Universidad de Las Palmas de Gran Canaria
Facultad de Informática
Departamento de Informática y Sistemas

Proyecto de Fin de Carrera

Sistema Integrado de Control para un Vehículo Submarino Autónomo

por

Enrique Fernández Perdomo

enrique.fernandez.perdomo@gmail.com

Tutor: Hernández Sosa, José Daniel
Co-Tutor: Cabrera Gámez, Jorge
Co-Tutor: Domínguez Brito, Antonio Carlos

Las Palmas de Gran Canaria, 20 de octubre de 2008

A mis padres
y a mi hermana Yolanda

Resumen

Este documento de Proyecto Fin de Carrera detalla el diseño y desarrollo de la especificación del equipamiento y las misiones que puede realizar un vehículo submarino autónomo. La aplicabilidad de estas especificaciones es extensible a un gran número de vehículos de exploración oceanográfica similares, como boyas, barcos, vehículos operados remotamente, etc.

Una vez definidas las características del vehículo y el alcance de la misión, se realiza un estudio de la arquitectura de múltiples sistemas embebidos en este tipo de vehículos y se propone un sistema, denominado *Sick***AUV**, para el cual se adopta una arquitectura híbrida basada en tareas, capaz de ejecutar misiones de exploración submarina. Se ha realizado un diseño en detalle y una implementación básica, mediante componentes CoolBOT, de varios elementos que permite mostrar algunos resultados representativos.

Varios Proyectos Fin de Carrera complementarios proporcionan herramientas de apoyo para la gestión de vehículos de exploración submarina:

- Un software gráfico facilita la elaboración de misiones y la especificación del equipamiento, así como la monitorización del estado del vehículo y los resultados parciales de la misión, durante su ejecución.
- Mediante una herramienta de simulación la misión puede realizarse con vehículos virtualizados que operarán en un entorno simulado. Además, permite que un vehículo use dispositivos virtualizados por el simulador, si éste no dispone de ellos físicamente.

El conjunto de todas estas herramientas ofrece un entorno software que cubre las diferentes fases de un experimento de exploración oceanográfica, desde la edición de la misión hasta su ejecución o simulación. También suponen un punto de partida en el análisis, diseño e implementación de las tecnologías aplicables al ámbito oceanográfico, con múltiples aplicaciones prácticas y posibles mejoras a corto y largo plazo.

Índice general

Agradecimientos

Mi más sincero agradecimiento va dirigido a los tutores de mi Proyecto Fin de Carrera. Ellos me han apoyado siempre, y la culminación del trabajo de dos largos años es mérito suyo. No sabría a quién de los tres agradecérselo más.

Durante el desarrollo de este proyecto ha habido momentos en los que parecía imposible avanzar. Sus consejos y directrices de trabajo han sido clave para filtrar el ingente flujo de información que se ha recopilado incesantemente. Siempre que me planteaba objetivos irrealizables o me extendía excesivamente en alguna fase, me devolvían a la realidad y renovaban mi ilusión en el proyecto. Eso también se lo agradezco.

También he de agradecer a Rodrigo Heredero Robayna y a Eliezer Ramírez Cabrera su colaboración en la realización de este proyecto. No sólo hemos compartido tutores, y reuniones de varias horas, sino que también hemos participado en tres proyectos relacionados con el campo de la exploración oceanográfica; entre todos hemos colaborado en coordinar aquellos aspectos comunes. Agradezco sus contribuciones y los ratos que hemos pasado juntos en numerosas tardes y fines de semana de trabajo en equipo.

A mi hermana Yolanda le quiero agradecer su ayuda en los momentos de crisis, por su inestimable colaboración en la redacción y corrección de la documentación del trabajo realizado. A mis padres también les agradezco la paciencia que han tenido durante todo este tiempo y que hayan confiado en mis posibilidades.

A todos aquellos que directa o indirectamente han permitido que haya podido llegar hasta aquí también les quiero dar gracias, en especial a los profesores y compañeros con los que he compartido los años de estudios universitarios.

Muchas gracias a todos.

Capítulo 1

Introducción

A lo largo de la historia el hombre ha mostrado interés por las tareas de observación y exploración oceánica, que parten desde los primeros conocimientos sobre las olas, corrientes e incluso el fenómeno de las mareas. Las primeras exploraciones modernas se centraban en la generación de documentación cartográfica limitada a la superficie marina. No sería hasta finales del siglo XVIII cuando se realizarían los primeros estudios sobre corrientes marinas documentados por James Rennell, que más adelante permitirían la elaboración de mapas como el que muestra la Figura 1.1. Pero hubo que esperar aún más para descubrir en 1849 qué se escondía más allá de las plataformas continentales, lo que se materializó en la publicación del primer libro de oceanografía en 1855, titulado *Physical Geography of the Sea*, de Matthew Fontaine Maury.

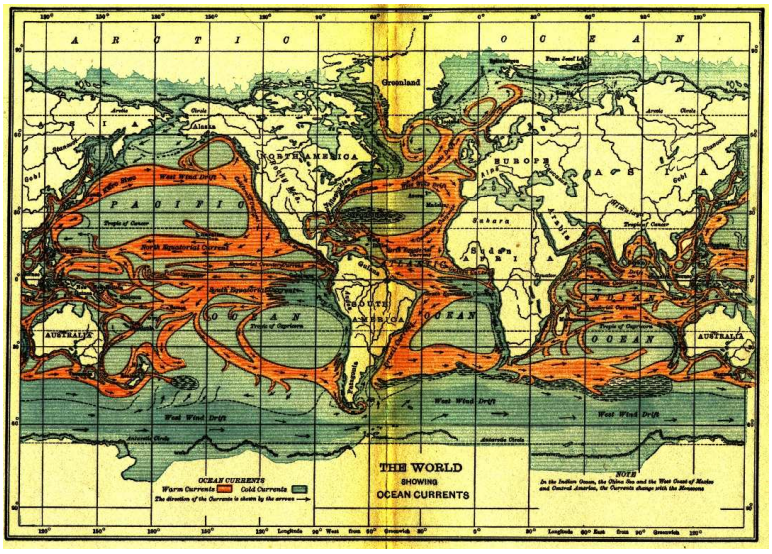


Figura 1.1: Mapa de Corrientes Marinas elaborado en 1911

La oceanografía, también conocida como Ciencias del Mar, es una ciencia amplísima con multitud de categorías centradas en el estudio de los océanos. La exploración de los

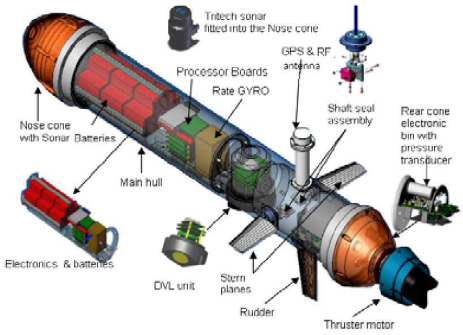
océanos y lagos es especialmente compleja y sólo en las últimas décadas hemos conseguido sumergirnos en las profundidades de éstos, gracias a los avances tecnológicos impulsados por un gran interés científico. Mucho ha tenido que ver en esto el legado de personajes como Jacques-Yves Cousteau, quien gustaba de autodefinirse como *técnico oceanográfico*. Sus contribuciones en este campo han permitido la exploración del *continente azul* tal y como lo conocemos hoy en día. Cuando en 1943, junto con Émile Gagnan, introdujo el primer prototipo de sistema de buceo autónomo conocido como *aqua-lung*, popularizó la exploración submarina, al otorgar al buzo independencia de la superficie, que ya no necesitaba un tubo para el suministro de aire, como ilustra la Figura 1.2 (a). A él se unen otros muchos investigadores e ingenieros que han participado en tareas de inspección oceánica desde diferentes enfoques, v. g. diseño de torpedos, inmersiones con batiscafos, desarrollo de submarinos y sumergibles civiles o militares , etc.

Desde entonces se ha avanzado mucho tecnológicamente y se ha suscitado una gran preocupación por la biología marina. Pero no es éste el único interés destacable, hay muchos otros: estudios climáticos, geología marina, topografía marina, investigación física y química, desarrollo y aprovechamiento de energías renovables, control de contaminantes, aplicaciones con fines militares, etc. No es de extrañar, por tanto, que el estudio de la oceanografía se divida en múltiples campos. Precisamente con el aprovechamiento de la tecnología para la exploración oceanográfica surge la Ingeniería Marina, como una rama de la ingeniería que trata el análisis, diseño y la planificación operativa de los sistemas que desarrollan su labor en el entorno marino.

La Ingeniería Marina abarca un amplio rango de sistemas, que cubre desde plataformas petroleras hasta los sumergibles. El diseño de vehículos de exploración oceánica no es en absoluto trivial e incluye una tipología de sistemas muy heterogénea, a la que esta ingeniería intenta ofrecer soluciones eficientes que faciliten las tareas de las misiones oceanográficas. Por ejemplo, para mejorar los modelos actuales del clima, se ha constatado la necesidad de aumentar la densidad de muestreo de ciertos parámetros físicos del mar, v. g. temperatura, salinidad, etc. La forma tradicional de recogida de datos, mediante campañas con buques oceanográficos, no resulta adecuada para cubrir estas necesidades debido a su altísimo coste. Nos encontramos ante una disciplina con amplio margen de innovación, ya que debe ofrecer soluciones tecnológicas a diversos y complejos problemas: autonomía, localización y navegación mediante cálculos de hidrodinámica, dificultades para la comunicación submarina, estructura física y durabilidad de los materiales bajo las adversas condiciones del medio acuático, etc.

Los primeros vehículos usados eran tripulados —como los batiscafos conectados mediante un cable umbilical a buques en superficie— pero pronto fueron surgiendo alternativas en forma de vehículos no tripulados y autónomos. El caso más simple y representativo lo constituyen las boyas oceanográficas, que son uno de los primeros sistemas de exploración oceanográfica autónoma, gracias a que en 1935 los investigadores del *Coastal and Geodetic Survey* inventaron una boya con telemetría automática por radio, eliminando la necesidad del apoyo por parte de buques tripulados. Las boyas permiten medir la meteorología en la superficie, lo que facilita la recopilación de información de la velocidad y dirección del viento, la presión barométrica, la temperatura del aire y el agua, la salinidad del agua, entre otros muchos parámetros físico-químicos. Hoy en día siguen usándose para la monitorización de fenómenos meteorológicos en zonas puntuales, como la flota de boyas oceanográficas instaladas en el Pacífico por el *Pacific Marine Environmental Laboratory* de la *National Oceanic and Atmospheric Administration* (NOAA) para observar

(a) Buceo. El joven Cousteau nadando con peces murciélago (Foto del *Ocean Futures Society* cortesía de KQED)



(b) Vehículo de exploración. Visualización en perspectiva isométrica del diseño tridimensional del AUV Maya [Madhan et al., 2005]

Figura 1.2: Exploración Oceanográfica

el fenómeno de *El Niño* y *La Niña*. Aún más destacables son los modelos de boyas a la deriva altamente avanzados, como las boyas perfiladoras de la red de muestreo global ARGO [Díaz, 2007].

Aunque no todas las boyas oceanográficas son iguales, en general comparten un problema de cobertura de exploración, que reduce su ámbito de actuación a zonas puntuales y en superficie. Por ello se ha evolucionado hacia sistemas submarinos móviles no tripulados, que se conocen como *Unmanned Underwater Vehicle* (UUV) —i. e. Vehículo Submarino No tripulado. También se han desarrollado vehículos móviles para la exploración en superficie, pero para esta labor los satélites oceanográficos, como los NOAA, ofrecen una solución adecuada, mediante imágenes de satélite que permiten determinar diversas magnitudes físico-químicas. Este tipo de vehículos requiere un sistema de navegación autónomo bastante complejo, lo cual motivó la apuesta por el control remoto desde buques oceanográficos en superficie. Se trata de los denominados *Remote Operated Vehicle* (ROV), que en todo momento se mantienen conectados al buque mediante un cable umbilical, a través del cual reciben los comandos de control remoto del piloto y envían información sensorial del entorno.

Gracias a la evolución de la Inteligencia Artificial (IA) y a los avances en las arquitecturas de control de sistemas robóticos móviles, se empezaron a plantear diseños que pudieran operar autónomamente. Esto permitió *cortar* el umbilical que unía los ROV a los mandos del piloto que se encontraba en superficie, y aparecerían en escena los *Autonomous Underwater Vehicles* (AUV). Los primeros prototipos eran parte de proyectos de investigación como el *Sea Grant's AUV Lab* del *Massachusetts Institute of Technology* (MIT), a lo largo de la década de 1970 —si bien el primer AUV propiamente dicho data de finales de la década de 1950, fruto del desarrollo de Stan Murphy, Bob Francis, y Terry Ewart, y conocido como *Self Propelled Underwater Research Vehicle* (SPURV) [Rudnick and Perry, 2003]. Desde entonces se han desarrollado gran variedad de AUVs, pudiéndose establecer toda una tipología en función de la misión para la que están diseñados, que se aborda en el Capítulo 2.

El proyecto aquí documentado se centra en el estudio y desarrollo de un sistema para AUVs, que incluye el estudio del equipamiento y la misión. Se comienza con el análisis del equipamiento y la tipología de vehículos de exploración submarina, con especial

atención en los AUVs. A continuación se define la tipología de misiones y se estudian las arquitecturas de especificación de misiones, para concluir con una propuesta basada en planes. Finalmente se analizan las arquitecturas de sistemas robóticos para AUVs y se propone *Sick***AUV** como sistema. Se tendrá como resultado el formato de especificación del equipamiento y de la misión, así como un diseño completo del sistema, que se complementa con una implementación básica y preliminar.

1.1. Objetivos

Aunque la materia de estudio de este proyecto es de caracter multidisciplinar, los objetivos que se persiguen se centran en el campo de la Ingeniería Informática, de modo que el objetivo principal puede resumirse con las siguientes líneas:

> Diseñar un sistema integrado de control para un AUV, que incluye mecanismos para la definición del equipamiento disponible a bordo del vehículo y de la misión a desarrollar.

Este proyecto tiene una finalidad fundamentalmente de estudio, análisis y diseño, puesto que parte desde cero, con un enfoque de investigación inicial de cara a futuros proyectos. No obstante, también se planteará una implementación básica como prueba de viabilidad del diseño propuesto, que servirá de prototipo o versión inicial en caso de darse continuidad al trabajo desarrollado mediante otros proyectos, o para facilitar la integración con proyectos complementarios.

El objetivo general del proyecto puede subdividirse en varios puntos más específicos:

Equipamiento Estudio del equipamiento físico y lógico que va embarcado en los vehículos de exploración submarina habitualmente, i. e. dispositivos e información, respectivamente. El estudio incluye un análisis de la tipología de este tipo de vehículos y se centra en los AUVs, para concluir con el desarrollo de una especificación del equipamiento que permita al vehículo disponer de una representación o modelo de sí mismo.

Misión Estudio y análisis de la tipología de misiones que acostumbran a realizar los vehículos de exploración submarina, con un sesgo especial hacia el uso de AUVs que operan en mar abierto. Se propone un formato de especificación de misiones adoptando una arquitectura y lenguaje apropiados, según el estudio de diversos casos documentados en la bibliografía.

Sistema Estudio de arquitecturas de sistemas de AUVs propuestas por multitud de institutos, universidades, empresas, etc., con el fin de realizar un análisis y valoración que permita una categorización de las mismas; como base se seguirá, en la medida de lo posible, la taxonomía de arquitecturas de control de sistemas robóticos móviles actualmente más aceptada por la comunidad científica. A partir del análisis se propone un sistema de desarrollo propio, para AUVs —aunque también será válido para otros vehículos de exploración oceanográfica similares. El sistema será capaz de ejecutar cualquier misión que se defina con el formato de especificación desa-

rrollado y, además, podrá usar la especificación del equipamiento para disponer de *autoconsciencia*[1] de los dispositivos, información y configuración del vehículo.

1.2. Estructura del Documento

El contenido del proyecto se divide en tres partes fundamentales, que tratan el equipamiento, la misión y el sistema de los vehículos de exploración submarina, en la Parte I, Parte II y Parte III, respectivamente. Para cada uno de estos tres aspectos se desarrollan las fases de estudio o análisis, diseño e implementación, con un alcance acorde a los objetivos del proyecto. En la Figura 1.3 se muestra un mapa conceptual donde se resumen la relación de cada uno de estos elementos con el vehículo, representado por un AUV.

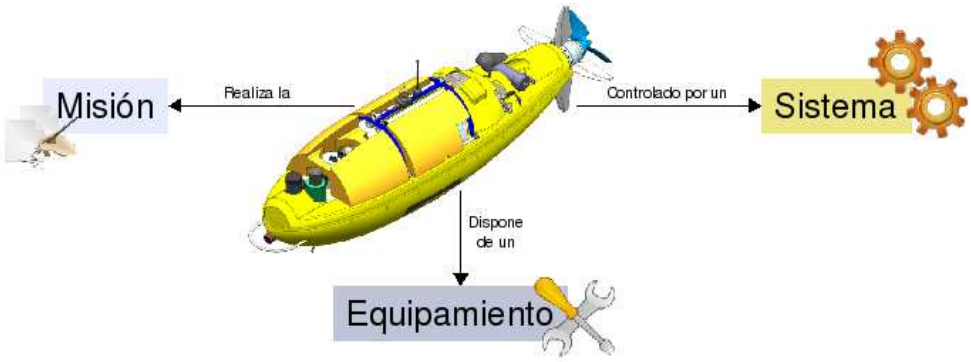


Figura 1.3: Mapa conceptual de la relación del AUV con el Equipamiento, la Misión y el Sistema

A continuación se desglosa el contenido de cada capítulo de este documento, así como las tres partes fundamentales:

Capítulo 2 Descripción del estado actual de la ingeniería oceanográfica, centrada en el desarrollo de software para AUVs.

Capítulo 3 Herramientas y proceso de desarrollo software empleado para la elaboración del proyecto.

Capítulo 4 Detalle de los recursos hardware, software o de otro tipo, necesarios para la consecución del proyecto. Se describe el equipamiento disponible para la preparación de un entorno de pruebas con las características de un AUV.

Capítulo 5 Desarrollo del plan de trabajo desglosado en etapas, con una estimación del tiempo de ejecución.

Parte I Estudio de los vehículos de exploración submarina y el equipamiento de los mismos, con un enfoque especialmente centrado en los AUVs.

Capítulo 6 Definición del equipamiento del cual disponen los vehículos de exploración submarina.

[1] El equipamiento permite al sistema construir un modelo de los dispositivos, la información y la configuración disponible.

Capítulo 7 Estudio de la tipología de vehículos de exploración submarina, que pueden beneficiarse de las misiones y el sistema propuesto para AUVs.

Capítulo 8 Análisis del equipamiento físico habitualmente embarcado y que determina las capacidades y misiones realizables.

Capítulo 9 Detalle del formato de especificación del equipamiento, características y configuración del vehículo.

Parte II Estudio de las misiones que realizan los vehículos de exploración oceanográfica, con la finalidad de proponer un formato de especificación de misiones usando la arquitectura más apropiada posible.

Capítulo 10 Definición de la misión en el ámbito de la exploración submarina.

Capítulo 11 Análisis de la tipología de misiones que se acostumbra a realizar con este tipo de vehículos, especialmente con AUVs.

Capítulo 12 Estudio de las diferentes arquitecturas de especificación de misiones propuestas en la bibliografía. Se identifican tres tipos de especificación fundamentales, basadas en **redes de Petri**, en **tareas** y en **comportamientos**, y se realiza una propuesta basada en la arquitectura de **tareas**, que organiza la misión en **planes**.

Capítulo 13 Detalle del formato de especificación de la misión mediante la arquitectura propuesta, i. e. los **planes de la misión**.

Parte III Estudio de los sistemas embebidos en los vehículos de exploración submarina, con un enfoque especial sobre la arquitectura y la aplicabilidad en AUVs.

Capítulo 14 Estudio de arquitecturas de sistemas desarrollados para AUVs, atendiendo a la estructura y a los módulos de la misma.

Capítulo 15 Sistema propuesto, bajo el nombre de *Sick***AUV**, para su implantación en vehículos de exploración submarina, especialmente AUVs, con soporte para manejar la especificación del equipamiento y la interpretación de la misión.

Parte IV Resultados experimentales y demostraciones del trabajo desarrollado dentro del ámbito del proyecto, mostrando algún ejemplo misión y ejecución del sistema con el nivel de implementación alcanzado.

Capítulo 16 Muestra un ejemplo representativo de misión de exploración oceanográfico y cómo se especifica con la sintaxis de los planes de la misión.

Capítulo 17 Estudio general de los resultados obtenidos para cada uno de los objetivos marcados y conclusiones derivadas de ello.

Capítulo 18 Contribuciones del proyecto para la realización de nuevos proyectos y el trabajo futuro que puede desarrollarse como continuación o complemento de éste.

Capítulo 2

Estado del Arte

El Océano.
Ese vasto cuerpo de agua que ocupa unos dos tercios de un mundo hecho para el Hombre
—que resulta que no tiene agallas.

— Ambrose Gwinett "Bitter" Bierce
1842–1914 (Escritor, periodista y editorialista estadounidense)

Las investigaciones en el campo de los AUVs son sólo una parte de los esfuerzos de investigación en el área de vehículos no tripulados en aire, tierra y mar. Los vehículos no tripulados, o *Unmanned Vehicles* (UVs), ya sean operados remotamente o autónomos, eliminan la necesidad de la presencia física del hombre en el propio vehículo. Los UVs operados remotamente usan la telerobótica para la navegación y control, mientras que en los autónomos no interviene ningún operario humano, de modo que disponen de inteligencia y un sistema de control embebido.

El diseño del sistema y la arquitectura de control subyacente es el mayor problema que nos encontramos en el desarrollo de vehículos autónomos no tripulados, debido al manejo de datos sensoriales multidimensionales, el procesamiento computacionalmente intensivo y la ejecución en tiempo real. Este problema es aún más complejo en el caso de los AUVs, como consecuencia de las limitaciones de potencia y comunicación [Valavanis et al., 1997].

A continuación se realiza un breve repaso de los proyectos de investigación y desarrollos de UVs más representativos. Se remite al lector al Capítulo 7 para la consulta de la clasificación realizada de los vehículos de exploración submarina y el estudio detallado de los mismos, especialmente centrado en los UVs. No obstante, el estado del arte mostrado se enfoca fundamentalmente en los AUVs, dado que el desarrollo de este proyecto tiene este modelo de vehículo como objetivo principal.

2.1. Vehículos no tripulados. Visión general

El término vehículos no tripulados hace referencia a cualquier tipo de vehículo sin un tripulante humano, pudiendo realizarse una clasificación en función del medio en el que operan: **aéro**, **terrestre** o **submarino**. Por norma general, éstos pueden ser **operados remotamente** o **autónomos**. Además, en el caso de ser operados remotamente, nos podemos encontrar con vehículos que disponen de un cable umbilical que los une a la estación de trabajo desde donde un operario humano envía los comandos de control, o vehículos sin umbilical, i. e. la comunicación es inalámbrica.

En el caso de los vehículos submarinos no tripulados, o *Unmanned Underwater Vehicles* (UUVs), las dificultades de comunicación en el medio subacuático suelen motivar el uso de umbilical; aunque si puede asumirse el coste económico y el consumo de un módem acústico puede eliminarse el umbilical. La eliminación del umbilical no sólo libera al vehículo de la conexión con el buque oceanográfico en superficie, sino que soluciona los posibles arrastres, minimiza los problemas de enredos y desaparecen las restricciones de profundidad y maniobrabilidad.

Un vehículo submarino sin umbilical no tripulado, o *Unmanned Untethered Underwater Vechicle* (UUUV), puede disponer de diferentes capacidades de maniobrabilidad, lo que permite una división en dos grandes grupos: **vehículos de crucero** y **vehículos de *hovering*** (capaces de mantenerse en una posición fija). La maniobrabilidad del vehículo determina la tipología de misiones que éste puede realizar y, por tanto, las aplicaciones prácticas del mismo. Así, los vehículos de crucero se usan en exploración, búsqueda, localización de objetos, etc., en mar abierto, donde hay grandes espacios y sólo se requiere de 3 grados de libertad para la navegación (avance longitudinal, control de rumbo y cabeceo). Mientras que los vehículos de *hovering* se destinan a tareas de inspección detallada y trabajos físicos sobre objetos fijos, para lo que se requiere de al menos 5 grados de libertad (avance en las 3 direcciones ortogonales xyz del espacio, control de rumbo y, en ocaciones, cabeceo).

Este proyecto se centra en los AUVs de crucero, que ofrecen una solución tecnológica muy efectiva en la exploración oceanográfica demandada en la época actual. A pesar de tratarse de un caso muy concreto, veremos como podemos encontrarnos numerosos casos innovadores, que intentan solventar los complejos problemas de diseño de este tipo de vehículos.

2.2. Vehículos Submarinos Autónomos

Los AUVs de crucero constituyen el diseño más común y disponen de una estructura hidrodinámica característica en forma de *torpedo*, como muestra la Figura 2.1. Son pequeños submarinos autónomos dotados de sistemas de propulsión generalmente eléctrica, donde gran parte del volumen del submarino está ocupado por baterías que proporcionan la energía a los motores y el resto se destina a los sistemas de medición y navegación.

Hoy en día podemos encontrar AUVs con diseños muy representativos, así como proyectos de investigación, *workshops*, concursos, etc., que cada año proponen nuevos avances. Pero aún hay bastante margen de mejora en el afán por conseguir sistemas que satisfagan por completo las demandas de los oceanógrafos de todo el mundo. Es mucha la información demandada y muy complejo el entorno de actuación, donde las diversas restriciones van desde los condicionantes físicos hasta el propio software del sistema em-

Figura 2.1: AUV de crucero. AUV Gavia

bebido —v. g. estructura e hidrodinámica del vehículo, equipamiento de comunicación, consumo energético, disipación de calor, etc. y localización, navegación, almacenamiento de muestras, etc., respectivamente— (véase la Sección 14.1).

A continuación se comentan algunos de los diseños de AUVs más representativos a nivel mundial, con especial atención a las contribuciones de instituciones europeas y del territorio español. En cada caso se intentan mostrar las características más novedosas en lo referente al equipamiento del vehículo, las misiones que son capaces de realizarse y la arquitectura del sistema embebido que se emplea para conseguirlo.

AUV SAUVIM El *Semi-Autonomous Underwater Vehicle for Intervention Missions* es un AUV de la Universidad de Hawaii, para el que el sistema implantado. En su versión más representativa se emplea una arquitectura de control híbrida organizada en tres capas: planificación (deliberativa), control y ejecución (reactiva) [Ridao et al., 2005].

La misión se planifica como una secuencia de sub-objetivos o tareas, que se ejecutará de forma supervisada. De este modo, las tareas consituyen el módulo básico de la arquitectura del sistema.

AUV MARIUS El sistema de control más destacable implantado en el AUV MARIUS, desarrollado en Sines (Portugal), sigue una metodología de diseño basada en primitivas parametrizables. Dichas primitivas se obtienen mediante la coordinación de tareas concurrentes, ejecutadas en función de los programas de la misión [Ramalho Oliveira et al., 1996, Ramalho Oliveira et al., 1998]. El análisis y diseño de estas primitivas se sustenta en la teoría de las redes de Petri, que están orientadas al modelado y análisis de sistemas de eventos asíncronos y discretos con concurrencia, como es el caso de los AUVs.

