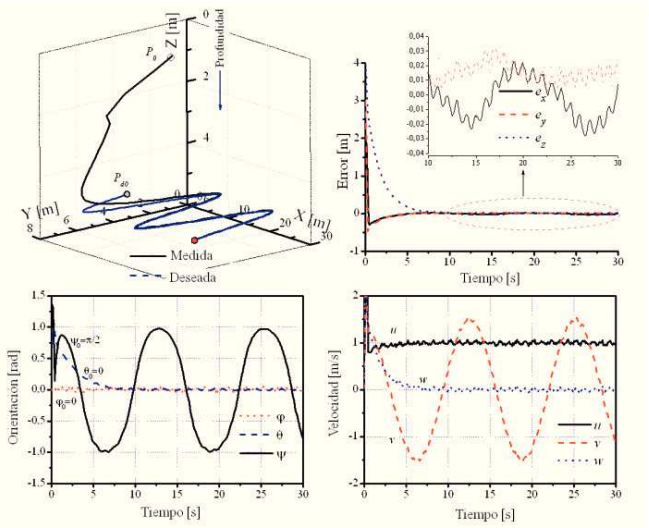

Figura 15. Control de seguimiento de una trayectoria deseada.

En la Figura 15, se muestra el control de seguimiento del vehículo submarino para una trayectoria sinodal que parte de la posición de referencia $P_{0r}(6,2,6)$, mientras que el vehículo parte de una posición $P_0(1,1,1)$ y orientación $O_0(0,0, \pi/2)$ inicial. Además, en esta figura se muestra el error de seguimiento, la orientación y la velocidad que adquiere el vehículo al seguimiento de la trayectoria.

7. Conclusiones y trabajos futuros

En este artículo se ha presentado el desarrollo del AEGIR en el contexto del proyecto MISSION-SICUVA, como una plataforma para la investigación oceanográfica y como un banco de pruebas para la implementación de estrategias de control y misiones oceanográficas en el área del Mar Menor con el objetivo de medir la calidad del agua, batimetría de alta resolución del fondo marino y generación de mapas oceanográficos. En el desarrollo del software de control se ha utilizado la cadena de herramientas C-Forge, que soporta el enfoque de desarrollo basado en componentes gracias a la infraestructura de diseño dirigido por modelos integrada en Eclipse. C-Forge es una cadena de herramientas abierta, que actualmente está constituida por un lenguaje de modelado de componentes, WCOMM, que modela componentes como caja blanca en los que el comportamiento se describe mediante autómatas temporizados, y un framework basado en componentes, denominado FraCC, que proporciona el soporte de ejecución necesario para ejecutar las aplicaciones modeladas con WCOMM.

Como características más reseñables de FraCC, cabe destacar la separación entre arquitectura y algoritmia, y entre arquitectura y despliegue. La primera ha permitido separar responsabilidades y reutilizar parte del código ya desarrollado por el LVS. Esta característica aporta como valor añadido el bajo acoplamiento que existe entre componentes y la independencia entre modelos y código, ya que el código de las actividades se enlaza con los modelos, no está embebido en ellos, y de esta manera pueden ambos evolucionar de forma independiente. La separación entre arquitectura y despliegue permite que el desarrollador de aplicaciones genere, analice y pruebe distintos escenarios de despliegue de la aplicación, tanto en nodos como en hilos concurrentes, sin tener que modificar su arquitectura. La pre-verificación temporal realizada con Cheddar proporciona seguridad al usuario de que las restricciones temporales van a ser satisfechas. Estos son, a nuestro juicio, los puntos más fuertes de FraCC.

Estos aspectos tienen especial interés en AEGIR debido al proceso de diseño incremental seguido en su diseño. Desde el principio era importante prototipar componentes simples que pudieran ser mejorados posteriormente y fácilmente modificados y ampliados no solo en su estructura interna sino también en su despliegue en distintos nodos computacionales. Como consecuencia inmediata del enfoque propuesto es conveniente destacar que los tiempos de desarrollo de los prototipos decrecieron sustancialmente, permitiendo a su vez el crecimiento progresivo de la funcionalidad de las aplicaciones. Cualitativamente, se ha comprobado que tanto el proceso de desarrollo como la integración de las distintas partes del código de la aplicación ha mejorado sustancialmente con el uso de C-Forge.