El AUV MARIUS usa los entornos de programación CORAL y ATOL para el desarrollo e implementación del sistema del vehículo de forma gráfica mediante librerías de primitivas y redes de Petri, que también permiten la especificación de la misión.

DeepC Compañía alemana que elabora AUVs de gran tamaño con un diseño formado por dos módulos con forma de *torpedo* que llevan la carga útil entre ellos. El sis-

tema está equipado con cámaras, sónar frontal y lateral, entre otros dispositivos o instrumentos de navegación, y con sensores de temperatura, conductividad y presión, para la misión. Este equipamiento es uno de los más básicos que podemos encontrarnos (véase la Parte I para una visión más detallada del equipamiento que suele embarcarse en este tipo de vehículos).

Bluefin Esta compañía es un *spin-off* del *MIT Autonomous Underwater Vehicle Laboratory*, que fabrica AUVs de pequeño y gran tamaño, todos ellos con capacidades de inmersión a grandes profundidades. La principal novedad del diseño de estos vehículos radica en la modularidad de sus componentes. Los submarinos pueden desensamblarse en control, carga útil y proa hidrodinámica, de los cuales se dispone en forma de secciones cilíndricas intercambiables, que en su conjunto dotan al vehículo de una morfología de *torpedo*.

AUV GARBI GARBI es un UV originalmente desarrollado como ROV por la Universidad de Girona, pero que se transformó en un AUV tras la implementación de un sistema de control basado en una arquitectura híbrida con aprendizaje por refuerzo y centrada en la coordinación híbrida de los diferentes comportamientos definidos [Carreras Pérez, 2003, Carreras Pérez et al., 2003]. El desarrollo de GARBI se basa en otros modelos previos del mismo, y tiene su continuación en otros como ICTINEU, así como modelos similares, v. g. GARBI II, URIS, RAO II. En GARBI se han probado y estudiado diversas arquitecturas de control, destacando los esquemas motores, que se basan en campos potenciales para modelar el sistema de navegación [Carreras Pérez et al., 2003]. Se puede consultar información del proyecto en la web del programa *airsub* (Aplicaciones Industriales de los Robots SUBmarinos):

`http://eia.udg.es/~pere/airsub/`

Aparte de los diseños de AUVs, también son dignas de mención las siguientes fuentes de recursos e información en este campo de investigación:

AUVSI La *Association for Unmanned Vehicle Systems International* es la mayor organización sin ánimo de lucro a nivel mundial dedicada en exclusiva al avance de la comunidad de sistemas no tripulados, promoviendo el desarrollo de este tipo de sistemas y tecnologías relacionadas, con el apoyo de organizaciones gubernamentales, industriales y académicas.

`http://auvsi.org`

UUVS La *Unmanned Underwater Vehicle Showcase* es una organización que organiza exhibiciones, conferencias y *workshops* de UUVs. Reune a los principales suministradores y productores de AUVs y ROVs a nivel internacional en un marco excepcional para la exhibición y comparación de las más modernas tecnologías y desarrollos en el campo.

`http://www.uuvs.net`

Sub-Log.com Es una página web sobre la exploración submarina, con un enfoque primordial en los submarinos, sumergibles, vehículos y tecnologías concomitantes. *Sub-Log* es una organización privada no afiliada a ninguna organización militar

ni gubernamental, con sede en Seattle (Washington State, USA). A través de la web ofrece múltiples noticias y bibliografía sobre el campo de los vehículos de exploración oceanográfica submarina.

http://sub-log.com

SUT La *Society for Underwater Technology* es una sociedad multidisciplinar donde confluyen organizaciones y particulares con intereses comunes en la tecnología submarina, las ciencias oceanográficas y la ingeniería marina. SUT se fundó en 1966 y tiene miembros de más de 40 países, que incluyen ingenieros, científicos y otros profesionales y estudiantes que trabajan en estas áreas. A través de su web ofrece información en forma de cursos, publicaciones, noticias, eventos, material educativo, etc.

http://www.sut.org.uk

WMO La *World Meteorological Organization* es una agencia especializada de las Naciones Unidades reconocida como portavoz autorizado en el estado y comportamiento de la atmósfera terrestre y su interacción con los océanos, el clima y la distribución resultante de las aguas.

http://www.wmo.int

De este modo, aunque el proyecto aquí documentado no parte de ningún desarrollo previo, aprovecha todo el *know how* apuntado en las fuentes bibliográficas y los recursos disponibles como punto de partida. Los anteriores ejemplos, que ilustran el estado del arte de la témática del proyecto, son sólo una pequeña muestra de los esfuerzos de investigación que se impulsan internacionalmente. Las fuentes consultadas para la elaboración de este proyecto no se reducen a estos ejemplos, como se observará a lo largo del documento, pero evidentemente sólo abarcarán un subconjunto del total. Sin embargo, se consideran suficientemente representativas de la tipología de AUVs y misiones de exploración oceanográfica, como para considerarse válidas para el estudio, análisis y diseño de las propuestas de equipamiento, misión y sistema para un AUV.

2.3. Exploración Coordinada

Los diferentes tipos de vehículos y sistemas de exploración oceanográfica pueden emplearse de forma combinada para trabajar en redes, donde cada nodo desempeñe un rol determinado y se coordine con el resto para conseguir un trabajo más efectivo [Borges de Sousa and Lobo Pereira, 2002]. Este marco de exploración requiere de sistemas con habilidades de coordinación y de dispositivos de comunicación adecuados al medio de actuación de cada elemento de la red. La tecnología disponible hoy en día permite dar solución a estos problemas y podemos encontrarnos con misiones de exploración coordinadas entre múltiples vehículos y sistemas, como AUVs, buques oceanográficos, satélites oceanográficos o de comunicación, boyas oceanográficas, etc. (véase la Figura 2.2).

Desde el punto de vista de un vehículo submarino como un AUV, este tipo misiones coordinadas aporta numerosos beneficios, como la posibilidad de comunicación con el exterior, posicionamiento global, etc., lo que se consigue fácilmente mediante la transferencia de información obtenida por un buque en superfice al AUV, usando algún tipo de dispositivo de comunicación subacuático.

Figura 2.2: Coordinación de múltiples vehículos o sistemas de exploración oceanográfica. *Autonomous Ocean Sampling Network* (AOSN)

El ejemplo más representativo de este tipo de misiones de exploración es el programa *Autonomous Ocean Sampling Network* (AOSN) [Hanrahan and Bellingham, 2008], que tal y como ilustra la Figura 2.2 reúne modernos y sofisticados vehículos robóticos con avanzados modelos oceánicos para mejorar las capacidades de observación y predicción oceanográfica. El sistema recopila datos mediante plataformas y sensores inteligentes y adaptables que transfieren la información a la costa en tiempo real, donde se integran en modelos numéricos de predicción.

El desarrollo de estrategias de control para comandar los vehículos de exploración hacia lugares donde los datos resultan más relevantes es un aspecto importante, al cual denominan *adaptive sampling* o muestreo adaptativo en el programa AOSN. La capacidad de predecir propiedades físicas del océano, como la temperatura y corrientes, así como sus contrapartidas biológicas (productividad del ecosistema) y químicas (fertilización de nutrientes) proporcionan una prueba fundamental para comprender los procesos oceánicos. En éste, como en mucho programas, el éxito depende de la colaboración de toda una red de instituciones diseminadas por todo el país, como Estados Unidos en el caso de AOSN.

En el diseño de la misión y el sistema propuestos en este proyecto no se dará soporte a la coordinación, pero será un aspecto recogido en el análisis. El esfuerzo necesario para realizar este tipo de misiones sólo está al alcance de instituciones con una gran financiación y disponibilidad de recursos. En cualquier caso, los diseños propuestos están preparados para que la incorporación de mecanismos de coordinación resulte sencilla.

Capítulo 3

Metodología

Los métodos son vías que facilitan el descubrimiento de conocimientos seguros y confiables para solucionar los problemas que la vida nos plantea.

— Miguel Martínez Miguélez
1999 (Dpto. Ciencia y Tecnología del Comportamiento
Universidad Simón Bolívar)

El objetivo de este Proyecto Fin de Carrera consiste en el desarrollo de un software, por lo que resulta apropiado la aplicación de los métodos, herramientas y procedimientos de la **Ingeniería del Software**. Esta disciplina adopta un enfoque sistemático para la producción que cubre las diferentes fases del desarrollo del software. Estas fases constituyen lo que se conoce como el **ciclo de vida del software** [Pressman, 1993].

El ciclo de vida convencional ha sido el paradigma de desarrollo por excelencia, constituyendo la base para el resto de paradigmas que han surgido posteriormente. Por este motivo, las fases por las que habitualmente pasa el proceso de desarrollo de un software son, secuencialmente:

Análisis Es el proceso de estudio, descubrimiento, refinamiento, modelado y especificación, cuyo principal objetivo es obtener modelos lógicos que definan el software que se desea construir.

Diseño El diseño es el proceso en el que se asienta la calidad del desarrollo de software. Define la estructura del software, i. e. su arquitectura, a partir de los requisitos del producto, obtenidos durante la fase de análisis.

Implementación La implementación o codificación, consiste en la traducción del diseño a un lenguaje de programación específico y apropiado, de acuerdo con las características del software y la tecnología disponible.

Prueba La prueba es un proceso de ejecución de un programa con la intención de descubrir errores. Es un elemento crítico para asegurar la garantía de calidad del software y demostrar que se han alcanzado los requisitos apuntados en el análisis.

Mantenimiento El mantenimiento constituye la última fase del ciclo de vida del software, una vez finalizado el desarrollo. La fase de operación y mantenimiento debe asegurar que el producto satisface los requisitos del análisis y que el software está diseñado de forma apropiada para facilitar las correcciones de errores, modificaciones y extensiones. Éste es un requisito implícito del software, que se aborda mediante la aplicación de una metodología de desarrollo software apropiada y de calidad.

A continuación se describe el paradigma de desarrollo empleado para la consecución del software que debe realizarse como parte de los objetivos del proyecto, el cual consta de tres elementos:

Equipamiento Especificación del equipamiento de vehículos de exploración oceanográfica, en especial un AUV.

Misión Especificación de las misiones que realiza este tipo de vehículos, utilizando una arquitectura apropiada.

Sistema Diseño del sistema integrado de control para un AUV.

3.1. Paradigma de Desarrollo

Los paradigmas de desarrollo de la **Ingeniería del Software** constituyen diferentes enfoques del ciclo de vida del software. La elección del paradigma atiende a la naturaleza del proyecto, los métodos y herramientas a utilizar y los plazos de entrega requeridos. Por este motivo, es importante usar un paradigma apropiado, aunque en el campo de los sistemas robóticos móviles existe todo un conjunto de entornos orientados al diseño e implementación de este tipo de sistemas, lo cual hace que el paradigma se adapte o emplee de una forma más libre.

Para la elaboración de la especificación del equipamiento y de la misión se realizará un análisis y estudio de los mismos en el ámbito de los vehículos de exploración oceanográfica. A continuación se diseñará un Lenguaje de Dominio Específico (DSL) que permita realizar dichas especificaciones. Los paradigmas clásicos no están pensados para el diseño de lenguajes, por lo que se usa una guía de diseño de DSLs que muestra diversos patrones de diseño aplicables según las características del dominio y el lenguaje que se desea elaborar [Spinellis, 2001, Usuarios Wikipedia, 2008a].

En el Apéndice B se comentan con más detalle los DSLs y los diferentes patrones de diseño documentados. Además, en la Definición B.3 se explica el proceso de Programación Orientada al Lenguaje (LOP) que consiste en la creación de DSLs como una parte del proceso de solución de problemas. Esto resulta especialmente útil cuando el lenguaje permite expresar un tipo particular de problemas o soluciones de forma más clara que con lenguajes ya existentes, y el problema en cuestión se repite con la suficiente frecuencia; éste el caso de la especificación del equipamiento y de la misión.

Se ha optado por el uso de XML como lenguaje base, aplicando así el patrón de diseño de DSLs conocido como **piggyback**, el cual consiste en aprovechar las capacidades de un

lenguaje existente como base para el nuevo DSL (véase la Sección B.3 para más detalles). Se usan esquemas XML (XSD) para definir la sintaxis del lenguaje, de modo que el diseño y la implementación se han abordado simultáneamente.

En primer lugar se desarrolla la especificación del equipamiento, para luego aprovechar el análisis realizado en el desarrollo de la especifición de la misión, en el que se estudia la tipología de misiones y las arquitecturas de especificación. Entre estas arquitecturas destacan las redes de Petri, que disponen de su propio formalismo y metodología de desarrollo. Sin embargo, se ha optado por una arquitectura basada en tareas.

Finalmente, se inicia el desarrollo del sistema que irá embarcado en el vehículo. En este caso se aplica el Modelo o Aproximación Incremental (AI) como paradigma de desarrollo (véase la Figura 3.1). En la AI se desarrollan múltiples versiones, donde en cada iteración del proceso de desarrollo se tiene mayor funcionalidad y capacidad. Este paradigma permite utilizar la versión anterior mientras se termina de desarrollar la siguiente, en paralelo. Las principales ventajas de la AI son:

1. Posibilita la implementación de ciertas funcionalidades antes que otras.
2. Es adecuada para equipos de desarrollo pequeños, lo cual hace que sea apropiada para proyectos donde sólo trabaja un desarrollador.
3. Es apropiada para proyectos de larga duración, ya que permite la consecución del mismo por partes.
4. Permite la realización de pruebas de los diferentes módulos del sistema de forma incremental, lo cual permite garantizar la fiabilidad de las partes del sistema, haciendo que éste sea más robusto. Esto implica un coste de tiempo importante, que suele considerarse una desventaja. Pero en el caso de los sistemas robóticos móviles es un aspecto importante en el desarrollo, que lo convierte en una ventaja.

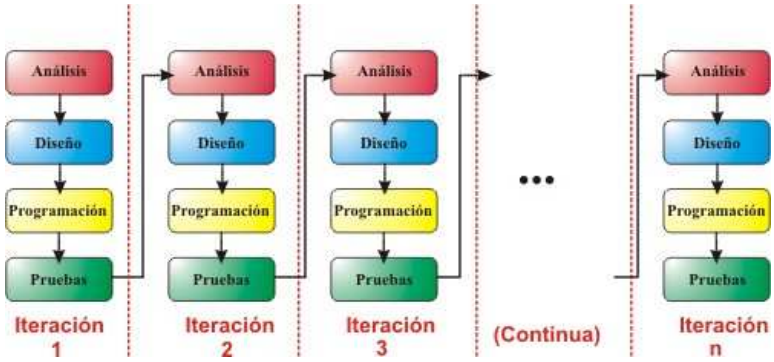


Figura 3.1: Paradigma de Desarrollo. Modelo Incremental

El paradigma de desarrollo del sistema ha sido adaptado para el uso de arquitecturas de control de sistemas robóticos móviles, en especial para AUVs. En concreto se ha aplicado una arquitectura de control híbrida (deliberativa y reactiva), modelada mediante componentes CoolBOT, que es un entorno orientado al desarrollo de este tipo de sistemas (véase la Sección E.1).

El uso de la AI obliga a la realización de pruebas de integración y regresión. Esta técnica consiste en probar el software al integrar los diferentes módulos del mismo, lo

cual es necesario debido a las características de la AI. Para poder realizar esta prueba sobre un módulo concreto se requiere la implementación de dos elementos adicionales:

Controlador Programa principal que acepta los datos del caso de prueba, pasa los datos al módulo e imprime los resultados.

Resguardo Programa que reemplaza los módulos subordinados, realiza alguna manipulación de datos, imprime una verificación de entrada y devuelve el control. A todos los efectos sería como una *caja negra* para el **controlador**.

La implementación de los intérpretes de la misión se realiza mediante una estrategia *top-down* o descendente, acorde con las puebas de integración. Se crean resguardos que reemplazan a los componentes que ejecutan las acciones de las tareas especificadas en la misión. De este modo pueden probarse los intérpretes de la misión sin haber desarrollado el resto del sistema.

Sin embargo, la construcción del resto del sistema sigue una estrategia *bottom-up* o ascendente, para ir desarrollando incrementalmente los diferentes componentes y subsistemas. A medida que se van terminando los módulos, se alcanzan nuevas versiones más funcionales y éstas se someten a pruebas de integración. En este caso se trata de pruebas de integración ascendente, de modo que no hay que implementar resguardos, sino controladores con las baterías de prueba del sistema.

3.1.1. Análisis y Estudio de Casos

El desarrollo comienza con el estudio de vehículos y su equipamiento, con vistas a la elaboración de la especificación del equipamiento. Al mismo tiempo se analiza la tipología de misiones que realizan los vehículos de exploración oceanográfica. Durante el análisis es habitual la realización de entrevistas para determinar el alcance y los requerimientos del software. Sin embargo, en lugar de aplicar técnicas de entrevista a oceanógrafos, se ha realizado un estudio profundo de la bibliografía más representativa. Esto requiere un esfuerzo mayor, pero ofrece una visión más global.

Finalmente, para el desarrollo del sistema se analizan arquitecturas de control de sistemas robóticos móviles para AUVs. Se establece una clasificación en base a la estructura y otra en función del tipo de módulos que se emplean. Esto permitirá determinar cuál es la arquitectura más apropiada para el sistema que se propondrá.

3.1.2. Diseño

A partir del análisis del equipamiento se elabora una estructuración y organización de la especificación del mismo. En base a la tipología de misiones y al estudio de arquitecturas de especificación de misiones expuestas en la bibliografía, se realiza una clasificación de las mismas y se propone una arquitectura propia, basada en tareas y compuesta por planes.

El sistema se diseña definiendo en primer lugar la arquitectura del mismo. Ésta será una arquitectura híbrida (deliberativa y reactiva) que se modela mediante componentes CoolBOT. Se elabora un diseño orientado a objetos para dar soporte a todas las funcionalidades del sistema. De acuerdo con la naturaleza de esta metodología de diseño, el sistema se organiza en subsistemas y los datos y operaciones se encapsulan en obje-

tos. Los subsistemas se modelan mediante componentes CoolBOT compuestos. A su vez, éstos contienen componentes CoolBOT internamente.

Se han aplicado diversos patrones de diseño a la hora de diseñar el sistema. Estos patrones permiten crear la arquitectura de diseño integrando componentes reusables [Gamma et al., 2003]. Se ha estudiado la aplicabilidad de diversos patrones para el modelado de determinadas partes del sistema. A continuación se enumeran algunos de los patrones de diseño más representativos que se han usado:

Factoría Abstracta Se ha usado una variante de este patrón para construir una factoría de clases, similar a la de la clase **Object** de Java.

Prototipo Se usa el patrón prototipo para facilitar la clonación de objetos prototípicos, sin necesidad de conocer la clase concreta de los mismos.

Singleton Se usa para aquellas clases de las que sólo puede existir una instancia.

Adapter También conocido como *wrapper* o envoltorio, se ha usado para crear clases que adaptan *drivers*, tipos primitivos, etc. a una interfaz requerida por el resto del sistema.

Fachada Se han creado determinadas clases que hacen de fachada de determinadas partes del sistema.

Comando Se usa para el diseño de las acciones de las tareas de la misión, permitiendo un tratamiento simple y homogéneo.

3.1.3. Implementación

La implementación del equipamiento y la misión se materializa con la especificación formal de ambos, en XML. Además, se define la sintaxis mediante esquemas XML (XSD), que permiten la validación automática utilizando librerías que realizan el análisis de ficheros XML.

El sistema se ha implementado en C++, usando diversas librerías o entornos de programación como apoyo, especialmente CoolBOT. Se trata, por tanto, de un *framework* para el desarrollo de sistemas robóticos móviles. Además, se han usado algunas técnicas de implementación —en ocaciones denominadas *idioms*— para solucionar problemas inherentes del lenguaje, como son:

pImpl Esta técnica, también conocida como **opaque pointer**, **d-pointer**, **handle classes**, o **"Chesire Cat"**, que permite la realización de una implementación privada (de ahí el nombre de **pImpl**) [Sutter, 2000]. Esto permite que se reduzcan las dependencias y el tiempo de compilación durante el desarrollo del sitema.

Contadores de Schwarz Esta técnica, desarrollada por Jerry Schwarz y también conocida como **nifty counters**, soluciona el problema del orden de declaración de los atributos **static** en las clases de C++ [Eckel, 2007, Cline et al., 1998]. Esto ocurre porque el compilador no garantiza ningún orden de compilación de los ficheros. Con esta técnica se fuerza la declaración justo en el orden de inclusión de los ficheros cabecera de las clases.

3.1.4. Prueba del Software

Se han realizado pruebas con ejemplos de la misión, realizando la carga de las mismas con los intérpretes del sistema, para comprobar que la validación XML mediante los esquemas XML (XSD) funciona correctamente (véase la Sección E.3 para más información sobre el proceso de validación XML). Para el sistema desarrollado se han ido realizando pruebas de integración y regresión. Éstas han sido descendentes para probar los intérpretes de la misión y ascendentes para probar incrementalmente los diferentes componentes y subsistemas desarrollados. También se han llevado a cabo pruebas de unidad para verificar la correcta adaptación de los dispositivos a *Player*, así como determinadas implementaciones puntuales, v. g. comandos, comunicaciones, serialización XDR, etc.

3.1.5. Mantenimiento, Gestión y Administración del Software

El diseño utilizado para la especificación del equipamiento y de la misión utiliza XML de forma adecuada, sustentado en esquemas XML (XSD). Este permite que el mantenimiento sea más sencillo, ya que la información de la sintaxis del lenguaje está autocontenida en la estrutura de ficheros de especificación. En el caso del sistema, el uso de un diseño basado en componentes CoolBOT para la arquitectura y un diseño orientado a objetos para los datos y procedimientos, permiten la reutilización y fácil mantenimiento del software desarrollado.

Durante el desarrollo del proyecto se ha hecho uso de herramientas como el control de versiones, lo cual es uno de los procedimientos del mantenimiento del software más importante. Esta tarea ha sido automatizada mediante el software **Subversion**, junto con otras herramientas gráficas que facilitan su gestión.

Capítulo 4

Recursos necesarios

Hay que tener aspiraciones elevadas,
expectativas moderadas y
necesidades pequeñas.

— William Howard Stein
1911–1980 (Químico, bioquímico y profesor universitario norteamericano
Premio Nobel de Química en 1972)

La realización de este proyecto requiere de diversos recursos, tanto *software* como *hardware*. A continuación se enumeran los principales recursos que han sido usados a lo largo de todo el proceso de desarrollo, distinguiendo entre el *software* y el *hardware* y utilizando la siguiente clasificación, de acuerdo al uso que se ha hecho de ellos.

Desarrollo Herramientas para el desarrollo del sistema que se propone en este proyecto.

Sistema Librerías software y equipamiento físico y lógico necesarios para el funcionamiento del sistema desarrollado (véase la Parte I para más detalles sobre la tipología del equipamiento que suele embarcarse en los vehículos de exploración oceanográfica).

Mantenimiento Herramientas para realizar las labores de mantenimiento durante el desarrollo del proyecto, como copias de seguridad, control de versiones, etc.

Documentación Material utilizado para documentar el trabajo realizado, lo que cubre el análisis, diseño e implementación.

Adicional Recursos adicionales que se han usado para determinadas pruebas o estudios durante el proyecto.

4.1. Software

El software aquí comentado no sólo hace alusión a las aplicaciones informáticas, en forma de algoritmos o ejecutables, sino también a la información que ha sido necesaria, i. e. fuentes de información, recursos o datos lógicos, etc.

De forma general, para las distintas tareas realizadas, se ha hecho uso de los siguientes sistemas operativos (en la Sección 4.2 se indica la lista de equipos informáticos y los sistemas operativos instalados):

Windows® Se ha usado **Windows®XP SP2** y **Windows®Vista SP1**, fundamentalmente para tareas de **documentación**.

GNU/Linux Se han usado distribuciones basada en **Debian**, porque gracias al sistema de actualizaciones del que disponen facilitan enormemente la instalación de las diversas aplicaciones y librerías que se han empleado en el proyecto. Concretamente se ha tomado la distribución **GNU/Linux Kubuntu**, que es una derivación de **Ubuntu** con el entorno de escritorio KDE preinstalado. Se ha usado KDE porque integra la mayor parte de herramientas de desarrollo usadas, tal y como se observa en la Sección 4.1.1, v. g. **KDevelop**. A lo largo del proyecto se han usado las siguientes versiones de esta distribución: 6.10 (*Edgy Eft*), 7.04 (*Feisty Fawn*), 7.10 (*Gutsy Gibbon*) y 8.04 LTS (*Hardy Heron*).
Respecto al *kernel* de Linux se han empleado las versiones 2.6.x, desde sus primeras versiones. Actualmente se está trabajando con la **2.6.24-19-generic**.

4.1.1. Desarrollo

El desarrollo del proyecto se ha realizado en el entorno **GNU/Linux**, fundamentalmente. Esto está motivado por la gran disponibilidad de recursos y librerías que se integrarán en el sistema desarrollado. La lista de aplicaciones y utilidades empleadas en el desarrollo del proyecto, especialmente en el diseño e implementación del sistema *Sick***AUV**, es la siguiente:

KDevelop Entorno de Desarrollo Integrado, o *Integrated Development Environment* (IDE) para C/C++, en el que se ha desarrollado *Sick***AUV**, los *drivers* de los dispositivos físicos del vehículo y algunas pruebas. Se ha usado desde la versión 3.4.x hasta la 3.5.3 actualmente.

make Creación y ejecución de ficheros **Makefile**, para la compilación de algunas pruebas o módulos del sistema desarrollado.

GCC Colección de Compiladores de GNU, o *GNU Compiler Collection*, que permite compilar programas escritos en C/C++. Se ha usado la extensión **g++** para C++.

autotools Herramientas para la compilación, *linkado* y construcción automática [Vaughan et al., 2000]. En el caso de **KDevelop**, esto ya está integrado en los proyectos generados por el propio IDE. No obstante, se ha usado en algunas pruebas (veáse la Sección D.1 para más detalles sobre **autotools**).

libtool Herramientas para trabajar y *linkar* librerías compartidas, o *shared libraries* [Drepper, 2006, Vaughan et al., 2000].

Generador de Componentes CoolBOT Se ha desarrollado un generador de esqueletos de componentes CoolBOT, para facilitar la tarea de declaración e implementación de los mismos (véase la Sección E.1.3 para más detalles sobre CoolBOT y los generadores de esqueletos de componentes disponibles).

Perl Se ha usado el lenguaje de programación Perl y sus herramientas de desarrollo e implementación para desarrollar varias tareas, como el desarrollo del generador de esqueletos de componentes CoolBOT y la generación de *wrappers* de los tipos primitivos de C++, para incluirlos en una factoría de clases.

bash Realización de *scripts* para las tareas de desarrollo.

GDB Depurador de GNU, o *GNU Debugger*, aunque suele usarse el entorno gráfico **kgdb**; se usa integrado dentro de **KDevelop**.

firefox Navegador web para consultas en Internet, especialmente de manuales de referencia de programación. Para esto también se usa el comando **man** y la ayuda integrada en **KDevelop**.

konqueror Exploración de ficheros en pestañas y posibilidad de navegación web.

kontact Gestión de contactos y correo electrónico, mediante **kmail**. Esto se usa para la comunicación con los tutores y los alumnos de proyectos complementarios a éste (véase el Apéndice A).

konsole Emulador de terminal usado para ejecutar comandos UNIX; normalmente se usa la versión integrada en los IDEs y en salvadas ocaciones se usa directamente el terminal.

wget Descarga de sitios con documentación, usando el parámetro **-r**.

KHexEdit Visor hexadecimal con interfaz gráfica utilizado en las pruebas de la librería de serialización XDR desarrollada. Como alternativa en modo texto se puede usar **xxd**.

Valgrind Herramienta para el análisis y detección de fallos en el software. Se ha empleado para detectar problemas como el *memory leak* en *Sick***AUV**.

4.1.2. Sistema

El sistema *Sick***AUV** desarrollado requiere de una serie de librerías software y de equipamiento lógico o información para su funcionamiento. Estos recursos son necesarios tanto para el desarrollo —i. e. procesos de compilación, construcción, depuración, etc.— como para la ejecución del sistema. En el caso del desarrollo se necesitarán las versiones de desarrollo de las librerías, mientras que para la ejecución bastará con la librería compilada. A continuación se enumeran y describen brevemente las librerías y la información que requiere el sistema *Sick***AUV**.

4.1.2.1. Librerías

Se han usado las siguientes librerías (bibliotecas) y entornos o *frameworks* de programación, para el diseño e implementación del sistema:

CoolBOT *Framework* de Programación Orientada a Componentes para sistemas robóticos [Domínguez Brito, 2003], que sirve de base para el diseño e implementación del sistema *Sick***AUV** propuesto (véase la Sección E.1).

STL *Standard Template Library* para programación con C++. Se han usado diferentes tipos de contenedores, v. g. **vector**, **map**, **set**, etc., y se ha hecho uso de funciones **hash** en los mapas asociativos. Se han usado las librerías habitualmente disponibles en las distribuciones GNU/Linux, concretamente la *GNU Standard C++ Library v3*, disponible como **libstdc++6** (versión 4.2.3-2ubuntu7) en los repositorios oficiales de Ubuntu, que incluyen una implementación de la STL estándar y completa.

Boost Librerías portables para C++ [Dawes et al., 2006], que aportan multitud de herramientas para la programación en C++. Se han estudiado las librerías que ofrece y en *Sick***AUV** se ha hecho uso de algunas, como **Functional/Hash**, **Preprocessor**, **TR1**[1], **Thread**, etc.

Player/Stage Plataforma de desarrollo para sistemas robóticos que ofrece un servidor para el control de robots. Dispone de una interfaz simple para los dispositivos sensoriales y actuadores a través del protocolo de comunicación IP [Gerkey et al., 2006]. Es utilizado para adaptar los *drivers* de dispositivos físicos al sistema *Sick***AUV**, aprovechando la infraestructura de abstracción hardware de *Player*. El simulador *Stage* no se ha usado en el desarrollo de *Sick***AUV**.

netCDF Se usa el formato binario *netCDF* y la librería **netcdfg** para la lectura de los datos almacenados en ficheros de información batimétrica, meteorológica o de muestras sensoriales.

***Drivers* de dispositivos** Se dispone de los *drivers* para la brújula TCM-50 de PNI Corp. y el giróscopo VG400 de Crossbow, ambos desarrollados en lenguaje C. Éstos son adaptados a *Player*.

gpsd Librería y servidor para el uso de receptores GPS, usada para obtener la información del modelo HI302CF de Haicom.

Video4Linux (V4L) *Application Programming Interface* (API) de captura de vídeo para Linux. Se ha usado la versión 2 (V4L2) para realizar pruebas de obtención de imágenes de una cámara web.

festival Servidor de síntesis de voz para Linux, para el que se ha desarrollado un cliente integrado como un dispositivo de *Player*.

acpitool Herramienta para obtener información de sensores mediante la *Advanced Configuration and Power Interface* (ACPI). Se ha estudiado para tomar valores de temperatura de la CPU y carga de las baterías, pero no se ha adaptado a *Player*.

[1] Se usan las extensiones del *Library Technical Report* (TR1) del *C++ Standards Committee* incluidas en las librerías de C++ que vienen instaladas en las distribuciones Linux usadas.

Xerces-C++ Analizador sintáctico (*parser*) de XML con soporte para validación. Tiene soporte para DOM, SAX y SAX2 (véase la Sección E.3). Se ha hecho uso de SAX2 para la carga de los planes de la misión.

pthreads Librería de hilos POSIX utilizada en casos puntuales, ya que por norma general se usa el soporte que ofrece CoolBOT.

Socket++ Librería de manejo de *sockets* para C++. Se usa como base para la implementación de *streams* de C++ realizada, adoptando la misma interfaz que los *sockets* de Java.

log4cxx Librería para realizar el registro del sistema. Dispone de una especificación en XML que se incluye en la misión como plan de *log* (PdL). Se ha usado la versión más actual, obtenida directamente del servidor **SVN**, ya que la versión estable tiene ciertos problemas y carencias.

4.1.2.2. Información. Equipamiento Lógico

El sistema requiere información para la realización de ciertas tareas, como es el caso de la batimetría y la información meteorológica. A continuación se indica de dónde se ha obtenido este equipamiento lógico y sus características (véase el **??** para un estudio más detallado):

Batimetría Ficheros con la profundidad para las diferentes latitudes y longitudes de una región. Estos ficheros disponen de una cierta resolución en el muestreo de la profundidad y suelen usar el formato *netCDF*. Se han usado los ficheros de *Unidata* como ejemplos, los cuales están diponibles en:

`http://www.unidata.ucar.edu/software/netcdf/examples/files.html`

Se ha desarrollado un *driver* para *Player* que permite obtener la información batimétrica como si se tratase de un profundímetro ubicado en el fondo del mar.

Parámetros físico-químicos *Unidata* también ofrece ficheros en formato *netCDF* con datos meteorológicos de parámetros físico-químicos, desde la misma web que la batimetría.

4.1.3. Mantenimiento

Durante la elaboración del proyecto se han realizado ciertas tareas de mantenimiento o administración. Se trata habitualmente de la realización de copias de seguridad, organización de la información, control de versiones y publicación de los estudios para su revisión por parte de los tutores. Las herramientas usadas para ello son las siguientes:

CVS Al inicio del proyecto se hizo uso de la herramienta de control de versiones *Concurrent Version System* (CVS).

Subversion La herramienta de control de versiones CVS se sustituyó por Subversion, que dispone de más funcionalidades.

kdeSVN Interfaz gráfica para gestionar el control de versiones, que se ha aplicado al sistema *Sick***AUV**. También se han usado las herramientas de control de versiones integradas en **KDevelop**, que soportan tanto CVS como Subversion.

webSVN Aplicación para la visualización web del control de versiones de los repositorios de Subversion; requiere de un servidor web —se ha usado Apache.

kompare Potente utilidad gráfica para la comparación de diferentes versiones de ficheros o carpetas. También se ha empleado la alternativa en modo texto que constituyen la pareja **diff**/**patch**, así como las búsquedas con el comando **grep**.

cron Se han programado tareas de mantenimiento mediante la utilidad **cron**, fundamentalmente copias de seguridad incrementales.

tar Utilidad para empaquetar múltiples ficheros y carpetas en uno sólo. Se ha empleado en los *scripts* de copias de seguridad programados con **bash** y **cron**.

ark Interfaz gráfica para gestionar ficheros comprimidos. También se ha usado **7-zip** y se han instalado las extensiones para soporte de fichero **rar** y **ace**; se trata de los paquetes **rar**, **unrar**, **unace** y **unace-nonfree**, dentro de los repositorios de Ubuntu.

aptitude Utilidad de línea de comando para la realización de actualizaciones del software GNU/Linux; también se suele conocer como **apt-get**.

Servidor de Ficheros Se ha usado el servidor **Pure-FTP** como *File Transfer Protocol* (FTP) para el paso de ficheros entre equipos. También se han usado las carpetas compartidas de Windows® y Samba. Además, mediante el servidor *Secure SHell* (SSH) **OpenSSH** se ha dispuesto de acceso a línea de comandos y al mismo tiempo a la transferencia segura de ficheros, mediante el protocolo **fish** o el comando **scp**. También se ha usado el servidor web **Apache** y el Sistema de Gestión de Contenido, o *Content Management System* (CMS), *moodle*; se ha experimento con **Plone** como alternativa a éste. La herramienta **moodle** se ha usado para facilitar la publicación de los estudios realizados y la coordinación de este Proyecto Fin de Carrera (PFC) con otros dos proyectos complementarios: **planificador** y **simulador** (véase el Apéndice A).

Otros comandos UNIX Se han empleado otros muchos comandos UNIX en las tareas de mantenimiento, como **mount**, **df**, etc.

4.1.4. Documentación

Se ha elaborado documentación para los estudios realizados a lo largo del PFC y para editar la memoria del mismo. Se han usado herramientas de procesamiento de texto, diagramación y tratamiento de imágenes.

GIMP *GNU Image Manipulation Program* es un programa de edición de imágenes GNU/Linux, utilizado para la preparación de imágenes.

Photoshop Programa de edición de imágenes, utilizado para la preparación de imágenes, aplicando filtros de limpieza de ruido.

ImageMagick Software para la creación, edición y composición de imágenes que dispone de utilidades desde la línea de comandos. Ha sido utilizado el comando **convert** para convertir las imágenes al formato PostScript encapsulado (EPS), que es el mejor soportado por LaTeX.

MS Office Suite ofimática, utilizada para la generación de varios documentos a lo largo del desarrollo del proyecto. Se ha usado **MS Word**, **MS PowerPoint** y **MS Visio**. Éste último se ha elegido para la edición de diagramas por generar un mejor acabado. Algunos diagramas se han realizado directamente usando paquetes de LaTeX, como **gastex**.

OpenOffice.org Suite ofimática multiplataforma, también utilizada para el procesamiento de texto y la generación de diagramas.

KOffice Suite ofimática que forma parte del entorno de escritorio KDE. También se ha utilizado para el procesamiento de texto y diagramas, además de la edición de fórmulas para LaTeX.

doxygen Generador de documentación para varios lenguajes de programación, entre los que se incluye C++. Se ha usado para documentar el código, aunque sólo parcialmente. También se ha usado la herramienta gráfica **doxygen-gui** para configurar la ejecución de **doxygen**; esto también puede hacerse directamente desde **KDevelop**.

krecordmydesktop Herramienta para la grabación de vídeos de escritorio, utilizado como registro de las pruebas realizadas.

LaTeX Se ha usado LaTeX 2_ε para la realización de la memoria del PFC, siguiendo los numerosos manuales disponibles [Oetiker et al., 2006, Lamport, 2000] y normas de estilo [Martínez de Sousa, 1995].

teTeX Distribución TeX para sistemas compatibles UNIX. Se comenzó utilizando esta distribución como base de LaTeX.

TeXLive Distribución TeX para sistemas compatibles UNIX. Ésta es la distribución que se ha terminado usando como base de LaTeX.

MiKTeX Distribución TeX para Windows®. Se ha usado para generar documentación en LaTeX desde esta plataforma.

Scientific Workplace Aplicación *What You See Is What You Get* (WYSIWYG) para generar documentación en LaTeX bajo la plataforma Windows®. Se comenzó trabajando en ella, pero finalmente ha resultado más apropiado no usar un WYSIWYG.

LyX Aplicación WYSIWYG multiplataforma para generar documentación en LaTeX. Se experimentó con ella en Linux, pero finalmente se optó por no usar un WYSIWYG.

Kile IDE para el desarrollo de documentación en LaTeX, que incluye editor con resalte de sintaxis y soporte para la generación de bibliografía con BibTeX, que ha sido la opción adoptada [Ataz López, 2006]. Se usa la versión española de **Kile** (**kile-i18n-es**).

KBibTeX Editor gráfico de bibliografía para BibTeX, que facilita la edición de los ítems bibliográficos.

kdvi Visor de ficheros *DeVice Independent* **DVI**, que son producidos por LaTeX. Este visor tiene problemas con la visualización de los colores, por lo que las imágenes se ven mal y los entornos de algoritmos se ven completamente en negro.

KGhostView Visor de ficheros PostScript, generados a partir de los ficheros **dvi** con el comando **dvi2ps**.

kpdf Visor de ficheros *Portable Document Format* (PDF), para la consulta de fuentes bibliográficas. También se usa para visualizar el resultado final de la memoria, tras convertirla a este formato con el comando **ps2pdf**; al pasar de **DVI** a **PS** y finalmente a **PDF** se consigue mantener los hiperenlaces, para lo que se usa el paquete **hyperref**.

Adobe Acrobat Reader Visor de ficheros PDF, necesario para poder visualizar extensiones como los comentarios de revisión.

latex-ucs Soporte para el sistema de caracteres *Universal Character Set* (UCS). Tanto la memoria como el código fuente del sistema se han realizado utilizando la codificación *8-bit Unicode Transformation Format* UTF-8.

Paquetes de LaTeX Se han usado múltiples paquetes de LaTeX, obtenidos de *the Comprehensive TeX Archive Network* (CTAN), como **SIunits** (Sistema Internacional de unidades), **lspace** (páginas apaisadas) y un largo **etcétera**.

myspell-es Corrector ortográfico GNU, empleado junto con **Kile**.

4.1.5. Software adicional. Pruebas y estudios

Durante el desarrollo del proyecto se ha realizado una serie de estudios y pruebas de diversas tecnologías y herramientas software que no han sido incorporadas en el diseño o implementación del sistema, pero que merecen ser mencionadas.

XDR La librería de *eXternal Data Representation* (XDR) presente en las librerías de desarrollo de GNU/Linux se usó en un principio (véase la Sección E.9), pero finalmente se optó por una implementación propia para *Sick***AUV**, usando la interfaz de los *streams* de C++. La utilidad de este formato de representación de datos es su independencia de la plataforma, haciéndolo especialmente útil para la serialización en las comunicaciones, usando un formato binario.

SOAP *Simple Object Access Protocol* (SOAP) es un protocolo que define cómo dos objetos en diferentes procesos pueden comunicarse por medio del intercambio de datos XML. Esto permite una serialización en formato texto, la cual se estudió como alternativa a XDR, pero finalmente no se utilizó.

libxml2 Librería para manejar ficheros XML, que constituye una alternativa a Xerces-C++. No se usó, en beneficio de Xerces-C++, porque su API es menos amigable y no hay un soporte para la validación con esquemas XML (XSD).

Prolog El lenguaje Prolog, junto con el intérprete **GNU Prolog**, se usaron para realizar algunas pruebas de implementación de Análisis Dimensional.

Marine GNC *Toolbox* de MATLAB para el Guiado, la Navegación y Control, o *Guidance, Navigation and Control* (GNC), de vehículos marinos. Dispone de herramientas de simulación, modelos de vehículos, resolución de ecuaciones hidrodinámicas, etc.

Forma parte de los resultados de los estudios de [Fossen, 2002]. Se ha usado como consulta, pero resultará de gran utilidad en el desarrollo de los subsistemas de guiado, navegación y control del sistema *Sick***AUV**.

ACE La librería *ADAPTIVE Communication Environment* (ACE) es un proyecto que ofrece un entorno con patrones para la programación de comunicaciones en C++ [Schmidt, 2006]. Se ha estudiado su uso en *Sick***AUV**, pero se ha descartado en beneficio de alternativas más simples.

IOStreams Librería que ofrece la interfaz de los *streams* de C++ para el uso de *sockets*; hay librerías como **Skstream**, basadas en ella. El nombre de esta librería en los repositorios software es **socket++** y ha sido la finalmente empleada en *Sick***AUV**, pues proporciona la interfaz más apropiada en C++; no obstante, se ha hecho un pequeño *wrapper* para usar una interfaz más cercana a la semántica usada en Java.

CPP streams *Wrapper* de los *sockets* de Linux para proporcionar una interfaz prácticamente igual a la de Java; no incluye la sentencia **select**.

BoostSocket Librerías de *sockets* de **Boost**.

Loki Librería software de Andrei Alexandrescu realizada como parte de su libro **Modern C++ Design** [Alexandrescu, 2001]. Utiliza la técnica de metaprogramación mediante las plantillas o *templates* de C++, para rediseñar los patrones de diseño de [Gamma et al., 2003]. Se ha usado como guía en la aplicación de algunos patrones de diseño en *Sick***AUV**, pero sin llegar a usar metaprogramación.

Crypto++ Librería para la encriptación de las comunicaciones. Se ha estudiado ligeramente, pero no se ha incluido aún en *Sick***AUV**.

PocketSphinx Software libre de reconocimiento de voz realizado por el *Sphinx Group Open Source Speech Recognition Engines* de la *Carnegie Mellon University* (CMU). Dispone de diversas versiones, donde **PocketSphinx** está orientado al reconocimiento en tiempo real. Esta versión que sustituye a **Sphinx2**, que fue la inicialmente estudiada, con vista a su adaptación a *Player*, lo cual no se ha realizado.

CppUnit Librería de apoyo para la realización de pruebas de unidad en el software desarrollado. Se ha estudiado, pero no se ha empleado.

DejaGNU Software para la realización de pruebas y test de verificación de aplicaciones. Se ha estudiado, pero no se ha empleado.

4.2. Hardware

El equipamiento hardware que se ha usado a lo largo del proyecto, para las distintas tareas que se han tenido que realizar, se muestra en las siguientes secciones. En general, para la consecución de estas tareas se ha hecho uso, en mayor o menor medida, de los siguientes equipos informáticos:

Disco Duro Externo Se dispone de un disco duro con un Sistema Operativo GNU/-Linux instalado y arrancable por USB. Esto permite su uso desde múltiples PCs.

Portátil I Modelo **ACER TravelMate 4000**, con CPU **Intel®Pentium®M** @ 1.5GHz, 1MB de caché, 512MB de RAM y 120GB de HD. **GNU/Linux Kubuntu 7.10**, para copias de seguridad.

Portátil II Modelo **HP tx1000**, con CPU **AMD®Turion™64bit × 2** @ 2.2GHz, 4MB de caché, 2GB de RAM y 160GB de HD. **Windows®Vista SP1**, para tareas puntuales de documentación.

PC Laboratorio Modelo **DELL Intel®Core™2 Duo** @ 2.40GHz, 4MB de caché y 2GB de RAM. No se usa el HD ni los sistemas operativos que tiene instalados.

PC Las Palmas Modelo **Intel®Pentium®IV** @ 1.4GHz, 256kB de caché, 256MB de RAM y 80GB de HD. *Dual boot* con **Windows®XP SP2** y **GNU/Linux Ubuntu 8.04 LTS** (actualmente).

PC Fuerteventura Modelo **Intel®Pentium®IV** @ 2.0GHz, 2MB de caché, 1GB de RAM y 160GB de HD. *Dual boot* con **Windows®XP SP2** y **GNU/Linux Kubuntu 6.10**.

4.2.1. Desarrollo

Para el desarrollo del PFC se han empleado los siguientes equipos informáticos:

Disco Duro Externo Utilizado para diseñar e implementar el sistema *Sick***AUV**, así como el formato de especificación de su equipamiento y de la misión (véase la Sección 4.2 para el detalle de las características de este equipo).

Portátil I Utilizado junto con el disco duro externo para llevar a cabo el desarrollo del PFC (véase la Sección 4.2 para el detalle de las características de este equipo). También se usa para la adaptación del receptor GPS a *Player*, ya que se necesita una conexión *Personal Computer Memory Card International Association* (PCMCIA).

PC Las Palmas Utilizado para la adatapción del giróscopo y la brújula a *Player*, ya que estos dispositivos utilizan el conector DB/9 de RS232, no disponible en el **Portátil I**.

4.2.2. Sistema

Los recursos hardware necesarios para el PFC cubren el equipamiento físico de un hipotético AUV. Aquí también podría hablarse de la construcción de un prototipo de AUV, pero eso forma parte de otros proyectos (véase el Apéndice A).

4.2.2.1. Dispositivos. Equipamiento Físico

En el Capítulo 8 se detallan estos y otros dispositivos, que suelen embarcarse en los vehículos de exploración oceanográfica. A continuación sólo se muestran aquellos que estaban disponibles para la realización del PFC, aunque no se ha trabajado con algunos de ellos:

Giróscopo vertical Modelo **VG400** de *Crossbow*. Ofrece las aceleraciones en los 3 ejes del espacio y la temperatura interna del dispositivo.

Brújula electrónica Modelo **TCM2-50** de *PNI Corp.*, que aparte de la brújula, también dispone de inclinómetros, magnetómetro y termómetro interno. El magnetómetro indica el campo magnético en los 3 ejes del espacio, lo cual permite saber si la brújula se está viendo afectada por el campo magnético que genera el equipamiento físico del vehículo.

Receptor GPS modelo **HI302CF** de *Haicom* de tarjeta receptora GPS con antena. Se usa un adaptador de Compact Flash (CF) a PCMCIA.

PC104+ PC embarcable en formato PC104+, equipado con un **Intel®Pentium®III** @ 800MHz MHz. No se ha usado ni probado en el desarrollo de *Sick***AUV**; en su lugar se usó un PC.

Adquisición de Datos Tarjeta de adquisición de datos en formato PC104+. No se ha usado en el desarrollo del proyecto.

Alimentación Tarjeta de control de alimentación. No se ha usado en el desarrollo del proyecto.

Controladora de Motores Tarjeta de control de motores de corriente continua. No se ha usado en el desarrollo del proyecto.

Motores Se adquirió una pareja de motores sumergibles, pero no se ha trabajado con ellos en el proyecto.

4.2.3. Mantenimiento

Para la realización de las tareas de mantenimiento se usará el software comentado en la Sección 4.1.3. Esto requiere el uso de varios equipos informáticos, que para este PFC han sido los siguientes:

PC Las Palmas Utilizado para realizar copias de seguridad (*backups*) y pruebas (véase la Sección 4.2 para el detalle de las características de este equipo).

Portátil I Utilizado para realizar copias de seguridad (*backups*) en su disco interno (véase la Sección 4.2 para el detalle de las características de este equipo).

Disco Duro Externo Modelo **Fujitsu SIEMENS** de 500GB, de uso doméstico y formateado con el sistema de ficheros NTFS.

4.2.4. Documentación

Para la elaboración de la documentación, tanto los estudios como la memoria del PFC, y su posterior impresión se han usado los siguientes recursos:

Portátil I Elaboración de la documentación en LaTeX, mediante **Kile** (véase la Sección 4.2 para el detalle de las características de este equipo)

Otros PCs El resto de equipos informáticos mencionados en la Sección 4.2, se ha empleado para la documentación de los estudios realizados, la construcción de diagramas y el tratamiento de imágenes.

Multifunción I Equipo multifunción modelo **HP PSC 750**, del domicilio particular.

Multifunción II Equipo multifunción de red modelo **HP C4380**, del domicilio particular; reemplaza al equipo anterior.

Impresora Laboratorio Impresora láser modelo **HP LaserJet 4200dtn**.

Material fungible Material para el análisis y estudio de las fuentes bibliográficas, así como el trabajo realizado. Se han usado folios, grapas, bolígrafos, marcadores, etc.

Capítulo 5

Plan de Trabajo y Temporización

Nadie planifica fracasar,
pero muchos fracasan al planificar

— Pierre-Augustin de Beaumarchais
1732–1799 (Dramaturgo francés
variante de la cita original)

Para la realización de este Proyecto Fin de Carrera (PFC) se elaboró una planificación de trabajo inicial, con una estimación temporal de las diferentes fases. La temporización se ajustó de acuerdo a la duración típica de este tipo de proyectos, para determinar aproximadamente la dedicación requerida por cada una de las fases o etapas del plan de trabajo. Se comenzó utilizando la herramienta de planificación **OpenWorkbench**, pero a lo largo del desarrollo del proyecto se adoptó una planificación menos esctricta, sin llevar un registro de control tan exhaustivo como el impuesto por esta herramienta y metodología de cumplimentación del trabajo realizado.

El plan de trabajo se divide en base a la metodología de trabajo mostrada en el Capítulo 3, i. e. de acuerdo con las diferentes fases de desarrollo del software. Aparte de estas fases se incluyen las etapas de obtención de conclusiones y resultados y la elaboración de la documentación. Además, la fase de mantenimiento no se considera en la planificación, ya que se realizará como parte de la gestión y administración del trabajo realizado a lo largo de todo el proyecto. La duración y las tareas realizadas en estas fases se comentan brevemente en las siguientes secciones, de las que se obtiene la distribución porcentual de la carga de trabajo, mostrada en la Figura 5.1 para cada una de ellas. En la Sección 5.7 se indica la duración completa del desarrollo del PFC, lo que permite determinar las duraciones de cada una de las tareas individualmente a partir de sus porcentajes.

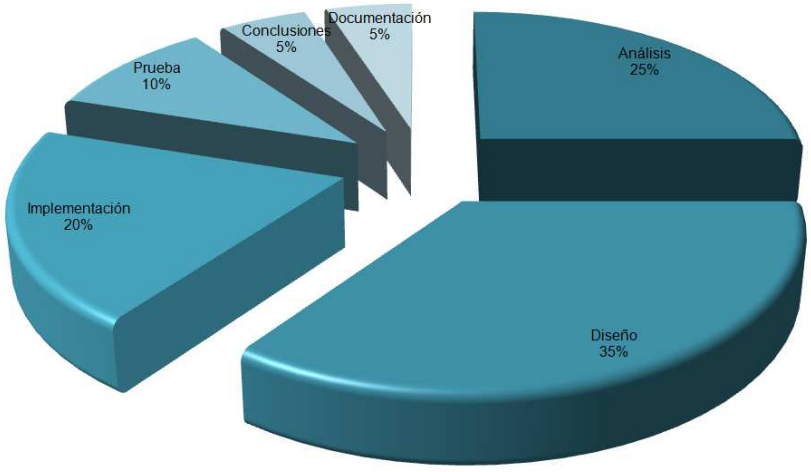


Figura 5.1: Diagrama de sectores con la distrución porcentual del tiempo dedicado a las diferentes fases del plan de trabajo

5.1. Análisis y Estudio de Casos

El desarrollo del proyecto se inicia con la recopilación y consulta de numerosa información sobre oceanografía y vehículos de exploración utilizados en este campo. A partir de toda esta documentación se inicia un análisis y estudio en detalle que abarca aproximadamente el 25 % del trabajo realizado. Se pueden distinguir las siguientes tareas de análisis:

1. Análisis del equipamiento de los vehículos de exploración oceanográfica, lo que cubre el estudio de la tipología de vehículos, el equipamiento físico y lógico (dispositivos e información) y las características de los AUVs (véase el Capítulo 7 y Capítulo 8). Esto constituye un 35 % del análisis.

2. Estudio y clasificación de la tipología de misiones que suelen realizar los vehículos de exploración oceanográfica (véase el Capítulo 11). También se realiza el estudio de arquitecturas de especificación de misiones (véase la Sección 12.1). Esto constituye un 40 % del análisis.

3. Estudio de arquitecturas de control para AUVs. Se realiza la clasificación de las mismas tanto en función de la estructura, como en base a los módulos que la componen (véase la Sección 14.1). Esto constituye un 25 % del análisis.

5.2. Diseño

A partir de los resultados del análisis realizado se lleva a cabo el diseño de la especificación del equipamiento y de la misión, así como del sistema. Esto supone sobre un 35 % del trabajo realizado. Se distinguen las siguientes tareas de diseño:

1. Diseño de la estructura de organización de la especificación del equipamiento (véase el Capítulo 9). Esto constituye un 10 % del diseño.

2. Diseño de la arquitectura propuesta para la especificación de la misión: **planes de la misión**. Se detallan las características de dicha arquitectura, los planes que la forman, los parámetros de la misión y la estructura de organización de la especificación de la misión (véase la Sección 12.2). Esto constituye un 25 % del diseño.