Finalmente, en la sección de resultados, se han presentado las pruebas realizadas con el vehículo en misiones en aguas costeras. La adquisición de datos con las sondas de calidad de agua, batimetría y de validación de modelos hidrodinámicos ofrecieron resultados satisfactorios y demostraron la validez de la integración de algoritmos de interpretación de datos en la arquitectura software propuesta. Las pruebas de navegación y evitación de obstáculos también resultaron exitosas, demostrando la validez de la arquitectura inspirada neurobiológicamente para el componente de control de navegación de AEGIR.

Como conclusión de los resultados obtenidos, se puede confirmar la validez de la plataforma, con su control neuronal, adquisición de datos y control de ejecución de misiones integrados en una arquitectura software basada en componentes, como vehículo multidisciplinar para investigación oceanográfica.

En cuanto a las líneas de trabajo futuro, actualmente estamos trabajando en la mejora del uso de la cadena de herramientas, la integración con ROS, la mejora del análisis de planificabilidad con el estudio de la influencia de la comunicación entre procesos, el uso de mecanismos de memoria compartida entre procesos en un mismo nodo, o la integración de middleware de terceros, como ICE, ACE o YARP, entre otros. En AEGIR se está trabajando en la mejora de los algoritmos de evitación de obstáculos con objetos móviles, así como en mejorar la reconstrucción de imágenes subacuáticas a partir de las medidas de un sonar lateral y diferentes configuraciones de reparto de la funcionalidad entre el vehículo de superficie y el submarino.

English Summary

Design of the control software of a UUV for oceanographic monitoring using a component model and framework with flexible deployment.

Abstract

Unmanned Underwater Vehicles (UUVs) explore different habitats with a view to protecting and managing them. They are developed to overcome scientific challenges and the engineering problems caused by the unstructured and hazardous underwater environment in which they operate. Their development bears the same difficulties as the rest of service robots (hardware heterogeneity, sensor uncertainty, software complexity, etc.) as well as other particular from the domain, like the underwater environment, energy constraints, and autonomy. This article describes the AEGIR UUV, used as a test bed for implementation of control strategies and oceanographic mission in the Mar Menor area in Spain, which is one of the largest coastal lagoons in Europe. It also describes the development of a tool chain that follows a model-driven approach, which has been used in the design of the vehicle control software as well as a component-based framework that provides the runtime support of the application and enables its flexible deployment in nodes, processes and threads and pre-verification of concurrent behavior.

Keywords:

UUV Unmanned Underwater Vehicle, oceanographic monitoring, component framework, component model, deployment, concurrency analysis.

Agradecimientos

Este trabajo ha sido parcialmente financiado por el proyecto financiado por la CICYT del Gobierno Español DIVISAMOS (ref. DPI2009-14744-C03-02) y ViSelTR (ref. TIN2012-39279), así como por el proyecto financiado por la Fundación Séneca de la Región de Murcia MISSION-SICUVA (ref. 15374/PI/10) y el proyecto "Coastal Monitoring System for the Mar Menor Coastal Lagoon (PEPLAN 463.02-08 CLUSTER de la Región de Murcia. Francisco Sánchez Ledesma agradece la financiación recibida por parte del programa de becas FPU del MEC (beca AP2009-5083). Por último, los autores quieren agradecer también a la Armada Española la cesión del vehículo UUV y su posterior ayuda en su reconstrucción.

Referencias

Alonso, D., Pastor, J., Sánchez, P., Álvarez, B., Vicente-Chicote, C., 2012. Generación Automática de Software para Sistemas de Tiempo Real: Un Enfoque basado en Componentes, Modelos y Frameworks. Revista Iberoamericana de Automática e Informática Industrial RIAI. Vol, 9, Num. 2, págs 170–181, doi: 10.1016/j.riai.2012.02.010.

Alonso, D., Vicente-Chicote, C., Ortiz, F., Pastor, J., Álvarez, B., 2010. V3CMM: a 3-View Component Meta-Model for Model-Driven Robotic Software Development. Journal of Software Engineering for Robotics, Vol.1, no 1, pp. 3-17.

Antonelli, G., Chiaverini, S., Sarkar,N., West, M., 2001. Adaptive control of an autonomous underwater vehicle: experimental results on ODIN, IEEE Trans Control Syst. Technol., vol. 9, Issue: 5, Sep. 2001, pp. 756-765.

Auke Jan Ijspeert, 2008. Central pattern generators for locomotion control in animals and robots: A review. Neural Networks, Volume 21 Issue 4, pp. 642-653.

Ben-Ari, M., 2008. Principles of the Spin Model Checker. Springer-Verlag.

Bengtsson, J., Yi, W., 2004. Timed Automata: Semantics, Algorithms and Tools. In: Lectures on concurrency and Petri nets, Springer-Verlag, vol. 3098, pp. 87-124.