3. Estudio del *framework* CoolBOT, para poder realizar el diseño del sistema correctamente. Esto constituye un 5 % del diseño.

4. Diseño de la arquitectura propuesta para el sistema integrado de control de un AUV: *Sick***AUV**. Se detalla el tipo de estructura y los módulos de la arquitectura adoptada, los subsistemas que contiene, los componentes CoolBOT de los que dispone cada subsistema y el diseño orientado a objetos de los modelos de datos y las diferentes funcionalidades del sistema (véase el Capítulo 15). Esto constituye un 60 % del diseño.

5.3. Implementación

Parte del diseño realizado en la etapa anterior es implementado en mayor o menor medida, según la complejidad de cada caso. Esto supone cerca de un 20 % del trabajo realizado. Se distinguen las siguientes tareas de implementación:

1. Especificación del equipamiento, desarrollando los esquemas XML (XSD) que definen formalmente la sintaxis del lenguaje. También se elabora un conjunto de ficheros de ejemplo que constituyen una especificación completa del equipamiento de un hipotético AUV (véase el Capítulo 9). Esto constituye un 8 % de la implementación.

2. Especificación de la misión, desarrollando los esquemas XML (XSD) que definen formalmente la sintaxis del lenguaje. También se elabora un conjunto de ficheros de ejemplo que constituyen una especificación completa de una hipotética misión, con todos los planes de la misión y una tabla de parámetros (véase el Capítulo 13). Esto constituye un 17 % de la implementación.

3. Diseño e implementación de un generador de esqueletos de componentes CoolBOT, en base al estudio del *framework*. Se usará para facilitar la implementación de los componentes CoolBOT del sistema, lo que incluye los subsistemas, pues éstos se modelan como componentes CoolBOT compuestos. Esto constituye un 5 % de la implementación.

4. Estudio del *framework Player*, con vistas a la adaptación de *drivers* de dispositivos y para su integración en el *Sick***AUV**, aprovechando la abstracción hardware que éste ofrece. Esto constituye un 2 % de la implementación.

5. Estudio de *netCDF* y ficheros de batimetría, para la implementación de un programa capaz de obtener la profundidad para una latitud y longitud dadas, incluyendo la interpolación para los valores no muestreados en la información batimétrica. Esto constituye un 1 % de la implementación.

6. Adaptación de *drivers* de varios dispositivos a *Player*. En concreto se ha adaptado un giróscopo, una brújula y un receptor GPS; también se han adaptado otro tipo de dispositivos virtuales, v. g. batimetría, sintetizador de voz, cámara. Esto constituye un 4 % de la implementación.

7. Diseño e implementación de una librería de *eXternal Data Representation* (XDR) usando *streams* de C++, que se usará para las comunicaciones. Esto constituye un 1 % de la implementación.

8. Implementación de *Sick***AUV**, lo que consiste fundamentalmente en implementar los subsistemas y los componentes CoolBOT de éstos. En este proyecto se han realizado las declaraciones de los subsistemas y componentes CoolBOT, y sólo se ha desarrollado una implementación parcial de algunos de ellos, como es el caso del intérprete de planes, el gestor de disparadores de las tareas y el subsistema sensorial, que usa un componente CoolBOT denominado **sensor** capaz de muestrear datos sensoriales —simulados en la versión actual. Además, se han implementado los mecanismos de comunicación entre componentes CoolBOT, mediante un sistema basado en mensajes, servicios, comandos y pedidos, una factoría de clases para facilitar la comunicación de datos entre diferentes plataformas, comunicaciones con el exterior (con una funcionalidad mínima), registro del sistema (utilizando **log4cxx**) y otras tareas puntuales. Esto constituye un 62 % de la implementación.

5.4. Prueba del Software

Aunque la implementación realizada no es completa, la aplicación de la Aproximación Incremental (AI) como paradigma de desarrollo software (véase la Sección 3.1), ha permitido la realización de varias pruebas de integración de los módulos del sistema *Sick***AUV**. Aparte de estas pruebas, también se han probado las especificaciones del equipamiento y de la misión. Esto supone cerca de un 10 % del trabajo realizado. Se distinguen las siguientes pruebas:

1. Carga y validación XML de los planes de la misión, utilizando el *parser* SAX2 (véase la Sección 4.1.2). En el caso del equipamiento esto se ha hecho con herramientas de validación XML fuera del sistema, porque *Sick***AUV** no dispone del modelo de datos del equipamiento. Se ha elaborado la especificación de una misión representativa para realizar estas pruebas, tal y como se muestra en el Capítulo 16. Esto constituye un 25 % de las pruebas del software.

2. Prueba de los dispositivos adaptados a *Player*. Esto constituye un 7 % de las pruebas del software.

3. Prueba del generador de esqueletos de componentes CoolBOT desarrollado; el uso del generador para el desarrollo de *Sick***AUV** ha constituido la batería de pruebas fundamental. Esto constituye un 5 % de las pruebas del software.

4. Pruebas de unidad de los diferentes elementos del sistema *Sick***AUV**, v. g. serialización XDR, comunicaciones entre componentes, comunicaciones con el exterior, etc. Esto constituye un 20 % de las pruebas del software.

5. Pruebas de unidad de los componentes CoolBOT desarrollados —aunque no dispongan de una funcionalidad completa. Esto constituye un 10 % de las pruebas del software.

6. Pruebas de integración de los componentes CoolBOT para constituir los subsistemas de *Sick***AUV**. También se incluyen las pruebas de integración del resto de elementos del sistema. Esto constituye un 15 % de las pruebas del software.

7. Interpretación y gestión de disparadores de una misión simple. Concretamente se trata de la interpretación de un único plan, con una especificación simple. Esto constituye un 25 % de las pruebas del software.

5.5. Análisis de Resultados y Conclusiones

A la luz de las pruebas realizadas y los resultados obtenidos en la Parte IV, se elaboran las conclusiones del trabajo realizado. Se determina hasta qué punto se han cumplido los objetivos del PFC y las contribuciones que se han hecho al campo de estudio. Esto queda recogido en el Capítulo 17 y supone el 5 % del trabajo realizado.

5.6. Documentación

A lo largo del desarrollo del proyecto se va generando documentación de los estudios realizados y se confecciona la memoria del PFC. Esto supone algo más del 5 % del trabajo realizado.

5.7. Temporización

El proyecto se comenzó en diciembre de 2006 y su desarrollo ha llevado cerca de 2 años. Mediante este valor y los porcentajes de tiempo que se han empleado en las tareas enumeradas en las secciones anteriores, se puede determinar la duración aproximada de cada una de las mismas. Si consideramos una jornada laboral de 8h y un horario de trabajo de 5 días semanales, dado que 1 año tiene aproximadamente 52 semanas, estaríamos hablando de unas 4160h trabajadas (véase el cálculo realizado en la Ecuación 5.1). Esto sería aproximadamente 4 veces más de la duración habitual de un PFC, si fijamos ésta en 6 meses.

$$8\text{h} \times 5\text{días} \times 52\text{semanas} \times 2\text{años} = 4160\text{h} \tag{5.1}$$

$$4160\text{h} \times 66\,\% = 2745\text{h} \tag{5.2}$$

Sin embargo, durante este tiempo, la elaboración del PFC se ha compaginado con cursos de idiomas, estudios univesitarios y la colaboración en una beca en el campo de las boyas oceanográficas. Por ello parece razonable pensar que la dedicación real al proyecto es inferior a la calculada. En principio, se considera que un 66 % de este tiempo se corresponde con el desarrollo del PFC, de modo que estaríamos hablando de 2745h, i. e. el equivalente a un año y un cuatrimestre. El excesivo tiempo empleado sólo puede ser justificado por la complejidad del PFC.

Parte I

Equipamiento

Capítulo 6

Definición

La pregunta de si un computador puede pensar no es más interesante que la pregunta de si un submarino puede nadar.

— Edsger Wybe Dijkstra
1930–2002 (Científico de la computación de origen holandés)

Entenderemos por equipamiento todos aquellos elementos que se embarcan en un vehículo como apoyo a la realización de la misión. Para determinar qué tipo de equipamiento suele emplearse, se realiza una clasificación previa de la tipología de este tipo de vehículos en el Capítulo 7.

Por norma general distinguiremos un equipamiento físico y otro lógico. Esta distinción diferencia los dispositivos físicos de la información. En cualquier caso, tanto unos como otros son imprescindibles en la realización de las misiones, en función de las funcionalidades que ésta demande. El equipamiento físico se muestra en detalle en el Capítulo 8. En cuanto al equipamiento lógico se puede definir el siguiente conjunto de datos que acostumbran en manejarse en misiones de exploración oceanográfica:

Meteorología Información de fenómenos meteorológicos, como puede ser el viento, la temperatura, etc. Incluso se pueden considerar los datos de parámetros físico-químicos que han sido muestreados en un zona, tanto espacial como temporalmente.

Corrientes Marinas La información de las corrientes marinas, así como el estado de la mar, i. e. el oleaje, son útiles para que el vehículo navegue de forma correcte. Este tipo de información se suele introducir para realizar correcciones en las ecuaciones hidrodinámicas [Fossen, 1994, Fossen, 2002].

Batimetría La información batimétrica proporciona información relativa a la profundidad del fondo oceánica para cada punto del mar. Se suele disponer de una rejilla de latitud y longitud con una resolución de muestreo dependiente de la calidad de la

batimetría. Existe fuentes que proporcionan batimetría gratuita de determinadas zonas del mundo.

Este tipo de información acostumbra a ser de un volumen bastante grande. Por este motivo suele almacenar en ficheros con un formato específico para ello. Es el caso de **netCDF**, que ofrece un almacenamiento binario vectorial de los diferentes parámetros de información que se almacenen.

En base al estudio del equipamiento que se realiza en las siguientes secciones, se ha elaborada una especificación del equipamiento. Ésta usa un lenguaje basado en XML, el cual se define mediante esquemas XML (XSD). Además, se ha diseñado y organizado en una estrectura de directorios según el tipo de dispositivo. En el Capítulo 9 se comenta esta estructura brevemente, como conclusión al análisis del equipamiento de los vehículos de exploración oceanográfica y, por ende, de los AUVs.

Capítulo 7

Vehículos de Exploración Oceanográfica

¿Que si creo que las máquinas llegarán a pensar?
Téngalo por seguro.
Yo soy una máquina, usted es una máquina y ambos pensamos, ¿no cree?

— Claude Elwood Shannon
1916–2001 (Ingeniero eléctrico y matemático. "Padre de la teoría de la información")

Dentro de los numerosos tipos de vehículos de exploración oceanográfica, en este proyecto nos centraremos en el estudio de los vehículos no tripulados. El término vehículos no tripulados hace referencia a cualquier tipo de vehículo sin un tripulante humano, pudiendo realizarse una clasificación en función del medio en el que operan: **aéro**, **terrestre** o **submarino**.

Vehículo Aéreo No tripulado O *Unmanned Aerial Vehicle* (UAV), que surgieron durante la 2ª Guerra Mundial con fines militares, pero más recientemente se ha suscitado mayor interés por modelos avanzados, como demuestran los proyectos siguientes:

Pioneer UAV Usado por la Marina estadounidense en la Operación Tormenta del Desierte y todavía en uso por la Cuerpo Marino.

Predator UAV Usado en Bosnia.

Teledyne Ryan Tier II+ Global Hawk Destinado a misiones de reconocimiento, con capacidades de operación a gran altitudes y larga duración bajo cualquier condición meteorológica.

Outrider Usado con fines tácticos por la Armada, Marina y Cuerpo Marino estadounidense.

En el caso de España, durante el segundo trimestre de 2008, el Ministerio de Defensa adquirió 31 UAVs: 4 son del modelo *Searcher Mark II J*, del fabricante israelí IAI (Israel Aerospace Industries Ltd.), y 27 mini-UAV del modelo *Raven 11B*, de fabricación nacional por la empresa española *Aerlyper*.
Aunque los proyectos más serios con UAVs suelen tener fines militares, también existen diversos proyectos de investigación o *amateur*.

Vehículo Terrestre No tripulado O *Unmanned Ground Vehicle* (UGV), que incluyen los vehículos teleoperados o autónomos, i. e. robots móviles. Se usan en aplicaciones militares y civiles, destacando las investigaciones y desarrollos de DARPA (*Defense Advanced Research Projects Agency*) y la NASA (*National Aeronautics and Space Administration*). Son muy comunes como plataformas de prueba para algoritmos de localización y construcción de mapas, como SLAM (*Simultaneous Localization and Mapping*) [Newman and Leonard, 2002, Karlsson et al., 2005, Sattar, 2005].

Vehículo Submarino No tripulado O *Unmanned Underwater Vehicle* (UUV), que incluyen los vehículos submarinos operados remotamente y los autónomos:

Vehículo Operado Remotamente O *Remotely Operated Vehicle* (ROV), está *atado* (*tethered*) a un buque en superficie mediante un cable umbilical que transmite las señales de control y energía al vehículo, que devuelve imágenes y otros datos sensoriales.

Un vehículo submarino sin umbilical no tripulado, o *Unmanned Untethered Underwater Vechicle* (UUUV), puede pertenecer a dos grupos diferentes en base a sus capacidades de maniobrabilidad:

Vehículo de crucero O *cruising vehicle*, que realiza misiones en mar abierto y suele usarse para exploración, búsqueda, localización de objetos, etc. Para estas tareas el diseño del vehículo sólo requiere 3 grados de libertad: **longitudinal**, **rumbo** (*yaw*) y **cabeceo** (*pitch*).

Vehículo *hover* O *hovering vehicle*, que es capaz de realizar la maniobra de *hovering*[1], i. e. que puede matenerse en una posición fija. Estos vehículos, usados para la inspección detallada y trabajos físicos sobre objetos fijos, sustentan estaciones robóticas a profundidad constante para que realicen sus tareas.
Para la realización de estas tareas se requiere la generación de propulsión dinámica que genere fuerzas en las tres direcciones ortogonales xyz del espacio, así como en las orientaciones de **rumbo** (*yaw*) y, en ocasiones, **cabeceo** (*pitch*). ortogonales

También es común encontrar la siguiente clasificación:

Planeadores submarinos Más comúnmente conocidos como *glider*, son pequeños submarinos autónomos capaces de explorar amplias zonas oceánicas [Díaz, 2007]. El

[1] De acuerdo con el *Merriam-Webster Online Dictionary*, el significado del término *hover* que se aplica a la maniobrabilidad de los UUUV es:

to remain suspended over a place or object

Se trata, por tanto, de la capacidad de mantenerse suspendido en una posición fija.

planeador utiliza su forma hidrodinámica, pequeñas alas y cambios de flotabilidad para inducir movimientos horizontales en la columna de agua (véase la Figura 7.1 (b)). Mientras está en superficie dispone de comunicación satelital con el laboratorio que puede estar localizado en cualquier parte del planeta, lo que le permite recibir la misión a realizar vía satélite. El vehículo realiza la misión desplazándose con un movimiento en *zigzag* en la columna de agua con una velocidad horizontal neta de 1.5km/h, gracias a que cuando el vehículo altera su flotabilidad, la forma hidrodinámica y las alas experimentan un empuje que permite un desplazamiento horizontal efectivo.

Este procedimiento ofrece una forma de desplazamiento con un mínimo consumo de energía y por tanto otorga al planeador una gran autonomía en el mar, que puede llegar a ser de hasta tres meses. El coste de operación es mínimo cuando se compara con el coste económico que implicaría realizar la misión con un buque oceanográfico.

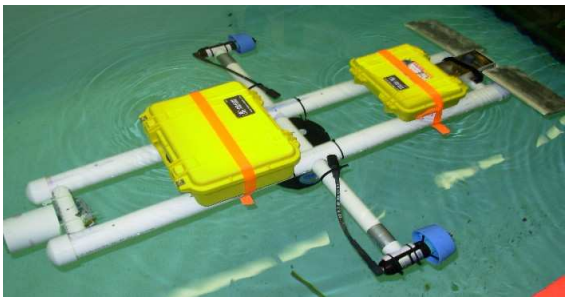
Sin embargo, la maniobrabilidad y potencia de estos sistemas es limitada en aquellas regiones donde existen fuertes corrientes, v. g. zonas costeras como playas, bahías, estuarios etc., donde forzamientos como las mareas pueden producir corrientes de gran intensidad y complejidad espacial y temporal. En estas zonas los planeadores submarinos no disponen de la potencia suficiente para contrarrestar dichas corrientes, siendo su operación inviable si la corriente supera el nudo y medio de velocidad.

El empleo de planeadores submarinos es todavía escaso. Esto es debido al carácter reciente de esta tecnología. Actualmente en Europa existen tan sólo 10 unidades en activo, dos de las cuales operan en el IMEDEA (Instituto Mediterráneo de Estudios Avanzados — Palma de Mallorca).

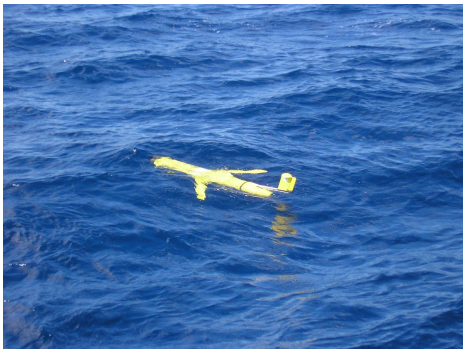
AUVs para exploración en mar abierto Constituyen el diseño de AUVs más común y disponen de un estructura hidrodinámica característica en forma de *torpedo*, como muestra la Figura 7.1 (c). Son pequeños submarinos autónomos dotados de sistemas de propulsión, generalmente eléctrica. Gran parte del volumen del submarino está ocupado por baterías que proporcionan la energía a los motores. El resto del volumen es para los sistemas de medición y navegación. Éstos últimos pueden ser de gran complejidad pues el AUV puede moverse a grandes velocidades, guiándose por estimaciones de su posición con respecto a determinados hitos. El AUV asciende a superficie cada cierto tiempo para proceder a la transmisión vía satélite de los datos.

Los AUVs no están exentos de problemáticas. Las más importantes son su autonomía y su posicionamiento. La autonomía de estas plataformas está limitada a una decena de horas. Por ello es necesario dotar a los AUVs de eficientes fuentes de energía. Se han realizado distintos prototipos usando pilas de combustible, baterías de litio o incluso la capacidad de recarga solar de baterías cuando éste está en superficie. La problemática sobre el posicionamiento del AUV cuando éste está sumergido es también de gran complejidad. Cuando un AUV toma un dato, no sólo el valor de la magnitud medida es importante, pues además debemos saber con precisión dónde se ha tomado ese dato. Este posicionamiento es muy delicado cuando uno se encuentra sumergido debido a la impenetrabilidad del GPS bajo la superficie marina. Los AUVs disponen de los llamados sistemas inerciales que miden con determinada precisión, las velocidades y aceleraciones en todos los gra-

dos de libertad del sistema. Con estas mediciones, el vehículo integra su posible trayectoria. Éste proceso no está exento de errores por lo que es necesario que el AUV emerja cada cierto tiempo para tomar un punto de referencia preciso a partir del cual integrar su trayectoria.



(a) Vehículo de *hovering*



(b) Planeador submarino (*glider*) SLOCUM existente en el IMEDEA

(c) Vehículo de crucero, con diseño en forma de torpedo. AUV Gavia

Figura 7.1: Ejemplo de diferentes tipos de AUVs

Existen diversos vehículos de exploración oceanográfica que podrán beneficiarse de parte de la especificación de las misiones que realizan los AUV, ya que se medio de actuación es similar. Destacan especialmente los siguientes:

Boya Oceanográfica Las boyas oceanográficas realizan la exploración, pero sin capacidades motrices. Se pueden distinguir dos tipos de boyas de acuerdo con la movilidad de las mismas:

Anclada Boya que se engancha a un sistema de anclaje para que se mantenga en una posición fija. Se conoce como *mooring* el proceso de anclaje o amarrado de este tipo de boyas. La aplicación de estas boyas está orientada al estudio de un zona a lo largo del tiempo; suele ser el tipo boya más utilizado.

A la deriva Se trata de una boya no anclada que se moverá a la deriva arrastrada por las corrientes marinas y los vientos. Son útiles para estudios de corrientes marinas o para abaratar costes al mismo tiempo que se estudia una zona amplia.

Barco Oceanográfico Cualquier tipo de vehículo de capacidades motrices acuáticas sobre la superficie del agua. Este tipo de vehículos se encuentran en la clasificación previamente hecha sobre la tipología de vehículos de exploración oceanográfica. Existe una variante de barco oceanográfico destacable, que se comenta a continuación:

R/V O *Research Vessel* es un buque de investigación de menor calado que un barco oceanográfico típico. Suele usarse para tareas de investigación, entrenamiento y excursiones educacionales —normalmente en aguas someras.

También existe un tipo de vehículo que opera en superficie de forma autónoma, i. e. es un AUV en esencia, pero que sólo opera en superficie —i. e. no se sumerge. Se comenta a continuación:

ASV O *Autonomous Surface Vehicle* es un vehículo con las mismas capacidades que un AUV, salvo la de inmersión. Opera en superficie y por ello suele estar equipado con dispositivos de comunicaciones que no funcionan sumegidos —v. g. GPS, comunicaciones WiFi, por satélite o radiofrecuencia, etc. En ocasiones también se denomina ASC (*Autonomous Surface Craft*) a los vehículos de este tipo.

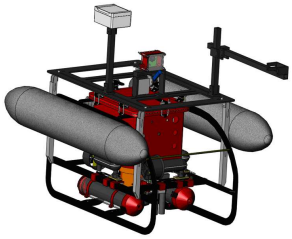
ROV Este tipo de artilugios de exploración submarina, operados remotamente gracias a un cable "umbilical" suelen tener características similares a los AUV en estructura y sistema, pero no disponen de mecanismos para la navegación de forma autónoma, por lo que se requiere de un operario en superficie para ello. Esto ya se comentó más en detalle en la clasificación inicial de UVs.

En la Figura 7.2 se muestran ejemplos de los vehículos antes mencionados: (a) boya oceanográfica, (b) barco oceanográfico, (c) R/V, (d) ASV, y (e) ROV.

(a) Boya oceanográfica. *TRIAXYS*$^{\text{TM}}$ Directional Wave Bouy

(b) Barco oceanográfico. Buque NOAA *Ronald H. Brown*

(c) R/V. *Cape Fear*



(d) ASV. SONIA ASV 2008

(e) ROV. *Hercules*

Figura 7.2: Vehículos similares a un AUV

Capítulo 8

Dispositivos. Equipamiento Físico

AUVs have many applications and consequently, the number of payloads which can potentially be carried is extensive.

— ISE Web Based AUV DesignInfo *2000* (International Submarine Engineering Ltd.)

Los vehículos de exploración submarina disponen de una tipología de dispositivos bastante común y específica, orientada a las tareas que se realizan. También se disponen de dispositivos de apoyo a la navegación y la comunicación, así como para otras funcionalidades que se desarrollan como parte de la ejecución de la misión. En las secciones siguientes se tratan los diferentes tipos de dispositivos en base a la Definición 8.1, categorizándolos según su funcionalidad y realizando una descripción individual de los mismos.

Definición 8.1 (Dispositivo (Equipamiento)). *Un dispositivo es un elemento con una funcionalidad clara, capaz de realizar una tarea física, y que forma parte del equipamiento del vehículo.*

8.1. Sensores. Dispositivos Sensoriales

En esta sección estudiaremos el equipamiento sensorial y actuador del AUV —los sensores y actuadores mínimos o clásicos que nos podemos encontrar en un AUV. Los dispositivos de comunicación no se tratan aquí, sino en la sección 8.3. Para cada dispositivo se indicará su utilidad y finalidad en el sistema, así como la interfaz de *Player* a la que dará soporte. En la sección E.2.10 se pueden consultar los detalles de la integración e implementación de éstos dispositivos —i.e. el tipo de implementación o integración que se hará y el modelo de dispositivo, si procede.

Finalidad	Sensor	Actuador
Navegación	Intrumentos de Navegación	Sistema de Impulsión
Misión	Sensores de Misión	Actuadores de Misión
Sistema Interno	Sensores Internos	Actuadores Internos

Cuadro 8.1: Nombres propios de sensores y actuadores según su finalidad

Podemos distinguir tres grupos bien diferenciados de sensores y actuadores, de acuerdo a la finalidad de los mismos —i.e. para qué se usan. En el cuadro 8.1 se muestra esta lista y los nombres propios de estos grupos de sensores o actuadores.

1. *Navegación* → Dispositivos que se emplean para la navegación. Los sensores se usarán para determinar el posicionamiento fundamentalmente y los actuadores para alcanzar el siguiente *waypoint* —i.e. navegar. A los sensores se les denominará *Instrumentos de Navegación* y los actuadores se integrarán dentro del *Sistema de Impulsión* del AUV; en el caso de los sensores, cuando se realice una misión de seguimiento de medidas, el sensor realmente usado será uno de los sensores de misión. Entre los instrumentos de navegación también se incluirán aquellos que se empleen en el guiado, i.e. dispositivos que se usen para corregir el valor del *waypoint* a alcanzar —e.g. un sónar frontal para evitar obstáculos, etc.

2. *Misión* → Dispositivos que se emplean para realizar las tareas indicadas en los planes de la misión. Lo normal es que se trate de sensores para cumplir las tareas del plan de medición. Se denominarán *Sensores de Misión* y eventualmente también se usarán para la navegación, en el caso de que ésta se defina en base al seguimiento de una medida. En cuanto a los actuadores, éstos se denominarán *Actuadores de Misión*. Aunque serán menos usuales, también podrán encontrarse en el equipamiento del AUV. Sin embargo, el control de los mismos se realizará *ad hoc* —i.e. con el software de control de los mismos desacoplado del sistema del AUV.

3. *Sistema Interno* → Dispositivos para analizar y actuar sobre elementos internos o sistémicos. Fundamentalmente se dispondrá de *Sensores Internos* para detectar posibles problemas internos observando su estado —e.g. medir la temperatura interna del AUV, garantizar la estanqueidad mediante sensores de humedad, etc. Aunque no es común, para analizar de forma homogénea sensores y actuadores, también se identifican los *Actuadores Internos*. Se encargarían de realizar tareas internas en el AUV —e.g. un actuador o sistema de evacuación de agua en caso de producirse una vía de agua.

En conclusión, disponemos de un total de seis tipos de dispositivos —sensores y actuadores—, como se observa en el cuadro 8.1. A continuación se enumeran los dispositivos concretos que se podrán incluir en el AUV. Se indicarán los más representativos en un vehículo con estas características, para conseguir ser fieles a la realidad —cuando no se disponga de los dispositivos reales se podrán *emular* con otros dispositivos o simularlos en software, de acuerdo a lo expuesto en la sección **??**. Para cada uno de los dispositivos se indicarán los siguientes datos, si proceden:

Tipo de dispositivo Nombre del tipo de dispositivo, de forma genérica.

Intefaz de *Player* a la que se da soporte Aunque relacionado con el tipo de dispositivo, se indicará a qué interfaz de *Player* dará soporte. Esto es importante de cara a la implementación.

Características Se explicarán las características del dispositivo, indicando de qué se trata y para qué sirve de forma general.

Finalidad Se citarán los motivos por los que es un dispositivo de interés en el AUV —i.e. la finalidad que cumple en el AUV o en la misión.

La explicación detallada de las interfaces de *Player*, a las que se dé soporte, se estructurará en secciones dentro de la sección E.2.4, de manuales de usuario. Cada interfaz tiene sus propias características diferenciadoras, por lo que cada una se explica por separado.