Behrmann, G., Larsen, K., Moller, O., David, A., Pettersson, P., Wang, Y., 2001. UPPAAL - present and future. Proc. of the 40th IEEE Conf. on Decision and Control.

Bruyninckx, H., 2008. Robotics Software: The Future Should Be Open. In: IEEE Robotics & Automation Magazine, Vol. 15, No. 1, pp. 9-11.

Carreras, M., Yuh, J., Batlle, J., Ridao, P., 2005. A behavior-based scheme Using Reinforcement Learning for Autonomous Underwater Vehicles. In: IEEE Journal of Oceanic Engineering, Vol. 30, No. 2.

Chang, C., and Gaudiano, P., 1998. Application of biological learning theories to mobile robot avoidance and approach behaviors. J. Complex Systems, vol. 1, pp. 79–114.

Eickstedt, D.P., Sideleau, S.R., 2009. The backseat control architecture for autonomous robotic vehicles: A case study with the Iver2 AUV. In: OCEANS 2009, MTS/IEEE Biloxi - Marine Technology for Our Future: Global and Local Challenges. 1-8.

Fossen, T., 1994. Guidance and control of ocean vehicles. John Wiley and Sons Ltd.

Fujii, T., Arai, Y., Asama, H., and Endo, I., 1998. Multilayered reinforcement learning for complicated collision avoidance problems. In: Proceedings IEEE International Conference on Robotics and Automation, vol. 3, pp. 2186-2191, Leuven, Belgium.

García-Córdova, F., 2007. A cortical network for control of voluntary movements in a robot finger. In: Neurocomputing, vol. 71, 2007, pp. 374-391.

Gonzalez, J. et al, 2012. AUV Based Multi-vehicle Collaboration: Salinity Studies in Mar Menor Coastal Lagoon. In IFAC Workshop on Navigation, Guidance and Control of Underwater Vehicles (NGCUV).

Guerrero-González, A., García-Córdova, F., Ruz-Vila, F., 2010. A Solar Powered Autonomous Mobile Vehicle for Monitoring and Surveillance Missions of Long Duration. In: International Review of Electrical Engineering, Part A, vol. 5, n. 4, pp. 1580-1587.

Guerrero-González, A., García-Córdova, F., Gilabert, J., 2011. A Biologically inspired neural network for navigation with obstacle avoidance in autonomous underwater and surface vehicles. In: OCEANS 2011 IEEE. Doi. 10.1109/Oceans-Spain.2011.6003432

Medina, J., González-Harbour, M., and Drake, J., 2001. MAST real-time view: A graphic UML tool for modeling object-oriented real-time systems. Proc. of the 22nd IEEE Real-Time Systems Symposium (RTSS), December, pp. 245–256. IEEE.

Pérez-Ruzafa, A., Marcos, C., Gilabert, J., 2005. The Ecology of the Mar Menor coastal lagoon: a fast-changing ecosystem under human pressure. In: Coastal lagoons. Ecosystem processes and modelling for sustainable use and development. CRC press. pp.: 392-422.

Ridao, P., Yuh, J., Sugihara, K., Batlle, J., 2000. On AUV control architecture. In: Proc. Int. Conf. Robots and Systems.

Ritter, H., Martinez, T., Schulten, K., 1989. Topology-conserving maps for learning visuo-motor coordination, Neural Networks, vol. 2, 1989, pp. 159-168.

Schlegel, C., 2006. Communication patterns as key towards component-based robotics. In: International Journal on Advanced Robotics Systems 3 (1), 49–54.

Schlegel, C., Steck, C., Lotz, A., 2011. Model-driven software development in robotics: Communication patterns as key for a robotics component model, En: Introduction to Modern Robotics. iConcept Press (ed.on-line)

Stutters, L., Liu, H., Tiltman, C., Brown, D., 2008. Navigation technologies for autonomous underwater vehicles. In: IEEE Transactions on Systems, Man and Cybernetics, Part C: Applications and Reviews, Vol. 38, 581-589.

OMG, 2009. UML profile for MARTE: Modeling and analysis of real-time embedded systems, formal/2009-11-02.

RoSta: Robot Standards and Reference Architectures, Coordination Action (CA) funded under the European Union's Sixth Framework Programme (FP6), http://www.robot-standards.org/. Last accessed 05/2013.

Singhoff, F., Plantec, A., Dissaux, P., Legrand, J., 2009. Investigating the usability of real-time scheduling theory with the cheddar project. Journal of Real Time Systems, 43, 259–295.