Como la clasificación se realiza considerando la finalidad del dispositivo en el AUV, las categorías no serán disjuntas. Se producirán solapamientos debido a que ciertos dispositivos tienen diferentes finalidades. En estos casos se incluirá el dispositivo en todos las categorías en que sea oportuno —e.g. el profundímetro es útil como instrumento de navegación y como sensor interno (ver secciones 8.1.1 y 8.1.3).

8.1.1. Instrumentación. Instrumentos de Navegación

Se identifican los siguientes intrumentos de navegación (en el cuadro 8.2 se proporciona una lista resumida):

1. *GPS*

   Tipo de dispositivo GPS (*Global Position System*).

   Interfaz de *Player* a la que se da soporte Dará soporte a la interfaz *gps* de *Player*. En realidad se podrían proporcionar más datos desde el GPS que los que requiere la interfaz *gps* de *Player*, pero nos ajustaremos a ésta.

   Características Un dispositivo GPS nos proporciona información sobre la posición global —en el planeta— y la hora. La posición se indica mediante la latitud, longitud y altitud, mientras que la hora/fecha se indica en UTC (*Universal Time Coordinated* – Tiempo Universal Coordinado). Posteriormente se podrá calcular el *Easting* y el *Northing* para las coordenadas UTM (*Universal Transverse Mercator* – Proyección Mercator Transversa Universal), a partir de la latitud y longitud.

   Finalidad La finalidad del GPS se centra en determinar la posición del AUV para facilitar la navegación. La hora/fecha que proporciona también será útil, pero es de menor importancia en la navegación —se podrá usar para crear sellos temporales y sincronizar datos. Adicionalmente, se pueden corregir los datos de posicionamiento del GPS mediante un DGPS (*Differential GPS* – GPS Diferencial), al cual nos podremos conectar sobre TCP/IP —incluso existen dispositivos DGPS como servidores, disponibles a través de Internet.

2. *Brújula*

   Tipo de dispositivo Brújula electrónica —o compás—, con inclinómetros.

Interfaz de *Player* a la que se da soporte Se dará soporte a la interfaz *position3d* de *Player*, para indicar la orientación del AUV, dentro de la labor de posicionamiento.

Características La brújula también dispone de magnetómetro y termómetro interno.

Finalidad Se usará para determinar el rumbo y para corregir las aceleraciones obtenidas con el giróscopo.

3. *Giróscopo*

Tipo de dispositivo Giróscopo vertical, con 2 DOF (*pan*/guiñada y *roll*/balanceo, en principio).

Interfaz de *Player* a la que se da soporte Se dará soporte a la interfaz *position3d* de *Player*, para indicar la orientación del AUV, dentro de la labor de posicionamiento. Complementa a la brújula electrónica.

Características Un giróscopo o IMU (*Inertial Movement Unit* – Unidad de Movimiento Inercial) es un dispositivo que permite determinar las aceleraciones angulares. Mediante la integración temporal de las mismas se obtiene fácilmente la orientación del vehículo en que se instala el giróscopo, conociendo la orientación inicial.

Finalidad Se usará para controlar la orientación del vehículos en los tres ejes de revolución del espacio.

4. *Profundímetro*

Tipo de dispositivo Profundímetro.

Interfaz de *Player* a la que se da soporte Se integrará en la interfaz *position3d* de *Player*, para proporcionar la componente z —altitud. No obstante, habrá que analizar si es lo correcto, pues el profundímetro proporciona la distancia a la superficie marina, i.e. la profundidad; a diferencia de lo que proporcionaría un altímetro, i.e. la distancia al fondo marino. También se debe integrar en la interfaz *aio* o *dio* de *Player* en el caso de que se *emule* el profundímetro con una tarjeta de adquisición de señal analógica o digital, respectivamente.

Características Dispositivo que determina la profundidad a partir de la presión de la columna de agua que hay sobre el vehículo.

Finalidad Permitirá determinar a qué profundidad se encuentra el AUV. Esto es útil para para complementar el posicionamiento, pudiendo navegar de acuerdo al plan de navegación y sin chocar con las formaciones del relieve del fondo marino —si se complementa con la información batimétrica.

5. *Sónar*

Tipo de dispositivo Sónar.

Interfaz de *Player* a la que se da soporte Se dará soporte a la interfaz *sonar* de *Player*.

Dispositivo	Interfaz *Player*
GPS	*gps*
Brújula	*position3d*
Giróscopo	*position3d*
Profundímetro	*position3d*
Sónar	*position2d*

Cuadro 8.2: Instrumentos de Navegación

Dispositivo	Interfaz *Player*
Termómetro	*opaque*
CTD	*opaque*
IOPS	*opaque*

Cuadro 8.3: Sensores de Misión

Finalidad Se dispondrá de un sónar frontal para la detección de obstáculos, que servirá para corregir el *waypoint* w_i a alcanzar. Esto se integrará dentro de los componentes del Subsistema de Guiado (véase la Sección 15.5).

8.1.2. Sensores de Misión

Se identifican los siguientes sensores de misión (en el cuadro 8.3 se proporciona una lista resumida):

1. *Termómetro*

   Tipo de dispositivo Termómetro —del entorno.

   Interfaz de *Player* a la que se da soporte

   Finalidad Medir la temperatura del entorno.

2. *CTD*

   Tipo de dispositivo CTD (*Conductivity, Temperature, Depth* – Conductividad (y por ende la salinidad), Temperatura, Profundidad).

   Interfaz de *Player* a la que se da soporte

   Finalidad Obtener muestras de salinidad/conductividad, temperatura y profundidad en ternas sincronizadas —tomadas prácticamente en el mismo instante temporal.

3. *IOPS*

   Tipo de dispositivo IOPS (*Inherent Optical Properties Sensor* – Sensor de Propiedades Ópticas Inherentes o Intrínsecas).

   Interfaz de *Player* a la que se da soporte

   Finalidad Medir las propiedades inherentes ópticas del entorno. Éstas suelen permitir estudios de contaminación o biología submarina.

8.1.3. Sensores Internos

Se identifican los siguientes sensores internos (en el cuadro 8.4 se proporciona una lista resumida):

1. *Baterías*

   Tipo de dispositivo Baterías —se tendrá que diferenciar entre la batería del sistema de impulsión y la de la CPU.

   Interfaz de *Player* a la que se da soporte Se dará soporte a la interfaz *power* de *Player*.

   Finalidad Determinar la carga de las baterías en cada momento para estimar cuándo se acabarán —según el consumo de los dispositivos del AUV— y actuar en consecuencia.

2. *Termómetro del AUV*

   Tipo de dispositivo Termómetro del compartimento estanco del AUV; podrá haber múltiples termómetros en diferentes zonas o componentes del AUV.

   Interfaz de *Player* a la que se da soporte

   Finalidad Verificar que la temperatura interna del AUV o de alguno de sus dispositivos es correcta —i.e. está dentro del rango de funcionamiento.

3. *Sensor de Estanqueidad*

   Tipo de dispositivo Sensor de estanqueidad —i.e. para verificar que no se ha producido alguna vía de agua. En principio, se tratará, realmente, de un sensor de humedad. Éste detectaría la presencia de agua en el interior del compartimento estanco en el caso de que se produzca alguna vía.

   Interfaz de *Player* a la que se da soporte

   Finalidad Verificar que no se han producido vías de agua en el compartimento estanco. En el caso de detectarse agua —o humedad— el AUV correría un grave riesgo y debería abortarse la misión y emerger inmediatamente.

4. *Profundímetro*

   Tipo de dispositivo Profundímetro.

   Interfaz de *Player* a la que se da soporte Se integrará en la interfaz *position3d* de *Player*, para proporcionar la profundidad. Como sensor interno sólo interesa la profundidad como tal. También se debe integrar en la interfaz *aio* o *dio* de *Player* en el caso de que se *emule* el profundímetro con una tarjeta de adquisición de señal analógica o digital, respectivamente.

   Finalidad Permitirá determinar a qué profundidad se encuentra el AUV. Esto es útil para no superar la profundidad crítica, i.e. la máxima profundidad a la que puede operar el AUV —la profundidad máxima viene determinada por la presion máxima soportable por la estructura del AUV. Esta finalidad es la que permite clasificar el profundímetro como sensor interno —no sólo con instrumento de navegación (véase la Sección 8.1.1).

Dispositivo	Interfaz *Player*
Baterías	*power*
Termómetro del AUV	*opaque*
Sensor de Estanqueidad	*opaque*
Profundímetro	*position3d*

Cuadro 8.4: Sensores Internos

Dispositivo	Interfaz *Player*
Motores	*position3d*
Superficies de control	*position3d*

Cuadro 8.5: Actuadores del Sistema de Impulsión

8.2. Actuadores. Dispositivos de Actuación

8.2.1. Actuadores del Sistema de Impulsión

Se identifican los siguientes actuadores del sistema de impulsión (en el cuadro 8.5 se proporciona una lista resumida):

1. *Motores*

   Tipo de dispositivo Tarjeta de control de motores.

   Interfaz de *Player* a la que se da soporte Se dará soporte a la interfaz *position3d* de *Player*; a la parte de actuación para comandar los motores del sistema de impulsión —en la navegación.

   Finalidad Los motores y la tarjeta de control de los mismos permitirán que el AUV pueda navegar. El sistema de impulsión que define el tipo de motores y su disposición en el vehículo determinará las entradas que éstos deben recibir. Esto permitirá la correcta navegación.

2. *Superficies de control*

   Tipo de dispositivo Superficies de control —e.g. aletas o alerones, timón de cola, etc.

   Interfaz de *Player* a la que se da soporte Se dará soporte a la interfaz *position3d* de *Player*; a la parte de actuación para comandar las superficies de control del sistema de impulsión —en la navegación.

   Finalidad Las superficies de control permitirán que el AUV pueda navegar —cambiar de rumbo u orientación. El sistema de impulsión que define el tipo de superficies de control y su disposición en el vehículo determinará las entradas que éstos deben recibir. Esto permitirá la correcta navegación y ejecución de maniobras.

8.2.2. Actuadores de Misión

Lo normal es que las misiones no incorperen este tipo de actuadores, por la complejidad de integración en un AUV genérico. Para aplicaciones *ad hoc* se podrían integrar

Dispositivo	Interfaz *Player*
Recogedor de muestras	*opaque*

Cuadro 8.6: Actuadores de Misión

este tipo de actuadores —e.g. un brazo robótico, un soplete para reparar el casco de un buque, un actuador de captura de muestras o recogedor de muestras, etc.— de forma desacoplada del sistema del AUV. Vamos a considerar sólo los siguientes actuadores de misión (en el cuadro 8.6 se proporciona una lista resumida):

1. *Recogedor de muestras*

   Tipo de dispositivo Recogedor o actuador de captura de muestras.

   Interfaz de *Player* a la que se da soporte

   Finalidad Capturar o recoger muestras con el actuador, que se depositarán en algún tipo de recipiente para su posterior análisis —tras terminarse la misión del AUV.

8.2.3. Actuadores Internos

No se identifican actuadores internos. Lo normal es que no haya actuadores internos, pues se suelen aplicar comportamientos reactivos bastante simples, sin llegar a la necesidad de incluir actuadores que realicen tareas de mantenimiento interno. Sólo en determinados casos se podría plantear algún actuador de este tipo, como podría ser un sistema de evacuación de agua en caso de producirse y detectarse una vía.

8.3. Comunicadores. Dispositivos de Comunicación

En esta sección estudiaremos el equipamiento para las comunicación, i.e. los dispositivos de comunicación de los que se dispondrá.

1. Tarjeta de red Ethernet, para comunicación en tierra
2. Tarjeta de red inalámbrica (WiFi), para comunicación durante la misión sin inmersión
3. Emisor/Receptor o equipo de radiofrecuencia
4. Emisor/Receptor o equipo para comunicación por satélite
5. Módem acústico

Capítulo 9

Especificación del Equipamiento

The key issues that determine the characteristics of the AUV and guide the system design are:
- *What sensors or other hardware must the AUV carry as payload?*
- *At what depth will the AUV operate?*
- *At what speed will the AUV operate?*
- *For how long will the AUV operate?*

— ISE Web Based AUV DesignInfo *2000* (International Submarine Engineering Ltd.)

Se pretende disponer de una definición de un AUV, que recoja los siguientes aspectos:

1. Características físicas del AUV.
2. Sistemas del AUV y características de éstos.

La definición permitirá:

1. Disponer de una definición formal en un lenguaje estructurado como es XML.
2. Facilitar y flexibilizar la definición de las características del AUV, sus sistemas y los elementos de éstos.
3. Jerarquizar y modularizar los distintos elementos para poder reaprovechar los distintos componentes.

Las características o criterios de calidad que rigen la definición del AUV son los siguientes:

1. Modularidad, definiendo distintos componentes en ficheros XML separados.
2. Flexibilidad, para que el mantenimiento y los cambios de en la definición sean de fácil realización y reducir los «efectos secundarios».

3. Unicidad en la identificación de los distintos componentes, para que en el reaprovechamiento esté controlado el versionado y no haya redundancias incontroladas.

9.1. Desarrollo

La definición del AUV se realiza en un árbol de directorios con ficheros de definición en formato XML, acompañados de esquemas XML (en fichero XSD) para permitir la validación de los ficheros XML de acuerdo a las características admitidas según el esquema usado por cada uno de éstos.

9.1.1. Árbol de Directorios

La raíz del árbol de directorios para la definición del AUV es una carpeta con el nombre auv. Opcionalmente se podría disponer del árbol de directorios en un fichero empaquetado (y opcionalmente comprimido) como el formato tar.

En la carpeta raíz auv se tienen los ficheros en los que se definen las características principales del AUV y sus sistemas.

El árbol de directorios, dentro de la carpeta auv, es el siguiente:

1. *actuador* → Sistema actuador, con actuadores incluidos en el AUV, normalmente orientados a la misión.
    a) *actuadores* → Definición de los distintos actuadores disponibles.
2. *alimentacion* → Sistema de alimentación.
    a) *baterias* → Definición de las características de las distintas baterías, incluyendo paneles solares.
3. *comunicacion* → Sistema de comunicación.
    a) *dispositivos* → Definición de las características de los dispositivos de comunicaciones.
4. *impulsion* → Sistema de impulsión.
    a) *mandos* → Definición de los mandos o superficies de control de la pose para la maniobrabilidad y navegación del vehículo.
    b) *motores* → Definición de los motores del sistema de impulsión.
5. *procesamiento* → Sistema de procesamiento, que tal vez no sea de gran interés en la definición.
    a) *cpus* → Definición de las CPUs.
    b) *memorias* → Definición de las memorias, tanto volátiles como no (almacenamiento).
    c) *tarjetas* → Definición de las tarjetas u otros elementos del sistema de procesamiento, como tarjetas de adquisición de señal.
6. *sensorial* → Sistema sensorial, que es una agrupación jerarquizada de los distintos dispositivos de medida.

a) *instrumentacion* → Definición de instrumentos empleados para la navegación instrumental del AUV.

b) *interno* → Definición de sensores de carácter interno, para la medida de variables o aspectos internos del AUV. Están orientados a vigilar por la seguridad y el correcto funcionamiento del vehículo.

c) *mision* → Definición de sensores de impuestos por la misión, para medidas del entorno, de carácter externo.

El árbol mostrado define los distintos sistemas del AUV e internamente se dispone de los elementos de éstos sistemas.

En el caso de trabajar con un sistema de ficheros con limitaciones se seguirá un convenio para simplificar el árbol de directorios definido.

Las posibles limitaciones que se contemplan son:

1. El número máximo de caracteres para directorios y ficheros.
2. El número máximo de caracteres para las rutas absolutas.
3. La profundidad máxima del árbol de directorios.

El convenio que se establece es el siguiente:

1. Se usarán sólo los 3 primeros caracteres del directorio o del nombre de los ficheros (se intentará mantener la extensión). Con esto se intenta solventar las limitaciones 1 y 2, de la lista anterior.
2. En el caso de existir algún problema con el árbol de directorios habrá que modificar el árbol de directorios; en principio no hay ninguna medida propuesta.

9.1.2. Formato de Especificación. XML y Esquemas XML (XSD)

Se usa el lenguaje o metalenguaje XML para definir el AUV, sus sistemas y los elementos que componen éstos. Para definir el lenguaje concreto que define cada componente, se usan esquemas XML, en formato XSD; existe la alternativa de los DTD, pero no usan sintaxis XML, por lo que son una peor opción hoy en día.

Los esquemas XML definen las restricciones en términos de las etiquetas que deben estar presentes en el fichero XML, así como la cardinalidad de éstas. Esto permite validar los ficheros XML con los esquemas a los que éstos se acogen; a parte de que de por sí deben ser ficheros XML bien formados.

Para cada carpeta del árbol de directorio nos encontraremos un esquema XML (XSD) en el que se especifican las restricciones de los ficheros XML del mismo directorio, que harán uso de este esquema.

Se pueden consultar directamente los ficheros XML y los esquemas XML (XSD). Existen aplicaciones como *Stylus Studio 2006 XML Enterprise Edition*, que permite la edición de XML y formatos relacionados (como XSD), la visualización del árbol DOM (Document Object Model) y la visualización en diagramas.

La especificación del equipamiento que se ha desarrollado, constituye una aproximación completa pero inicial. El sistema desarrollado aún no dispone de un modelo de datos para representar todo el equipamiento definido, por lo que es posible hacer adaptaciones en la especificación si se estima oportuno. En cualquier caso, como se ha comprobado

que la especificación del equipamiento es correcta, mediante la validación XML, se puede considerar que para un primer diseño del vehículo y la integración del sistema en él, son posibles sin mayores problemas.

Parte II

Misión

Capítulo 10

Definición

A mission (is) defined by the user,
resulting in a high-level definition of the task to be performed by the robot...

— M. João Rendas y Emmanuel Turcci
1998 (Investigadores Laboratorio I3S (CNRS-UNSA))

Las misiones cubrirán las tareas típicas que puede efectuar un robot o vehículo con las capacidades y características de un AUV (véase la Definición 10.1). Las misiones pueden utilizarse por otros vehículos, destacando especialmente los comentados en el Capítulo 7, por las similitudes que comparten con los AUV. A continuación se comenta hasta qué punto estos vehículos aprovechan la misión.

Boya Oceanográfica Las boyas oceanográficas podrán aprovecharse de las misiones de un AUV, exceptuando lo referido a la navegación y guiado, ya que una boya carece de capacidades motrices[1].

Barco Oceanográfico Cualquier tipo de vehículo con capacidades motrices acuáticas sobre la superficie del agua, pero sin la posibilidad de inmersión, podría aprovechar las misiones en prácticamente toda su amplitud; sin este apoyo sólo sería capaz de una navegación bidimensional. Concretamente será un ASV el tipo de barco oceanográfico que podrá realizar misiones.

ROV Pueden aprovechar partes de la misión para simplificar el control del operario — v. g. recopilación y almacenamiento de muestras, comunicación. Normalmente la navegación y guiado no se aprovechará, pues por definición son tareas realizadas de forma teleoperada.

Definición 10.1 (Misión). *Definiremos una misión como el conjunto de tareas que debe realizar un robot o vehículo.*

[1] Será indiferente si la boya es anclada o está a la deriva.

En el Capítulo 11 se realiza el análisis y estudio de los tipos de misiones que puede realizar un AUV —u otros vehículos similares. A partir de la tipología de misiones del Capítulo 12 se realiza el estudio de diversas arquitecturas de especificación de misiones para terminar proponiendo una basada en planes en la Sección 12.2, cuya sintaxis —o especificación formal— se muestra en el Capítulo 13.

Puede identificarse un ciclo de vida de la misión, el cual se comenta en la Sección 12.3 y consta de varias fases. En ellas se observan y analizan las dependencias con el equipamiento y el sistema embebido en el vehículo que realizará la misión (véase la Parte I y Parte III, respectivamente).

Capítulo 11

Tipología

AUVs applications concern:
ocean study, ocean intervention missions (offshore,
sea exploitation, surveys), military applications
(mine warfare and mine countermeasures, surveillance,
intelligent data collection and tactical oceanography).

— Claude Barrouil y Jérôme Lemaire
1998 (Directores de los proyectos
DCSD y GESMA)

Para el estudio de los diferentes tipos de misión realizables por un vehículo de las características de un AUV vamos a enumerar una serie de casos categorizados por temática. Cada tipo de misión se explica e ilustra con una misión básica, formada por la información mínima e imprescindible del ámbito de especificación, a la que pueden incorporarse diversas **restricciones temporales** o de **toma de muestras**. La sintaxis empleada en los ejemplos de tipos de misiones, aunque es muy simple y con fines meramente divulgativos, se explica en la Sección E.12.

Definición 11.1 (Tipología de misiones). *Estudio y clasificación de los tipos de misión definidos por una serie de tareas específicas del ámbito de aplicación concreto del robot o vehículo que realizará las misiones.*

Definición 11.2 (Toma de muestras). *Componente de la misión donde se especifican las medidas de las que deben tomarse muestras con una determinada configuración —v. g. frecuencia, resolución, etc.*

Definición 11.3 (Restricciones temporales). *Componente de la misión donde se indican las condiciones relativas al tiempo de realización de la misión o de sus partes.*

El estudio consiste en la elaboración de una taxonomía de misiones que definen tareas estrechamente relacionadas entre sí y de un ámbito bien diferenciado, que a la postre serán

combinables entre sí (véase la Sección 11.7). Los tipos de misiones que se estudian en las siguientes secciones muestran simplemente casos básicos centrados en el ámbito cubierto por el tipo de misión en cuestión y no deben interpretarse como el alcance máximo de la especificación de la misión. Mediante la combinación de otros elementos se podrán definir misiones mucho más potentes que las mostradas en los ejemplos incluidos en este capítulo. Se remite al lector al Capítulo 13 para la consulta de una especificación de misiones formal y completa, donde se documenta la semántica de los atributos empleados en los diferentes tipos de misiones tratados en este capítulo; no obstante, esta especificación está basada en la arquitectura propuesta tras el análisis de la tipología de misiones y el estudio de arquitecturas de especificación de misiones (véase la Sección 12.2 para conocer los detalles de la arquitectura propuesta).

Durante el desarrollo de la misión se considerará el consumo y la carga de las baterías disponible para evitar su agotamiento durante el desarrollo de la misión, evitando que ésta quede comprometida; esto es especialmente crítico en el caso de las misiones de navegación, ya que el estado del entorno afecta significativamente. Será el sistema del vehículo que ejecute la misión el que tendrá esto en consideración (véase la Parte III).

11.1. Boya. Toma de muestras

En el caso de que el vehículo que ejecuta la misión deba mantenerse en una posición fija, se comportaría como una boya oceanográfica, tomando muestras en dicha posición. No obstante, la toma de muestras es igualmente aplicable cuando el vehículo se mueve. El AUV mantendrá la posición que se indique en la especificación de la misión —similar a la del Algoritmo 11.1— bien con un anclaje, o bien a base de rectificaciones posicionales usando los motores[1]. También cabe la posibilidad de que el AUV se deje a la deriva, lo cual también queda contemplado en la especificación de este tipo de misiones (véase la Sección 13.6 para más detalles) —i. e. no se indicaría ninguna posición concreta.

Tal y como muestran el Algoritmo 11.2 y Algoritmo 11.3, la tipología de misión de boya puede ir acompañada de restricciones temporales y toma de muestras, respectivamente —esto será lo habitual, si no el AUV estaría ocioso, en principio.

```
1 // Posicion a mantener
2 Posicion.latitud = 10 N
3 Posicion.longitud = 3 E
4 Posicion.profundidad = 0 metros
```

Algoritmo 11.1: Boya básica

```
1 // Posicion a mantener
2 Posicion.latitud = 10 N
3 Posicion.longitud = 3 E
4 Posicion.profundidad = 0 metros
5
6 // Restricciones globales, variables de la mision
7 TiempoTotal.tiempo = 1 hora
8 TiempoTotal.holgura = 10%
```

Algoritmo 11.2: Boya con restricciones temporales

```
1 // Posicion a mantener
2 Posicion.latitud = 10 N
```

[1]Cuando el AUV está en superficie —como es el caso con una tipología de misión de boya— puede conocer su posición usando un GPS, lo que permite mantener una posición preestablecida usando un algoritmo de control más o menos simple.

```
3  Posicion.longitud = 3 E
4  Posicion.profundidad = 0 metros
5
6  // Lista de restricciones de muestreo
7  Muestreos[1].medida = temperatura
8  Muestreos[1].muestras = Infinitas // Aunque se podra indicar el numero de
9                                    // muestras
10 Muestreos[1].frecuencia = 1 Hz
11
12 Muestreos[2].medida = salinidad
13 Muestreos[2].muestras = Infinitas // Aunque se podra indicar el numero de
14                                   // muestras
15 Muestreos[2].frecuencia = 1 Hz
```

Algoritmo 11.3: Boya con toma de muestras

11.1.1. Elección del sensor

En la especificación de misiones de toma de muestras o de tipo boya surge la posibilidad de elegir el sensor con el que muestrear una determinada medida. En la Parte I se realiza un estudio de todo el equipamiento, donde se incluye el sensorial y explica la especificación del equipamiento (véase el Sección 8.1 y Capítulo 9). A la hora de especificar la misión se favorecerá su independencia del equipamiento. En este sentido y en el caso concreto de la especificación de la toma de muestras, esto se materializa en el hecho de no ser necesario indicar con qué sensor concreto realizar el muestreo de una determinada medida.

Una especificación de toma de muestras donde sólo se haga referencia a la medida que se desea muestrear, es una especificación intercambiable entre vehículos con equipamiento diferente —v. g. dos vehículos que midan la temperatura del mar con sensores de temperatura diferentes podrán usar la misma misión. Esta independencia entre equipamiento y misión es el fundamento básico para el reaprovechamiento de misiones o de sus componentes.

Sin embargo, en determinados casos puede interesar indicar el sensor concreto que se desea usar para muestrear una determinada medida; opcionalmente también podrá indicarse la configuración con la que éste debe operar. Por este motivo, la elección del sensor queda contemplada en la especificación de este tipo de misión de forma opcional, recogiendo así ambas alternativas de especificación. Además, en el caso de que existan varios sensores que puedan muestrear una misma medida, se permite que en la especificación se indique el conjunto o grupo de sensores que pueden usarse para tomar muestras de la misma; e.g. en el Algoritmo 11.4 se especifica el sensor **SBE 37-SIP** para muestrear la **temperatura** usando la configuración indicada en el fichero **modoSync.cfg**. Cuando no se indica un único sensor, el proceso de elección es realizado de forma transparente y no es necesario indicar ningún dato al respecto en el **planificador** (véase la Apéndice A); en la Sección 15.7 se detalla la gestión que realiza el sistema embebido en el vehículo cuando la elección del sensor no es unívoca[2]. En resumen, la especificación admitirá las siguientes alternativas:

1. Indicar qué medida quiere tomarse y restricciones sobre el muestreo de la misma —v. g. frecuencia de muestreo y resolución, fundamentalmente. El sistema seleccionará el sensor de forma transparente.

[2]La elección del sensor es unívoca cuando se indica un y sólo un sensor con el que muestrear una determinada medida.

2. Indicar qué sensor usar para tomar una determinada medida.

3. Indicar un conjunto de sensores para tomar una determinada medida. El sistema seleccionará el sensor de forma transparente, dentro del conjunto definido.

```
1  // Posicion a mantener
2  Posicion.latitud = 10 N
3  Posicion.longitud = 3 E
4  Posicion.profundidad = 0 metros
5
6  // Lista de restricciones de muestreo
7  Muestreos[1].medida = temperatura
8  Muestreos[1].muestras = 2000
9  Muestreos[1].frecuencia = 1/3600 Hz // Periodo = 1 hora
10 Muestreos[1].sensor[1] = SBE 37-SIP
11 Muestreos[1].sensor[1].configuracion = modoSync.cfg
```

Algoritmo 11.4: Toma de muestras especificando sensor

Con el modelo de elección del sensor planteado pueden aparecer incompatibilidades entre la misión y el equipamiento del vehículo que la realizará. Este tipo de incompatibilidades se denominan anticipables y pueden detectarse fácilmente mediante un proceso de validación de la misión (véase la Sección 12.3).

11.2. Seguimiento de Rutas

El seguimiento de una o varias rutas será uno de los principales tipos de misión que puede realizar cualquier vehículo en general y en especial un AUV. Si el AUV sólo navega por la superficie, actuando como un barco, la tipología de misiones de navegación (léase seguimiento de rutas, exploración de áreas y seguimiento de medidas) sigue siendo igualmente útil, pues el soporte para la navegación tridimensional incluye la bidimensional y unidimensional. El Algoritmo 11.5 muestra la especificación más básica del seguimiento de una ruta, donde simplemente se declara la ruta y se ordena su seguimiento.[3]

Definición 11.4 (*Waypoint*). *Punto de paso. Por lo general pertenecen al recorrido de una ruta.*

Definición 11.5 (Transecto). *Un transecto o transepto*[4] *consiste en una línea —real o imaginaria— que cruza una zona y une dos puntos, que en el caso de las rutas se denominan* waypoints.
En el ámbito de la biología hace especial alusión al recorrido lineal sobre una parcela o terreno, sobre el cual se realiza algún tipo de muestreo (véase la Figura 11.1).
Aunque ni el concepto ***transecto*** *ni* ***transepto*** *están recogidos por la RAE, se suelen usar en disciplinas técnicas.*

Definición 11.6 (Ruta). *Especificación del camino que debe seguir el vehículo que realiza la misión. Una ruta se define mediante una secuencia de* waypoints *básicamente*

[3] La definición de una ruta se analiza en la Sección 13.6 —i. e. especificación de los *waypoints* que forma la ruta, información adicional como velocidades entre *waypoints*, etc.

[4] Dentro de la terminología arquitectónico religiosa, **transepto** tiene un significado diferente del que se usa en esta memoria. Etimológicamente proviene de las voces latinas *trans* y *septum*, que significa seto o barrera, haciendo alusión al hecho de que se está interpuesto. En este sentido, también puede considerarse que tiene un significado equivalente a **transecto**.

Figura 11.1: Ejemplos de realización de transectos en biología

(véase la Definición 11.4). Además, incluirá varias condiciones adicionales relativas a la forma de recorrer la ruta y alcanzar los waypoints*; v. g. en la Sección 13.6 se muestra cómo definir condiciones relativas a los transectos (véase la Definición 11.5).*

Se pueden plantear casos más complejos añadiendo restricciones temporales o la indicación de toma de muestras a lo largo de la ruta, como se observa en el Algoritmo 11.6 y Algoritmo 11.7, respectivamente. También se pueden indicar datos de soporte para la navegación como la información batimétrica. Este tipo de datos sólo serán referenciados desde la misión, de modo que la representación de los mismos no es relevante a efectos de especificación de la misión y será específica de cada caso.

```
1 // Datos de la mision
2 Ruta ruta1
3
4 // Datos de soporte (opcional)
5 Batimetria batimetria
6
7 // Comandos
8 Seguir(ruta1)
```

Algoritmo 11.5: Seguimiento básico de ruta

```
1 // Datos de la mision
2 Ruta ruta1
3
4 // Comandos
5 Seguir(ruta1)
6
7 // Restricciones globales, variables de la mision
8 TiempoTotal.tiempo = 1 hora
9 TiempoTotal.holgura = 10 %
```

```
10
11 // Lista de restricciones locales, entre waypoints:
12 // a) Los waypoints deben ser crecientes, evitando solapes
13 // b) No se permitira indicar una restriccion del waypoint 1 al ultimo,
14 //    pues eso se indicaria con TiempoTotal
15 RangoTiempo[1].waypoints = [1 4]  // waypoint inicial y final del rango
16 RangoTiempo[1].tiempo = 10 minutos
17 RangoTiempo[1].holgura = 1%
```

Algoritmo 11.6: Seguimiento de ruta con restricciones temporales

```
1  // Datos de la mision
2  Ruta ruta1
3
4  // Comandos
5  Seguir(ruta1)
6
7  // Lista de restricciones de muestreo, entre waypoints
8  Muestreos[1].medida = temperatura
9  Muestreos[1].waypoints = [10 15]
10 Muestreos[1].muestras = 1000
11 Muestreos[1].frecuencia = 1 Hz
12
13 Muestreos[2].medida = profundidad
14 Muestreos[2].waypoints = [10 15]
15 Muestreos[2].muestras = 1000
16 Muestreos[2].frecuencia = 1 Hz
```

Algoritmo 11.7: Seguimiento de ruta con toma de muestras

En la especificación del seguimiento de rutas hay algunos atributos que se restringen mutuamente —i. e. definen un mismo aspecto de formas diferentes— siendo necesario controlar que no se produzcan casos inconsistentes o incoherentes —v. g. las muestras y frecuencia definen indirectamente el tiempo que se tardará en seguir la ruta— (véase la Sección 12.3.1 para conocer más sobre lo que se denominan incompatibilidades de la misión).

Este tipo de problemas se solucionarán controlándolos al definir la misión. Para ello se dispone de la interfaz de la aplicación de planificación de misiones (véase la Apéndice A), que controla la introducción de datos —v. g. en el ejemplo anterior sólo será necesario introducir la frecuencia y al proporcionarse el tiempo durante el que debe muestrearse se hallará el número de muestras según la frecuencia— para definir la misión y permite su validación en base a la propia misión y el equipamiento del AUV (véase la Parte I). Con esta aplicación todo el proceso será transparente al usuario, evitando la introducción de datos redundantes y, por tanto, la posibilidad de incoherencias. La Figura 11.2 muestra un ejemplo de especificación del seguimiento de ruta de forma gráfica [Issac et al., 2005]. Además, a lo largo de la ruta se suelen añadir otras especificaciones, como tareas a realizar, tal y como se observa en la Figura 11.9 —i. e. se tendría una misión combinada.

Cuando se haya generado y validado una misión, no sólo se enviará ésta al AUV, sino también la especificación del equipamiento, que puede estar precargado. Esto es necesario porque el propio sistema debe conocer su equipamiento hardware para la ejecución de las misiones, como se muestra en el Ejemplo 11.1. Esto facilita la versatilidad del sistema, dada la posibilidad de autoconfiguración —v. g. si un sensor se estropea, basta con elegir otro que proporcione la medida necesaria, si lo hubiere (véase la Sección 11.1.1 para más información sobre la gestión interna de los sensores para la toma de muestra de una determinada medida).

Ejemplo 11.1 (Utilidad del equipamiento). *Sea una misión en la que se especifique el muestreo de* ***temperatura***, ***salinidad*** *y* ***presión***, *el equipamiento permitirá al sistema saber de qué sensores dispone para la toma de dichas medidas, de acuerdo con*

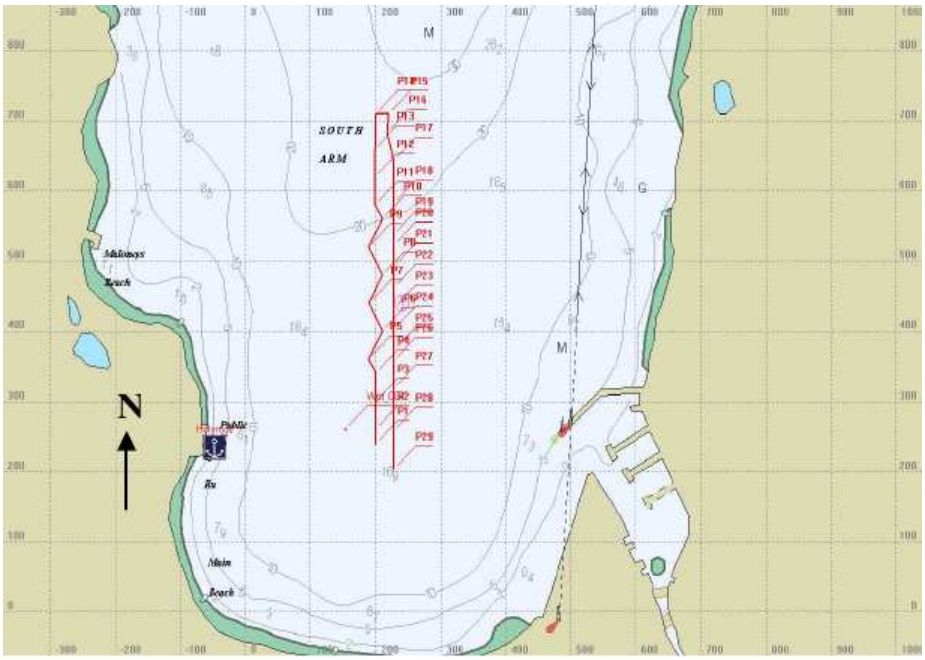


Figura 11.2: Ejemplo de especificación gráfica de seguimiento de ruta —compuesta de *waypoints*— con el software de [Issac et al., 2005]

las restricciones de frecuencia y resolución indicadas en la misión (véase la Sección 13.5 para comprender mejor este ejemplo). En este caso se comprobaría la presencia de un ***termómetro***, ***sensor de conductividad*** *—que permite el cómputo matemático de la salinidad— y* ***profundímetro***, *respectivamente, o bien la de un* ***CTD*** *(Conductividad Temperatura Presión), que permite el muestreo de las tres medidas mencionadas.*

11.3. Exploración de Áreas

A partir de un área definida por el usuario en la interfaz del planificador de misiones se creará una misión en la que además se indicará cómo explorarla. El usuario puede definir el área como un polígono de n vértices, indicando a continuación cómo recorrerlo en base a los siguientes aspectos (véase el Algoritmo 11.8 como ejemplo básico de exploración de áreas):

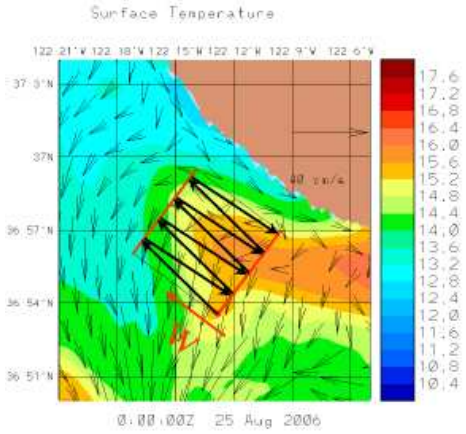
1. Modo de recorrido o exploración (en [Issac et al., 2005, Wang, 2007] pueden verse algunos ejemplos y estudios, tanto desde la especificación como el control en la ejecución de las maniobras de recorrido), que puede ser:

   a) Espiral, hacia dentro o hacia fuera.

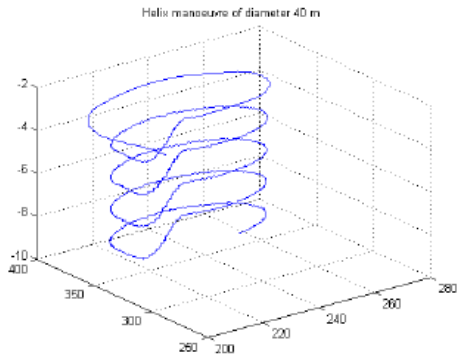
   b) Transversal —i. e. en *zigzag*.

   c) Otros (véase la Sección 13.6 para una lista más completa y detallada).

   En la Figura 11.3 se muestran algunos ejemplos gráficos de especificación del tipo



(a) Recorrido en *zigzag*. Especificación gráfica sobre el mapa, como parte de la misión



(b) Recorrido en espiral. Maniobra helicoidal (*helical manoeuvre*)

Figura 11.3: Ejemplo de especificación gráfica de exploración de área. Diferentes recorridos especificables

de recorrido para la exploración de un área: (a) en *zigzag*[5] y (b) en espiral o forma helicoidal [Issac et al., 2005].

2. Frecuencia del barrido al explorar el área (véase la Definición 11.8); este aspecto se tratará de forma específica según el modo de recorrido seleccionado.

3. Resolución, i. e. detalle con el que explorar el área.

4. Profundidad a la que explorar el área.

5. Otros (véase la Sección 13.6 para una lista más completa y detallada).

Definición 11.7 (Área). *Especificación de la zona que el vehículo explorará describiendo un recorrido concreto. El área se puede definir como una superficie poligonal —bidimensional— a la que se añadirían profundidades, lo cual constituye un espacio volumétrico de exploración.*

Definición 11.8 (Barrido). *Proceso de exploración sistemática de un espacio para recorrerlo u obtener información del mismo.*
El barrido hace referencia a la forma en que se recorre el espacio o área a explorar, ya que indica un patrón repetitivo u oscilatorio como el que se observa en la Figura 11.3 (a).

Cuando el **planificador** construye la misión a partir de las especificaciones del usuario, existen dos posibles representaciones de la misión de navegación resultante:

1. Especificar el área indicando sus características, i. e. el modo en que se explorará. No se convierte el área a una ruta equivalente, por lo que el vehículo debe planificar la exploración del área en tiempo de ejecución.

[5] El recorrido en *zigzag* también puede tener variantes según el tipo de giro entre cada transecto, v. g. *mowing* o *yoyo* [Wang, 2007].

2. Convertir el área en una ruta equivalente definida por *waypoints*, tramos o transectos internos al área. La ruta equivalente se genera en base al área definida y cómo debe explorarse.

Ambas posibilidades están contempladas, tanto en *Sick***AUV**, como en el planificador de misiones (véase la Apéndice A). Se puede hacer que todas estas decisiones sean transparentes al usuario o bien permitir que éste indique si desea que las áreas se conviertan a rutas.

Resulta más interesante no convertir las áreas a rutas equivalentes. Esto añade complejidad al sistema del AUV, pero permite una mayor potencialidad, tal y como se ilustra en el Ejemplo 11.2. Así, durante la ejecución de la misión se pueden determinar o refinar aspectos concretos relativos a la exploración del área —i. e. reconfigurarla dinámicamente.

Ejemplo 11.2 (Configuración dinámica de exploración de áreas). *Según el flujo de las corrientes marinas —i. e. el vector (dirección y amplitud) de desplazamiento de las aguas— se puede reducir el consumo energético aprovechándolas para explorar el área trazando transectos que sigan dichas corrientes. Igualmente, en base al tiempo y la frecuencia de exploración del área se podrá determinar el número de muestras a tomar durante la misma; también se considerará el consumo energético que ello implique.*

Tal y como muestran el Algoritmo 11.9 y Algoritmo 11.10, la exploración de áreas puede ir acompañada de restricciones temporales y toma de muestras, respectivamente —como en el caso de seguimiento de rutas (véase la Sección 11.2). La definición del área se puede consultar en la Sección 13.6.2.2, así como la especificación de la forma de recorrerla, indicando el modo, resolución, etc.

```
1  // Datos de la mision
2  Area area
3
4  // Comandos
5  Recorrer(area, modo, resolucion)  // Recorre el area en un modo (espiral,
6                                    // zig-zag, ...), con una resolucion (o
7                                    // frecuencia)
8
9  // Restricciones, como la profundidad
10 Profundidad.limites = [0 100] metros  // Limite minimo y maximo de profundidad
11 Profundidad.incremento = 20 metros    // Incremento para volver a recorrer el
12                                       // area a la nueva profundidad
```

Algoritmo 11.8: Exploración básica de área

```
1  // Datos de la mision
2  Area area
3
4  // Comandos
5  Recorrer(area, modo, resolucion, profundidad)
6
7  // Restricciones globales, variables de la mision
8  TiempoTotal.tiempo = 1 hora
9  TiempoTotal.holgura = 10%
```

Algoritmo 11.9: Exploración de área con restricciones temporales

```
1  // Datos de la mision
2  Area area
3
4  // Comandos
5  Recorrer(area, modo, resolucion, profundidad)
6
7  // Lista de restricciones de muestreo
```

```
8  Muestreos[1].medida = temperatura
9  Muestreos[1].muestras = Infinitas // Todas las que sean en el area
10 Muestreos[1].frecuencia = 1 Hz
11
12 Muestreos[2].medida = profundidad
13 Muestreos[2].muestras = Infinitas // Todas las que sean en el area
14 Muestreos[2].frecuencia = 1 Hz
```

Algoritmo 11.10: Exploración de área con toma de muestras

En entornos multi-AUV la exploración de un área puede coordinarse, mejorando así la eficiencia en el proceso.

11.4. Seguimiento de Medidas. Gradiente, Rango, etc.

Puede ser de interés que la ruta seguida por el AUV esté determinada dinámicamente durante el transcurso de la misión en base a una función aplicada a una medida muestreada por el sistema. Ésta es la finalidad de una misión de seguimiento de medidas.

La función que se aplique a la medida para determinar la ruta a seguir puede ser muy variada. No obstante, ateniéndonos al ámbito de vehículos de exploración submarina podemos enumerar las siguientes como las principales:

Gradiente Sigue el gradiente estimado de una medida, es decir, la dirección de máxima variación de la misma. En el Algoritmo 11.11 se muestra cómo se especificaría en una misión. Este tipo de función es especialmente útil para casos como el del Ejemplo 11.3.

Rango Se sigue una medida dentro de un rango. Si se sale del rango se rectifica la posición para volver a donde el valor esté dentro del rango (véase el Algoritmo 11.12).

Otras funciones Según el objeto de estudio se tendrán diferentes funciones de utilidad en el seguimiento de una medida. Estas funciones tomarán una o varias medidas como entrada y como salida deben proporcionar simplemente un vector de navegación, i. e. un vector velocidad tridimensional que determina cómo debe moverse el vehículo.

Ejemplo 11.3 (Seguimiento de un vertido). *El seguimiento del gradiente de una medida puede permitir la localización del emisario de un vertido de hidrocarburos. Para ello bastaría con un sensor adecuado y la definición de una misión para el seguimiento del gradiente de la medida de concentración de hidrocarburos en el agua. El vehículo navegará siguiendo una estrategia de estimación del gradiente de la concentración, de modo que se irá acercando paulatinamente al origen del vertido, i. e. al emisario [Ridao, 2006].*

Las misiones de seguimiento de medidas se caracterizan por constar de varias fases para su correcta realización. A continuación se muestran dichas fases, donde el paso de una fase a otra viene determinados por una serie de condiciones. En la Figura 11.4 se muestra el autómata que rige este flujo de control, donde cada estado representa una fase y las transiciones muestran las condiciones o eventos que se deben producir para el cambio de fase. Se observa como la fase de **búsqueda** es la inicial y la de **finalización** es la final, lo que indica que se ha terminado el seguimiento de la medida.

Búsqueda Cuando no se han alcanzado los valores de la medida que satisfacen el seguimiento especificado, el vehículo realizará una búsqueda mediante predicciones a

partir de las muestras de la medida. Esta será la fase inicial y también se pasará a ella desde la fase de mantenimiento si se pierde la medida (transición de **perdida**).

Mantenimiento Cuando se hayan detectado los valores de la medida que satisfacen el seguimiento especificado (transición **encontrada**), el vehículo intentará mantenerlos, i. e. seguirá la medida de acuerdo con la técnica especificada.

Finalización Bajo determinadas condiciones el vehículo dejará de seguir la medida. Esto puede ocurrir tanto desde la fase de búsqueda como desde la de mantenimiento (transición de **parada**).

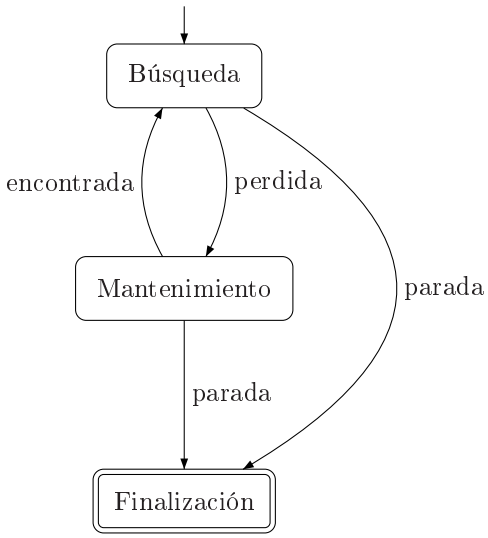


Figura 11.4: Autómata de las fases que se realizan en el seguimiento de medidas

El seguimiento de una medida quedará completamente definido con un área que determina los límites para seguir la medida, al igual que un tiempo. Esto suele ser necesario como condición de parada. Tal y como muestran el Algoritmo 11.13 y Algoritmo 11.14, el seguimiento de medidas puede ir acompañado de **restricciones temporales** y **toma de muestras**, respectivamente —como en el caso de seguimiento de rutas (véase la Sección 11.2).

```
1  // Seguimiento
2  SeguimientoGradiente.medida = temperatura
3  SeguimientoGradiente.direccion = Creciente  // Creciente o decreciente
4  SeguimientoGradiente.limites = [0 100] C    // Limites inferior y superior, para
5                                              // que no siga gradiente fuera de
6                                              // ellos
7
8  // Area limite opcional; fuera de ella no se sigue el gradiente
9  Area areaLimite                  // Area limite (poligonal)
10 Profundidad.limite = [0 100] metros // Limite minimo y maximo de profundidad
```

Algoritmo 11.11: Seguimiento de Gradiente

```
1  SeguimientoRango.medida = temperatura
2  SeguimientoRango.rango = [10 30] C // Rango minimo y maximo de temperatura
3
4  // Area limite opcional; fuera de ella no se sigue el gradiente
5  Area areaLimite                  // Area limite (poligonal)
6  Profundidad.limite = [0 100] metros // Limite minimo y maximo de profundidad
```

Algoritmo 11.12: Seguimiento de Rango

```
// ... (GRADIENTE O RANGO) ...

// Restricciones globales, variables de la mision
TiempoTotal.tiempo = 1 hora
TiempoTotal.holgura = 10%
```

Algoritmo 11.13: Seguimiento con restricciones temporales

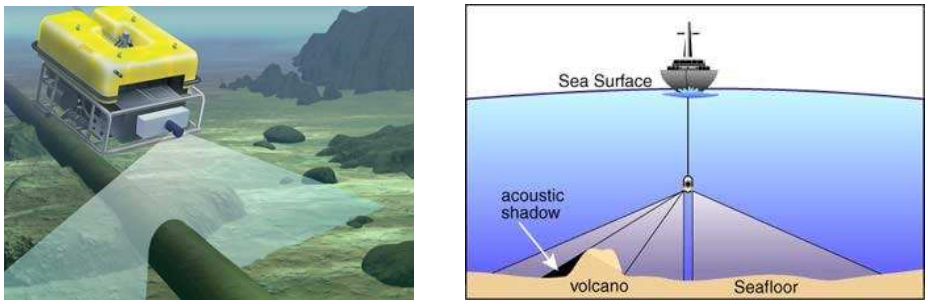
```
// ... (GRADIENTE O RANGO) ...

// Lista de restricciones de muestreo
Muestreos[1].medida = temperatura
Muestreos[1].muestras = Infinitas // Todas las que sean en el area
Muestreos[1].frecuencia = 1 Hz

Muestreos[2].medida = profundidad
Muestreos[2].muestras = Infinitas // Todas las que sean en el area
Muestreos[2].frecuencia = 1 Hz
```

Algoritmo 11.14: Seguimiento con toma de muestras

Además, existen determinadas misiones *ad hoc* que quedarían integradas dentro del seguimiento de medidas. Es el caso del seguimiento de tuberías o *pipeline tracking* (véase [Evans et al., 2003, Petillot et al., 2003]), en cuyo caso la medida sería una imagen sonar. En la Figura 11.5 (a) se ilustra el proceso de seguimiento de una tubería usando un sonar frontal o *Forward Looking Sonar* (FSL). No obstante, lo más común es el uso de un Sonar de Barrido Lateral (SSS), como el de la Figura 11.5 (b). De hecho, las imágenes mostradas en la Figura 11.6 han sido tomadas con un SSS. En la Figura 11.6 (a) se muestra un imagen RAW —i. e. la imagen original tomada con el sonar— obtenida en este tipo de misiones; la Nota 11.1 explica brevemente la imagen RAW que proporciona el SSS montada en el AUV. Esta imagen original se procesa para detectar las tuberías presentes en ellas (véase la Figura 11.6 (b)). A partir de estos resultados es posible la reconstrucción de un mapa tridimensional del fondo marino y las tuberías detectadas, tal y como muesta la Figura 11.6 (c).

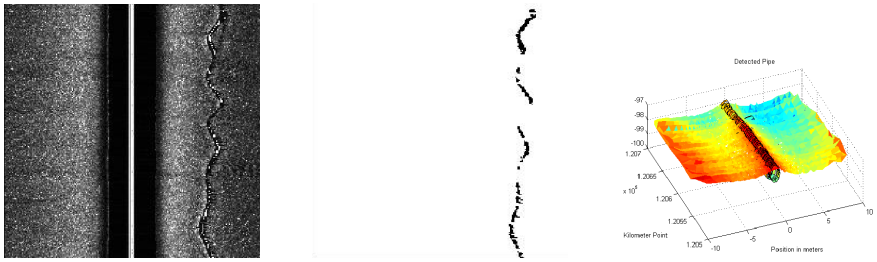


(a) Seguimiento de tubería mediante AUV con sonar frontal

(b) Sonar de Barrido Lateral. Funcionamiento

Figura 11.5: Seguimiento de tuberías. Procedimiento mediante sonar

Nota 11.1 (Imagen RAW de tubería). *La imagen RAW obtenida por un sonar, concretamente un SSS, como en el caso de la Figura 11.6 (a), es la imagen original, i. e. primaria, no procesada.*



(a) Imagen RAW. Una tubería

(b) Imagen (a) umbralizada y segmentada

(c) Reconstrucción tridimensional de tubería y fondo marino

Figura 11.6: Imágenes sonar de tuberías. Imagenes originales y procesadas para la detección y seguimiento de tuberías

Al usar un SSS se observa una franja negra vertical en el centro de la imagen. Esto es debido a la ubicación del SSS, el cual está dispuesto como se muestra en la Figura 11.5 (b). Como se observa en la figura, en la zona central no se toman datos, ya que el sonar se ubica en los laterales, debido a que en el centro se encuentra el fuselaje del vehículo submarino. En cualquier caso, en algunos casos esto puede evitarse en el diseño del vehículo.
Aunque la imagen RAW es de baja calidad se puede observar una tubería en la parte derecha, la cual queda claramente identificada en la imagen segmentada de la Figura 11.6 (b).

En entornos multi-AUV el seguimiento de medidas puede coordinarse, mejorando así la eficiencia en el proceso, pudiendo maximizar el área abarcada. Como cada AUV tiene su propio sensor para el muestreo de la medida, toda la flota de AUVs —gracias a la comunicación entre AUVs y a la coordinación— puede seguir la medida considerando las muestras obtenidas por toda la flota; según la estrategia de coordinación cada AUV podría adoptar un rol diferente en el seguimiento de la medida.

11.5. Comunicación

La tipología de misiones de comunicación tienem la finalidad de indicar al AUV cuándo enviar o recibir determinados datos —la recepción puede implicar la petición previa o simplemente la puesta a la escucha, i. e. conectarse a la red de comunicaciones que esté disponible. Desde este punto de vista, no es un tipo de misión *per se*, sino una tipología de tareas de comunicación común a la gran mayoría de misiones, lo que justifica su estudio. En cualquier caso, existen diversos casos susceptibles considerarse misiones de comunicación, e.g. si el vehículo que actúa como enlace de comunicación con otros vehículos, coordinación multi-AUV, etc.

11.5.1. Roles y Diálogos de Comunicación

La especificación de la comunicación puede ser especialmente compleja y dependerá del sistema del AUV y el resto de dispositivos que se comuniquen con él. En el proceso de comunicación participarán distintos actores o terminales de comunicación. Según sus

características se les asignará un rol (véase la Definición 11.9). Se dispone de los siguientes roles de comunicación:

Planificador Usuario que podría estar en tierra, en un buque, etc. y mediante una aplicación de monitorización y control de misiones se comunicaría con el AUV (véase la Apéndice A como ejemplo de una aplicación de estas características). Incluso podría interactuar con otros planificadores, con los que se coordinaría.

AUV AUV equipado con un módulo de comunicaciones que permite que el planificador se comunique con él y viceversa. También podrá comunicarse con otros AUV —si nos encontramos en un entorno multi-AUV—, bien para el paso de datos, bien para la coordinación.

Definición 11.9 (Rol (comunicación)). *Función que alguien o algo cumple, i. e. papel de un actor o participante. Acepción etimológicamente procedente del inglés* role*, papel de un actor, y éste del francés* rôle.
En el ámbito de las comunicaciones hace referencia al papel que desempaña un determinado participante, i. e. la forma en que actúa durante el proceso comunicativo.

Definidos los roles de comunicación, entre ellos podrán darse diferentes diálogos —i. e. tipología de misiones de comunicación entre varios participantes, de distinto o igual rol—, según los siguientes aspectos:

1. Roles que intervienen en el diálogo.
2. Tipos de datos transmitidos, atendiendo tanto al formato como al volumen.
3. Dirección del flujo de datos comunicado —v. g. unidireccional o bidireccional, *half-duplex* o *full-duplex*.
4. Características de la finalidad del proceso comunicativo, atendiendo a facilidad de uso por el usuario, diseño de la arquitectura y protocolo de comunicaciones subyacente, etc.
5. Restricciones físicas, v. g. alcance de la señal de comunicación, velocidad de transmisión (ancho de banda, que puede diferir entre los participantes del diálogo), consumo energético, etc.
6. Protocolo de comunicaciones empleado.

Definición 11.10 (Diálogo (comunicación)). *Proceso de comunicación entre un emisor y un receptor que interactúan entre sí de acuerdo a un protocolo de comunicación (véase la Definición 11.11).*

Definición 11.11 (Protocolo (comunicación)). *Conjunto de estándares que controlan la secuencia de mensajes que ocurren durante una comunicación. Se trata más concretamente del conjunto de reglas que especifican el intercambio de datos u órdenes.*

En el diagrama de la Figura 11.7 se ilustran los tipos de diálogos posibles —i. e. denominados **A**, **B**, **C** y **D**—, que se comentan a continuación:

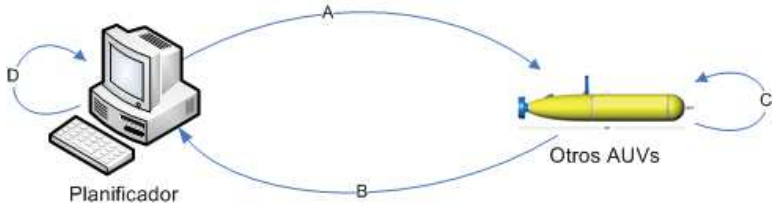


Figura 11.7: Diagrama de diálogos de comunicación entre los roles (Planificador y AUV)

1. **A. Comandar** No se trata de ninguna misión, sino de los comandos que el planificador puede enviar al AUV para controlarlo remotamente o configurar algún aspecto del sistema. También permiten el control y la modificación de la misión.
2. **B. Comunicación del AUV con el Planificador** El AUV inicia la comunicación. Esto puede ocurrir por los siguientes motivos:
   a) **Misión** El AUV se comunica de acuerdo con lo estipulado en la misión —i. e. la especificación relativa a la comunicación.
   b) **Excepciones** Al producirse algún evento dentro del sistema del AUV, se lanzará alguna rutina para gestionarlo, lo que podría desencadenar su notificación al exterior mediante los dispositivos de comunicación del AUV.
3. **C. Comunicación del AUV con otros AUVs** Dentro de un entorno multi-AUV, un AUV inicia la comunicación. Esto incluye los motivos de comunicación enumerados para el diálogo **B** (véase el Ítem 2) y la coordinación entre AUVs para la realización de ciertas tareas.
4. **D. Comunicación entre planificadores** Un planificador se comunicará con otro para coordinar varias misiones de AUVs o compartir datos. En principio, este tipo de comunicación no se tratará en este proyecto ni en los complementarios (véase la Apéndice A).

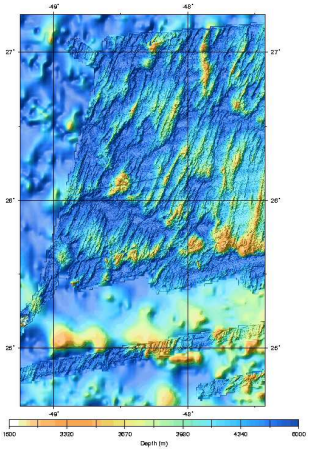
De los diálogos de comunicación identificados, en este documento vamos a centrarnos en uno muy concreto: el diálogo **B**, concretamente el caso de la **Misión**, comentado en el Ítem 2*a* —i. e. el diálogo de comunicación que forma parte de la especificación de la misión.

11.5.2. Especificación de la comunicación

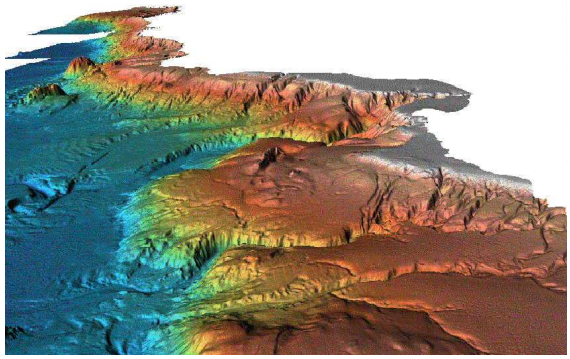
Previamente, en la Sección 11.5.1, se hizo alusión a los diferentes aspectos que determinan los diferentes diálogos de comunicación. A continuación se enumera la información básica empleada para el estudio y elaboración de la especificación de las misiones de comunicación:

Momento o ubicación Se trata de indicar en qué momento comunicarse, lo cual puede especificarse como una condición temporal (cuándo) o una localización/ubicación (dónde), a grandes rasgos.

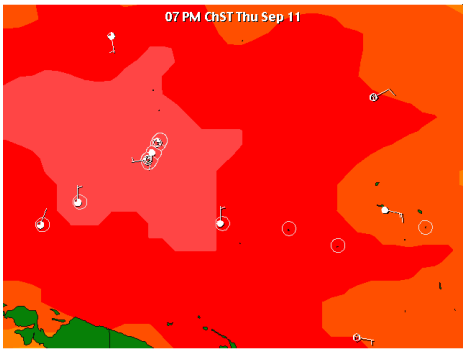
Datos La información o datos a comunicar —i. e. qué comunicar—, o bien algún tipo de aviso.



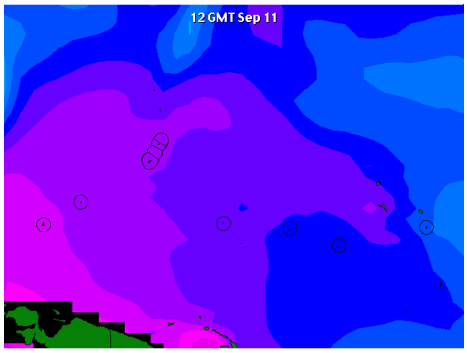
(a) Ejemplo de información batimétrica mostrada de forma bidimensional



(b) Información batimétrica mostrada de forma tridimensional. Costa de California (Dr. Lincoln Pratson, INSTAAR/NGDC, 1996). Mapa batimétrico construido con datos tomados por sonar multihaz



(c) Ejemplo de información meteorológica de tempertura del mar en la costa de Guam (Océano Pacífico)



(d) Ejemplo de información meteorológica de oleaje —altura de las olas— de la costa de Guam (Océano Pacífico)

Figura 11.8: Información relevante que puede recibir un AUV. Datos de batimetría, meteorología, etc.

El modo en que se comunicará el AUV será transparente al planificador de la misión. El sistema del AUV realizará una asociación entre la información a comunicar, según sus características —v. g. volumen, formato—, y los dispositivos que cumplan con los requerimientos mínimos necesarios —v. g. ancho de banda, compresión, portabilidad de la serialización, seguridad, etc.[6]— (véase la Sección 15.4 para más información).

Conforme a la información básica antes mencionada, se dispone de una clasificación completa de la especificación de comunicación en la misión, de acuerdo a este criterio.

1. **Momento o ubicación:** Según se especifique una u otra se tendrá una especificación temporal o espacial —i. e. cuándo o dónde comunicarse, respectivamente.

   a) **Temporal:** Se puede especificar con los siguientes datos:

[6]Este tipo de asociación es similar al que se realiza en la gestión de sensores al asociar o seleccionar un sensor para el muestreo de una medida (véase la Sección 15.7).

1) **Frecuencia:** Comunicarse cada cierto tiempo de acuerdo a una frecuencia —v. g. con un periodo de 10 minutos (1/600Hz), como se muestra en el Algoritmo 11.15—, que en la interfaz del planificador de misiones puede indicarse como un periodo y convertirse a frecuencia al crear la misión con su formato (véase la Sección 13.4).

b) **Espacial:** Se puede especificar en base a diferentes datos:

1) ***Waypoints*:** Durante el seguimiento de una ruta, como ésta está definida por *waypoints* (véase la Sección 11.2 y Sección 13.6), se pueden indicar tareas de comunicación en base a los mismos, como se muestra en el Algoritmo 11.16. Los casos posibles son:

a′ Al alcanzar o pasar por un *waypoint* w.

b′ Cada n *waypoints*.

Existe una dependencia entre la especificación del seguimiento de la ruta y las tareas de comunicación, por lo que debe chequearse que son compatibles, i. e. la comunicación es posible en los *waypoints* indicados, según el equipamiento de comunicaciones disponible.

2) **Posición:** Al alcanzar o superar una determinada posición en el espacio —tridimensional o simplificado a alguna de las coordenadas espaciales— el AUV se comunicará (véase el Algoritmo 11.17, que es una primera aproximación de la especificación de comunicación en base a la posición del AUV). Se pueden plantear distintas condiciones relativas a la posición, en función de la precisión con la que deba determinarse la posición.
Debido a que la forma de indicar la posición comentada en el borrador anterior es muy restrictiva, se ha adoptado una especificación basada en condiciones o predicados (véase el Algoritmo 11.18). Estos predicados harán referencia a variables observables del sistema del AUV —i. e. medidas proporcionadas por los sensores o internas al sistema—, que para el caso de la comunicación en base a la posición del AUV será **posicion** —del GPS, estimada[7], etc.
En el Algoritmo 11.18 se observa como la especificación dispone de una lista de condiciones, que determinan en qué posiciones comunicarse. Estas condiciones podrán usarse también para definir la comunicación relativa a medidas, que se verá en los siguientes puntos. Este tipo de especificación es más potente que el borrador original del Algoritmo 11.17, ya que es posible indicar que el AUV se comunique tanto al sumergirse a más de una profundidad, como al emerger por encima de ella. Al usar condiciones es necesario realizar la validación y estudio de viabilidad de la misión (véase la Apéndice A), para comprobar que un número excesivo de comunicaciones afecte negativamente al consumo energético.

3) **Odometría:** La comunicación se realiza tras recorrer una distancia determinada de forma periódica, como muestra el Algoritmo 11.19. Esto será especialmente útil para el chequeo de la posición en el seguimiento de rutas explicado en la Sección 11.2, ya que permite actualizar o sincronizar

[7] Es habitual que un AUV sólo pueda conocer su posición mediante estimaciones basadas en históricos de datos y modelos matemáticos de hidrodinámica, ya que cuando se encuentra sumergido no puede hacer uso de información de sistemas de geolocalización o posicionamiento en el entorno —v. g. GPS, triangulación, etc.

la posición con un sistema de posicionamiento o geolocalización —v. g. GPS.

c) **Relativa a medidas:** Se usan las medidas de los sensores u observables del sistema para determinar cuándo comunicarse. Se permiten las siguientes formas de integrar las muestras de las medidas para la especificación de la comunicación:

1) **Cantidad de muestras** Al tomar un determinado número n de muestras de una medida se producirá la comunicación. Se tratará de forma periódica (véase el Algoritmo 11.20).
2) **Valor** Cuando se alcance un cierto valor de una medida sensorial se realizará la comunicación. Realmente se evaluará una condición o predicado sobre el valor. Se especificará una lista de condiciones tal y como se muestra en el Algoritmo 11.21.
   Este esquema es idéntico al usado para indicar la comunicación en base a la posición, lo cual simplificará enormemente la especificación de la comunicación en la misión (véase el Capítulo 13, especialmente la Sección 13.4).

d) **Excepciones** En el sistema del AUV se generarán eventos o excepciones que serán tratadas por rutinas de manejo. Éstas incorporan tareas de salvaguarda o comunicación que permitirán que el AUV se ponga a salvo en caso de algún error o fallo en el sistema —v. g. subir a la superficie en caso de agotamiento de las baterías o de memoria.

La especificación de la comunicación en base a las excepciones se fundamenta en un álgebra de eventos discretos con un vocabulario que contiene los nombres de todos los eventos o excepciones que están soportados y que pueden ser disparados por los diferentes módulos o componentes del sistema del AUV (véase la Parte III). En el Algoritmo 11.22 se muestra un ejemplo básico que ilustra la simplicidad del esquema adoptado.

2. **Datos** A continuación se listan las diferentes tareas de comunicación, según los datos y el flujo de los mismos:

a) **Enviar datos** El AUV enviará datos al planificador o a otros AUVs (véase el Algoritmo 11.23, donde se especifica la lista de datos a enviar). Entre los datos que se pueden enviar se encuentran:

1) Muestras de las medidas tomadas por los sensores con los que está equipado el vehículo.
2) Valores de variables observables del sistema —v. g. estado de realización de la misión.
3) Registros (*logs*) —i. e. listas de valores— de medidas u observables del sistema del AUV.

El sistema del AUV dispondrá de una lista de medidas y datos (véase la definición de medidas y datos en la Definición 13.7 y Definición 13.8, respectivamente) que pueden usarse en las especificaciones de la misión (véase la Parte III para una explicación de la arquitectura que da soporte para esto y una lista más detallada de medidas y datos).

b) **Recibir datos** El AUV recibirá datos provenientes y proporcionados por otros actores (véase el Algoritmo 11.24) —i. e. planificador u otros AUVs. Entre los datos que pueden recibirse destacan los siguientes:

1) Información batimétrica —por lo general relativa a la posición del AUV o de acuerdo a la especificación de la misión.[8] En la Figura 11.8 (a) y (b) se muestran ejemplos de batimetría representada bidimensional y tridimensionalmente, respectivamente.
2) Información meteorológica, corrientes marinas y cualquier información útil para la realización de la misión o funcionamiento del sistema del AUV. En la Figura 11.8 (c) y (d) se muestran ejemplos de información meteorológica representada gráficamente: temperatura del mar y altura de las olas, respectivamente.
3) Comandos —v. g. el planificador de la misión puede comandar remotamente el AUV.
4) Información de coordinación —sólo en el caso de entornos multi-AUV.

El sistema recibirá el dato con el protocolo de comunicación adecuado, por lo que este aspecto será transparente al usuario en la especificación de la misión.

La recepción de datos puede ser:

1) **Pasiva** Consiste en ponerse en línea —i. e. conectarse a una red de comunicaciones o con algún dispositivo— y esperar a que se reciba el dato. Esta estrategia tiene el inconveniente de no estar sincronizada con el emisor. Además, el emisor no puede saber si el receptor está listo para la recepción —v. g. un vehículo de las características de un AUV realizando una misión típica normalmente no será capaz de ponerse en línea exactamente en el momento requerido.
2) **Activa (Petición)** Se realiza una petición del dato antes de recibirlo para que sea enviado en ese momento por el emisor. Con este protocolo se soluciona el problema identificado en la recepción pasiva —i. e. sin petición. La petición tiene una funcionalidad doble:
   a′ Notifica al emisor que se está listo para la recepción.
   b′ Indica el dato que se solicita —en el caso de las misiones esto no es necesario, pues ésta suele ser conocida también por el emisor, pero resulta especialmente útil para peticiones de datos al margen de la misión.[9]

c) **Enviar avisos** El AUV enviará avisos al planificador u otros AUVs. Se muestra un caso básico en el Algoritmo 11.25, el cual enumera una lista de avisos a enviar[10]. A continuación se enumeran algunos de los avisos más importantes:

[8] También es posible que un AUV equipado con una ecosonda o sonar multihaz (*multibeam sonar*) tome él mismo la información batimétrica directamente.

[9] En determinados casos la especificación de la comunicación en la misión debe indicarse, aparte de los datos, información adicional —v. g. resolución y posición actual para obtener una batimetría apropiada. Cuando las peticiones no son parte de la misión es evidente que las propias peticiones deben iniciar dicha información contextual. También puede ser útil en el caso de formar parte de la especificación de la misión para precisar determinados aspectos —v. g. la posición exacta.

[10] Se indicará un único aviso, según la tarea de comunicación que se esté indicando —v. g. aviso de SOS para indicar que se ha producido una excepción grave, como un fallo en un sensor o el sistema.

1) Aviso para indicar que se está operando correctamente —i. e. indicar que el sistema está funcionando.
2) Grado de realización de la misión.
3) SOS, en caso de error —v. g. si se produce una excepción crítica, como el agotamiento de las baterías.

d) **Ponerse en línea** El AUV se pondrá en superficie o alguna zona en que esté en línea[11], i. e. que otros actores se puedan comunicar con él —y él con ellos, según la direccionalidad de la comunicación.

El Algoritmo 11.26 muestra como para ponerse en línea no hay que indicar nada —el AUV estará preparado para la recepción de datos, que será lo que se pretenda, i. e. que esté en línea cada cierto tiempo o en un lugar concreto. El sistema del AUV se encargará de avisar que está en línea o disponible para recibir información —salvo que esto se detecte automáticamente. Esto permitirá a los otros actores enviar datos al AUV.

```
// Restriccion temporal
Comunicacion.frecuencia.temporal = 0.01 Hz
```

Algoritmo 11.15: Comunicación con restricción temporal

```
// Indicando waypoints
Comunicacion.waypoints = [1, 10, 15, 20, 25, 100]

// Indicando cada cuantos waypoints
Comunicacion.frecuencia.waypoints = 5 // Cada 5 waypoints
```

Algoritmo 11.16: Comunicación en base a *waypoints*

```
// Indicando posicion (3D: latitud, longitud, profundidad)
Comunicacion.alcanzar[1].posicion = [10, 20, 4]
Comunicacion.superar[1].posicion = [10, 20, 5]
Comunicacion.superar[2].posicion = [20, 40, 10]

// Indicando posicion unidimensional (ejemplo: profundidad)
Comunicacion.superar.profundidad = 100 metros
```

Algoritmo 11.17: Comunicación en base a la posición (1ª aproximación)

```
// Condiciones espaciales (se usa la posicion, cuyos campos seran:
// latitud, longitud y profundidad)
Comunicacion.condicion[1] = posicion == [10, 20, 4]
Comunicacion.condicion[2] = posicion > [10, 20, 5]
Comunicacion.condicion[3] = posicion > [20, 40, 10]
Comunicacion.condicion[4] = posicion.profundidad > 100 metros
Comunicacion.condicion[5] = posicion.profundidad < 10 metros
```

Algoritmo 11.18: Comunicación en base a la posición

```
// Restriccion de distancia recorrida
Comunicacion.distanciaRecorrida = 1000 metros
```

Algoritmo 11.19: Comunicación en base a la odometría

[11] Dependiendo de los dispositivos de comunicación las posibilidades de comunicación serán diferentes. El equipamiento indicará los dispositivos de comunicación disponibles y sus características, de modo que el sistema del AUV podrá determinar las acciones que debe realizar para ponerse en línea —conectarse— con los otros actores que intervendrán en el diálogo de comunicación.

Tipo	Dato	Descripción
Temporal	Frecuencia	Cada cierto tiempo, según una frecuencia o periodo
Espacial	*Waypoints*	Al alcanzar un *waypoint* w o cada n *waypoints*
	Posición	Al alcanzar o superar una posición
	Odometría	Tras recorrer una distancia —periódicamente
Medidas	Muestras	Cantidad de muestras tomadas
	Valor	Cuando el valor de una medida cumple un predicado
Excepciones	Excepción	Al producirse una excepción

Cuadro 11.1: Especificación del momento o ubicación de la comunicación. Tipos de especificación, datos especificados y descripción breve de cuándo o dónde se realiza la comunicación

```
// Restriccion de numero de muestras de una medida
Comunicacion.muestras[1].numeroMuestras = 100
Comunicacion.muestras[1].medida = temperatura
Comunicacion.muestras[2].numeroMuestras = 1000
Comunicacion.muestras[2].medida = profundidad
```

Algoritmo 11.20: Comunicación en base a seguimiento de n muestras

```
// Condiciones en base al valor de una medida
Comunicacion.condicion[1] = temperatura == 10 C
Comunicacion.condicion[2] = profundidad >= 150 metros
```

Algoritmo 11.21: Comunicación en base a seguimiento de un valor

```
// Excepciones para comunicarse
Comunicacion.excepcion = AgotamientoBaterias
Comunicacion.excepcion = FalloSensorTemperatura
```

Algoritmo 11.22: Comunicación por evento o excepción

```
// Envio de datos
Comunicacion.enviar.datos[1] = Registro.temperatura
Comunicacion.enviar.datos[2] = Registro.posicion
Comunicacion.enviar.datos[3] = Registro.realizacionMision
```

Algoritmo 11.23: Envío de datos

```
// Toma de datos
Comunicacion.tomar.datos[1] = posicion
Comunicacion.tomar.datos[2] = batimetria
Comunicacion.tomar.datos[3] = corrientesMarinas
```

Algoritmo 11.24: Toma o recepción de datos

```
// Avisos
Comunicacion.aviso = OK
Comunicacion.aviso = nivelRealizacionMision
Comunicacion.aviso = SOS
```

Algoritmo 11.25: Envío de avisos

```
ponerseEnLinea()
```

Algoritmo 11.26: Ponerse en línea

Tarea	Información	Descripción
Enviar datos	Muestras	Muestras de medidas sensoriales
	Valores	Valores de variables observables del sistema
	Registros	Volúmenes de datos (muestras o valores) registrados
Recibir datos	Batimetría	Información batimétrica del entorno del AUV
	Meteorología	Información meteorológica, corrientes marinas, etc.
	Comandos	Comandos de control remoto
	Coordinación	Información de coordinación
Enviar avisos	Aviso	Información corta de aviso, como SOS —en caso de error— o grado de realización de la misión
Ponerse en línea		El sistema simplemente se pone en línea; puede notificar de ello.

Cuadro 11.2: Especificación de las tareas de comunicación de un AUV. Tareas soportadas, datos o información comunicada y descripción breve de la tarea o la información involucrada

En el Cuadro 11.1 y Cuadro 11.2 se muestra un resumen de la clasificación de la especificación de comunicación de la misión realizada previamente. En el Cuadro 11.1 se muestran los diferentes tipos de especificación del momento o ubicación en que comunicarse, acompañando a cada uno de los diferentes datos que pueden usarse en la especificación y una breve descripción. En el Cuadro 11.2, para cada tarea de comunicación identificada se indican los diferentes tipos de datos o información que pueden comunicarse, describiendo brevemente qué se hace o comunica. Todas las especificaciones de comunicación mostradas pueden agruparse en una lista como la del Algoritmo 11.27 —i. e. se define una lista de tareas de comunicación. En la Sección 13.4 se explica la especificación formal de la tipología de misiones de comunicación y se comenta la detección de posibles incompatibilidades con el resto de la misión. En el caso de que la incompatibilidad no sea anticipable (véase la Sección 12.3.1), si se produce durante la ejecución de la misión, será el sistema el que determine las acciones a tomar en base a su configuración (véase la Sección 13.7 y Sección 15.8).

```
1  // Tarea 1
2  Comunicacion[1].frecuencia.temporal = 0.01 Hz
3  Comunicacion[1].aviso = nivelRealizacionMision
4
5  // Tarea 2
6  Comunicacion[2].frecuencia.waypoints = 5  // Cada 5 waypoints
7  Comunicacion[2].tomar.datos[1] = posicion
8  Comunicacion[2].tomar.datos[2] = batimetria
9
10 // Tarea 3
11 Comunicacion[3].muestras[1].numeroMuestras = 100
12 Comunicacion[3].muestras[1].medida = temperatura
13 Comunicacion[3].condicion[1] = temperatura == 10 C
14 Comunicacion[3].enviar.datos[1] = Registro.temperatura
15 Comunicacion[3].enviar.datos[2] = Registro.posicion
16
17 // Tarea 4
18 Comunicacion[4].excepcion = AgotamientoBaterias
19 Comunicacion[4].aviso = SOS
```

Algoritmo 11.27: Tareas de comunicación combinadas

11.6. Chequeo del Sistema

Para realizar el **autodiagnóstico** del sistema se dispondrá de una serie de tareas de chequeo. El chequeo del sistema no será una misión, sino una funcionalidad básica del sistema del AUV.

Se incluye el estudio del chequeo del sistema dentro de la tipología de misiones porque forma parte de las posibles especificaciones de tareas realizables por el AUV. No se considera una misión porque será un elemento más básico y de diagnóstico del sistema —i. e. no forma parte del estado de operación normal del AUV.

Los siguientes tipos de chequeo son un breve ejemplo de las funciones de chequeo que se estiman necesarias:

1. El Algoritmo 11.28 considera la posibilidad de un chequeo completo del sistema cuyas características se definirían internamente en el sistema del AUV.

2. El Algoritmo 11.29 muestra una especificación de chequeo de *grano* más fino, en la que se indicará el subsistema a chequear —el sistema del AUV tendrá una arquitectura formada por varios subsistemas (véase el Capítulo 14).

3. El Algoritmo 11.30 muestra una especificación de *grano* aún más fino que en el Algoritmo 11.29, llegando al nivel de componentes que se deben chequear —cada subsistema estará subdividido en varios componentes (véase el Capítulo 14).

```
1 // Chequear todo
2 ChequeoTotal()
```

Algoritmo 11.28: Chequeo Completo

```
1 // Chequeo de sistema
2 ChequeoSubsistema.sensorial()
3 ChequeoSubsistema.impulsion()
4 ChequeoSubsistema.procesamiento()
```

Algoritmo 11.29: Chequeo de Subsistemas

```
1 // Chequeo de componentes
2 componentes = [termometroAcme CTDAcme]
3 ChequeoComponente(componentes)
```

Algoritmo 11.30: Chequeo de Componentes (por extensión)

Los elementos chequeables dependen del equipamiento y de los componentes del sistema embebido en el vehículo que ejecuta la misión (véase la Parte I y Parte III, respectivamente). El chequeo también podría incluir la **calibración** de algunos de estos elementos. En cualquier caso, el soporte para el chequeo lo proporcionará el propio sistema mediante funcionalidades específicas (véase la Capítulo 15 para ver cómo se materializa esto en *Sick***AUV**).

11.7. Misión Combinada

Las especificaciones de los tipos de misión mostrados en la Sección 11.1, Sección 11.2, Sección 11.3, Sección 11.4 y Sección 11.5 pueden combinarse entre ellas para constituir una misión completa que denominaremos misión combinada. Una misión combinada estará constituida por una lista de tipos de misiones; es bastante ilustrativo el Ejemplo 11.4,

que muestra un esquema que facilitaría bastante todo el proceso logístico de realización de una misión. Opcionalmente, una misión puede proporcionarse al AUV por partes, i. e. se enviarían varias misiones para que se vayan realizando una a continuación de otra. En lo sucesivo, cuando nos refiramos a una misión estaremos haciendo alusión a una misión combinada, siendo ésta la semántica más lógica, ya que por lo general siempre se tienen misiones de estas características.

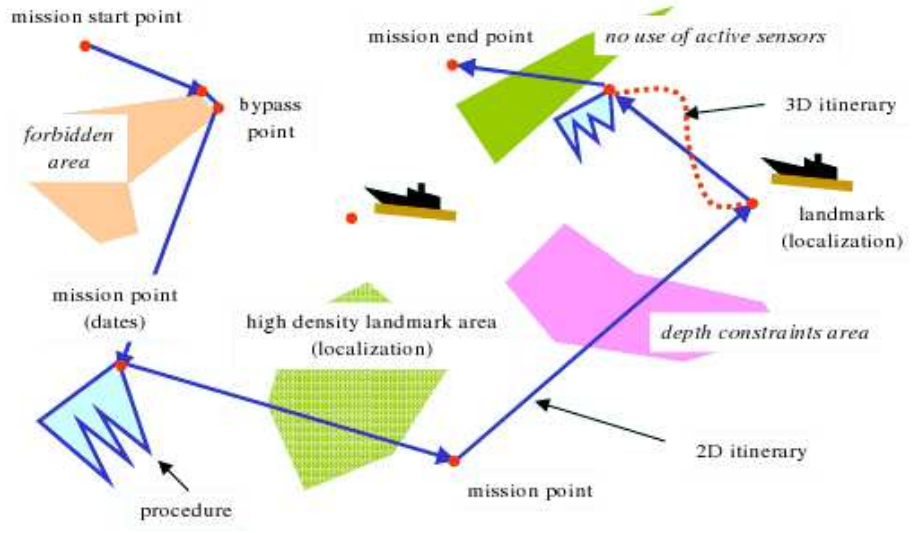


Figura 11.9: Ejemplos de especificación combinada de ProCoSa [Barrouil and Lemaire, 1998]. Seguimiento de ruta, comunicación, realización de determinadas tareas o procediminetos, exploración de áreas, etc.

Ejemplo 11.4 (Misión combinada). *Una misión típica puede estar constituida por una primera fase con una tipología de seguimiento de ruta para llegar a una posición concreta saliendo desde el muelle, una segunda de boya que constituiría la verdadera finalidad de la misión y finalmente otra de seguimiento de ruta para regresar al muelle, como muestra la Figura 11.9.*

Resulta interesante permitir que el usuario pueda especificar bucles de tareas o submisiones a realizar, agrupando secuencias para formar bloques de tareas. Esto ofrece cierto grado de abstracción y reutilización a las misiones. El planificador de misiones (véase la Apéndice A) se encargará de traducir la misión a un diagrama de bloques, donde éstos contendrían tareas de los diferentes tipos de misiones, enlazadas entre sí. Cada misión, o cada una de sus tareas, dispondrá de una lista de precondiciones a modo de chequeos, que permitirán determinar si se puede llevar a cabo la misión; en caso negativo se pasaría a la siguiente, según la lógica de gestión de este tipo de excepción —v. g. deshacer tareas/misiones previas, seguir con la siguiente tarea/misión o alguna concreta dependiendo de la excepción producida.

En el Capítulo 12 se definirá la arquitectura adoptada para la especificación de misiones de acuerdo con la tipología de misiones estudiada en el Capítulo 11. La arquitectura determina el diseño de una organización de la misión donde la especificación formal que se mostrará en el Capítulo 13 dará soporte a los aspectos comentados en los diferentes

tipos de misión y la propia misión resultante —i. e. la misión combinada. El chequeo del sistema no constituye un tipo de misión propiamente dicho (véase la Sección 11.6). Por este motivo se mantiene al margen de la misión combinada y se usa independientemente; tampoco forma parte de la arquitectura propuesta ni se recoge en la sintaxis de la especificación formal de la misión (véase el Capítulo 12 y Capítulo 13).

Capítulo 12

Arquitectura. Planes de la Misión

Reconfigurability and reliability are two keys for the success of an AUV mission...

— Richard P. Blank
1993 (Master's Thesis by DTIC)

...mission [...] consists in reaching several points where specific procedures have to be executed.

— Magali Barbier et al.
2001 (Miembros de los proyectos ONERA y GESMA)

Se dará soporte a los distintos tipos de misiones comentados en el Capítulo 11 usando una arquitectura de especificación de misiones apropiada. Para determinar qué arquitectura aplicar se realiza un estudio en la Sección 12.1 y finalmente se explica la arquitectura propuesta en la Sección 12.2, si bien el detalle de la especificación formal de las misiones conforme a la arquitectura adoptada se explica en el Capítulo 13.

12.1. Estudio de arquitecturas

El estudio de la arquitectura de especificación de misiones consiste en un análisis de arquitecturas consultadas en Internet y en la bibliografía. Se han estudiado arquitecturas para la representación de misiones de AUVs al margen de la tipología de misiones mostrada en el Capítulo 11. No obstante, a la hora de decidir la adopción de la arquitectura sí se considera el soporte completo, eficiente y apropiado para dicha tipología de misiones, como se observa en la Sección 12.2.

Las secciones siguientes resumen *grosso modo* algunas de las arquitecturas de especificación de misiones más importantes consultadas, para terminar analizando y concluyendo el estado del arte de este aspecto de diseño de las misiones (véase la Sección 12.1.4). También se determinan las bondades de las diferentes alternativas y el soporte para la tipología del Capítulo 11, antes de la explicación detallada de la arquitectura propuesta en la Sección 12.2.

Definición 12.1 (Borrador (misión)). *Especificación inicial de la misión, en base a la tipología de misiones del Capítulo 11, que constituye un primer esbozo de la misión donde se indicarán las tareas que deben realizarse en función de una serie de restricciones de diversa índole.*

Definición 12.2 (Módulo (misión)). *Un módulo es un elemento de la misión con entidad propia y una funcionalidad bien determinada.*
De forma general se define un módulo como un componente autocontrolado de un sistema, con una interfaz bien definida. En este sentido, un módulo de la misión está altamente desacoplado del resto de módulos y será fácilmente integrable con ellos; incluso es reaprovechable por otras misiones.

En el Capítulo 11 ya se mostró el tipo de misiones soportadas y la estructura básica de especificación de las mismas. A partir del análisis de la tipología de misiones se puede concluir lo indicado en la Definición 12.1, i. e. se puede plantear un borrador de la misión donde las tareas y restricciones aplicables son las vistas en el Capítulo 11. Se considera que las arquitecturas —y la definición formal— de especificación de misiones que son objeto de estudio darán soporte a una misión de estas características[1] conforme a los siguientes criterios de calidad enunciados en la Definición F.1, Definición F.2, Definición F.3, Definición F.4, Definición F.6, Definición F.7, Definición F.8 y Definición F.9 (véase el Apéndice F).

Modularidad La misión podrá estar compuesta por varios módulos idealmente independientes entre sí. Se buscan arquitecturas de especificación de misiones lo más modulares posible, pues esto favorece la reutilización, intercambio y combinación de módulos de la misión (véase la Definición 12.2). La arquitectura del sistema también podrá beneficiarse de la modularidad de la misión (véase el Capítulo 14).

Flexibilidad El mantenimiento y los cambios en la especificación de misiones deben ser de fácil realización, minimizando los "efectos secundarios". La arquitectura de la especificación de la misión debe soportar cualquier tipología de misión del Capítulo 11, así como posibles nuevos tipos de misión de AUVs, que deben poder asimilarse fácilmente por la arquitectura.

Monitorización La arquitectura de la misión y el sistema debe facilitar la monitorización, depuración y supervisión del estado actual de ejecución de la misma —v. g. posición alcanzada, nivel de realización de la misión, errores producidos, etc.

Control remoto Tanto la misión como el sistema (véase la Parte III) deben soportar la posibilidad de recibir y ejecutar comandos remotos. La arquitectura de la misión debe estar diseñada para soportar el comando y control remoto sin que ello

[1] La validación y ejecución de la misión está también apoyada en el propio sistema y en la especificación del equipamiento (véase la Parte I), que complementan a la misión.

comprometa su ejecución (véase la Sección 11.5 para más detalles sobre los tipos de comunicación soportados); la coordinación entre AUVs, en el caso de entornos multi-AUV, se sustentará en la capacidad de comando y control remoto de la misión y el sistema subyacente.

Facilidad de definición La arquitectura de especificación de misiones debe facilitar la definición de misiones. Igualmente, la especificación debe ser potente y completa, i. e. debe permitir la especificación de múltiples tareas, su configuración o parametrización, restricciones de ejecución, etc. Los diferentes tipos de misión del Capítulo 11 deben ser fáciles de definir, cubriendo todos los detalles de las mismas. La misión podrá definir comportamientos con diferentes niveles de abstracción y para distintos ámbitos —v. g. comportamientos frente a excepciones.

Portabilidad El formato de especificación de la misión debe estar soportado por distintas plataformas, sistemas o lenguajes de programación, tanto para su definición como para su interpretación durante la ejecución de la misión. Se puede definir un formato propio —v. g. DSL (Lenguaje de Dominio Específico), analizado en el Apéndice B)— o bien usar un paradigma contrastado —v. g. redes de Petri (véase el Apéndice C y Sección 12.1). Lo ideal es que la especificación de la misión disponga de herramientas multiplataforma a lo largo de todo su ciclo de vida (véase la Sección 12.3).

Reconfiguración y Aprendizaje La reconfiguración y el aprendizaje de la misión son valores añadidos que permiten la modificación de elementos de la misión dinámicamente. La misión será reconfigurable en tanto que permita ser modificada en tiempo de ejecución. Su arquitectura debe estar preparada para que determinados elementos puedan modificarse dinámicamente.[2] El aprendizaje se sustentará sobre una arquitectura de especificación de misiones reconfigurable; en este caso, será un módulo de aprendizaje integrado en el sistema (véase la Parte III), si lo hubiere, el que reconfigure la misión, en lugar de un usuario planificador de la misma (véase la Apéndice A y Sección 11.5.1).

12.1.1. Redes de Petri

Las redes de Petri (RdP en lo sucesivo) —explicadas brevemente en el Apéndice C— son una alternativa bastante común como arquitectura de especificación de misiones y del sistema de vehículos autónomos, especialmente en el caso de los AUVs.

En el caso de adoptar las RdP como arquitectura de especificación de misiones, el sistema se basará en una arquitectura soportada por el diseño con RdP y un motor de interpretación de las mismas (véase la Sección 14.1.5). Este hecho impone una dependencia entre la especificación de la misión y el sistema, pero también aporta múltiples beneficios inherentes, tanto a la misión como al sistema (véase el análisis siguiente y el estudio de la Sección 14.1.5).

Una arquitectura basada en RdP suele consistir (de acuerdo con el estudio realizado en [Barrouil and Lemaire, 1998, Ramalho Oliveira et al., 1996]) en la especificación de la misión en base a una serie de primitivas u operaciones elementales implementadas como RdP. Aprovechando la potencialidad de las RdP modulares (véase la Sección C.2.2) es

[2] El sistema estará preparado para actualizarse ante la reconfiguración de la misión.

posible definir gran variedad de tipos de misiones interconectando diferentes RdP básicas —i. e. RdP que modelan primitivas u operaciones elementales. Esta tarea suele apoyarse en entornos de programación gráficos donde se acostumbra a editar la misión con diagramas de RdP (véase la Sección C.4); v. g. CORAL (véase la Sección 14.1.5 y consúltese la fuente original [Ramalho Oliveira et al., 1996] para más información), ProCoSa (véase la Sección 14.1.5 y consúltese la fuente original [Barrouil and Lemaire, 1998] para más información), etc.

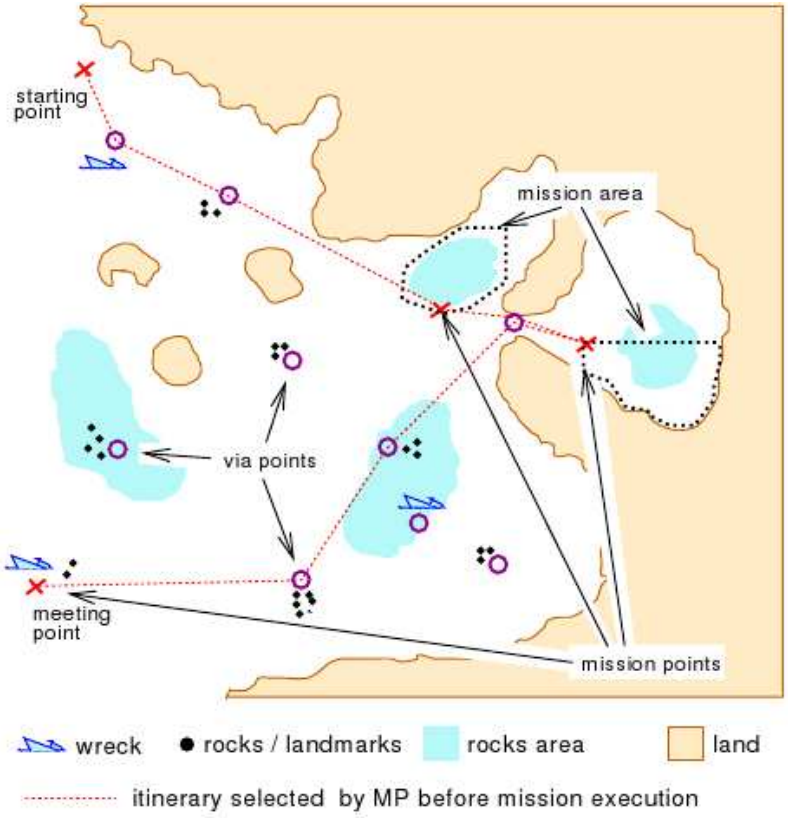


Figura 12.1: Especificación de la misión con ProCoSa. Definición de las tareas de navegación sobre el plano

La Figura 12.1 muestra la especificación de una misión centrada especialmente en las tareas de navegación —i. e. especificación de la ruta a seguir en base a *waypoints*, áreas a explorar, etc. Esta especificación gráfica sobre el plano[3] será común a cualquier arquitectura presente en la bibliografía (véase la Sección 12.1), si bien en la Figura 12.1 ha sido tomada concretamente de ProCoSa [Barrouil and Lemaire, 1998].

En el caso de una arquitectura basada en RdP, la especificación de la misión mostrada en la Figura 12.1 —además de especificaciones de otros tipos de misión— se representará o traducirá a una RdP como la mostrada en la Figura 12.2. Esta figura muestra un procedimiento de alto nivel representado gráficamente por una RdP. Este tipo de arqui-

[3]El plano o mapa puede ser una carta náutica o una representación del entorno más simple, con información batimétrica de profundidad —v. g. en la Figura 12.1 sólo se distingue entre zonas de mar y de tierra, siendo éste el tipo de plano más simple posible.

tectura modela un sistema de eventos discretos (véase la Definición C.1 y ?? para más información sobre el tipo de sistemas que modelan las RdP y la Sección 14.1.5 para más información sobre las características de los sistemas estudiados que usan RdP como base para su arquitectura). Las reacciones a tomar frente a los eventos se especifican como procedimientos codificados con RdP.

En el caso concreto de ProCoSa (véase la Sección 14.1.5 para más información sobre la arquitectura del sistema y [Barrouil and Lemaire, 1998]) se dispone de un monitor de ejecución (véase la Figura 12.3) compuesto de un motor de interpretación de RdP, bajo el nombre de *Petri Player*, el cual analiza los eventos y decide qué acción ejecutar. Estas acciones se codifican en lenguaje Lisp para ser interpretadas y ejecutadas por el sistema (véase la Sección 14.1.5 para más detalles sobre la arquitectura) mediante un intérprete de Lisp —i. e. *mini LISP interpreter* en la Figura 12.3.

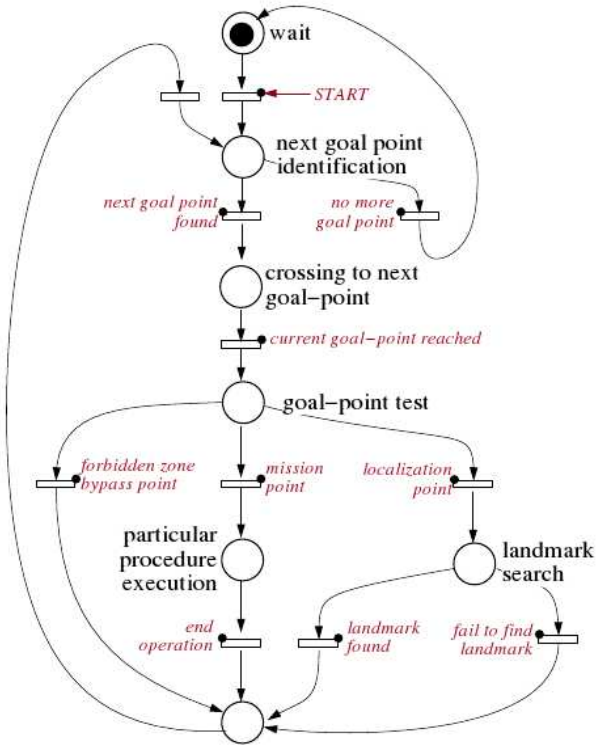


Figura 12.2: Especificación gráfica de la misión usando una RdP. Ejemplo de procedimiento de alto nivel especificado con ProCoSa

En el caso de la arquitectura de [Ramalho Oliveira et al., 1996] ocurre de forma similar a la de [Barrouil and Lemaire, 1998], i. e. se dispone de un software llamado CORAL —similar a ProCoSa en lo que se refiere a su función dentro de la arquitectura de especificación de misiones. Se trata de un conjunto de herramientas que permiten la construcción gráfica de las primitivas del vehículo mediante RdP y la ejecución de éstas en tiempo real en el sistema (véase la Sección 14.1.5 para más información sobre la arquitectura del sistema).

CORAL está compuesto por dos módulos fundamentales:

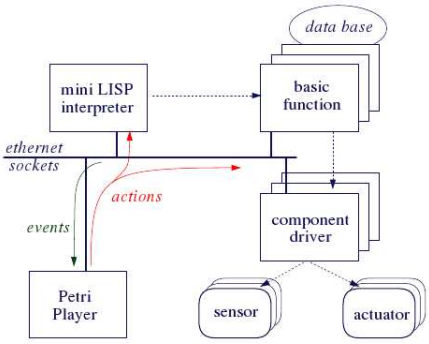


Figura 12.3: Ejecución de la misión con ProCoSa

1. Editor y generador de librerías de primitivas.
2. Motor de CORAL.

En la Figura 12.4 se observan los módulos que forman esta arquitectura. De arriba a abajo en la imagen se pasa de la edición y creación de las RdP gráficamente —con el *Vehicle Primitive Editor*— hasta la compilación para la posterior interpretación mediante el motor de CORAL.

El proceso utiliza inicialmente una representación gráfica de la RdP, o *Graphical Primitive Description* en la Figura 12.4, que se traduce a una representación textual, o *Textual Primitive Description* en la Figura 12.4. La Figura 12.5 muestra un ejemplo de traducción entre estas dos representaciones de RdP.

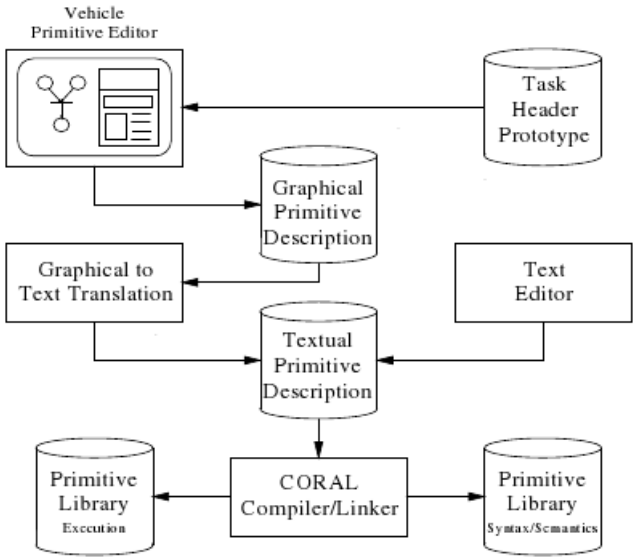


Figura 12.4: Programación de primitivas con CORAL. Flujo de tareas y recursos para la especificación de las primitivas de la misión

Tanto con ProCoSa como con CORAL se traduce la representación gráfica de la RdP a lenguajes que faciliten la interpretación y ejecución de la misma en tiempo real dentro

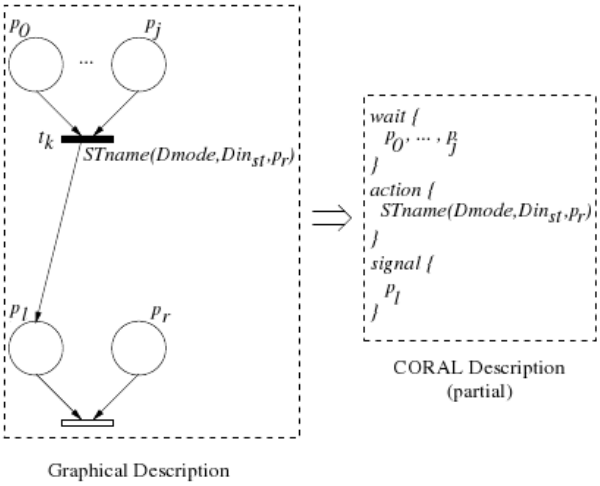


Figura 12.5: Representación gráfica de una RdP y representación textual equivalente mediante CORAL

del sistema —i. e. en lenguaje Lisp y un DSL (véase el Apéndice B) propio de CORAL, respectivamente.

A continuación se analizan las arquitecturas basadas en RdP según los criterios de calidad enunciados en la Sección 12.1:

Modularidad El uso de las RdP modulares (véase la Sección C.2.2) dota de gran modularidad a esta arquitectura. Todas las arquitecturas basadas en RdP estudiadas hacen uso de este tipo de RdP para facilitar el diseño del sistema (véase la Sección 14.1.5) y la especificación de misiones apoyada en herramientas gráficas.

Flexibilidad Las RdP y las herramientas de soporte que las acompañan, en especial las orientadas a la especificación de misiones —v. g. ProCoSa y CORAL— permiten la elaboración de nuevas RdP en base a primitivas u operaciones elementales. Además, gracias a las RdP modulares se puede escalar fácilmente la incorporación de nuevas primitivas. Estas características permiten la edición de múltiples tipos de misión con gran facilidad e, incluso, la incorporación de tareas no consideradas *a priori* en el análisis (véase el Capítulo 11). El grado de flexibilidad es muy alto.

Monitorización El propio formalismo de las RdP permite que la monitorización de la misión y el sistema sea sencillo y robusto (véase la Sección C.1). La marcación de la red —i. e. la ubicación de las marcas en los lugares de la RdP (véase la Sección C.1)— permite conocer en todo momento el estado de la misma y, por tanto, el de la misión.

Control remoto La RdP dispondría de primitivas preparadas para el control remoto, las cuales se comunicarían con el exterior para tomar la información de control e inyectarla en la RdP o el sistema. Disponiendo de este tipo de primitivas el control remoto se cubriría perfectamente por una arquitectura basada en RdP.

Facilidad de definición La definición de la misión se realiza de forma gráfica mediante la construcción de diagramas de RdP. Las herramientas que acompañan a las

RdP incorporan una GUI para la edición y creación de RdP que permite especificar la misión o elementos de la misma —v. g. interfaz *Vehicle Primitive Editor* de CORAL, mostrada en la Figura 12.4. Esto facilita enormemente la definición de la misión[4], si bien es necesario conocer su formalismo. No obstante, también se necesitan otras interfaces para la especificación de aspectos concretos de la misión —v. g. las tareas de navegación, para lo que suele disponerse de una GUI como la mostrada en la Figura 12.1. Aunque las RdP *per se* ya facilitan mucho la definición de la misión, la disponibilidad de herramientas adicionales es necesaria para satisfacer en grado máximo este criterio de calidad.

Portabilidad Al definir la misión con un diagrama de redes de Petri la portabilidad a alto nivel está asegurada —siempre que se disponga de herramientas software para la plataforma en que se esté trabajando (véase la Sección C.4 para conocer la disponibilidad de herramientas para RdP a nivel general). Sin embargo, los motores de interpretación de RdP trabajan con representaciones textuales de la misma que usan lenguajes de programación concretos —v. g. DSL propio de CORAL o Lisp si se usa ProCoSa. Esto obliga a desarrollar un traductor a lenguaje deseado o impuesto, en el peor de los casos. El grado de portabilidad puede considerarse medio si consideramos la interpretación de la misión; si sólo consideramos la definición, al reducirse a la edición de RdP, sería alto.

Reconfiguración y Aprendizaje La reconfiguración de la misión se consigue fácilmente modificando la RdP creada para la misión. El caso más simple lo constituye la sustitución de un módulo de la red, i. e. al definir la misión con un RdP modular, en el mejor de los casos, bastaría con reemplazar un módulo de la misma —que sería una RdP, que se sustituiría por otra. Respecto al aprendizaje, éste puede apoyarse en la RdP para modelar arquitecturas de aprendizaje simples. En el caso de usar una RdP generalizada (véase la Definición C.4), donde se pueden usar pesos diferentes de $w \in \{0, 1\}$, es posible aplicar técnicas de aprendizaje sobre los pesos de los arcos de la RdP.[5]

En conclusión, el aprendizaje es posible con esta arquitectura de especificación de misiones, pero se requiere el diseño de arquitecturas de aprendizaje dentro del sistema (véase el caso de la arquitectura estudiada en la Sección 14.1.5) —constituyéndose así una arquitectura híbrida. En conjunto, puede calificarse de alto el grado de reconfiguración y aprendizaje.

12.1.2. Sub-objetivos y Tareas

La especificación de la misión puede simplificarse de tal forma que consista en la especificación de un conjunto de tareas (véase la Definición 12.3) o sub-objetivos a realizar en un determinado momento y bajo unas determinadas condiciones. Sirva como ejemplo el caso de [Roberts et al., 2003], que plantea una arquitectura multicapa compuesta por tres

[4]Las herramientas incluyen traductores de las RdP a representaciones textuales de la misma mediante algún lenguaje de programación —v. g. DSL propio de CORAL del ejemplo de la Figura 12.5, lenguaje Lisp usado en ProCoSa (véase la Figura 12.3).

[5] Una RdP generalizada puede transformarse a una RdP ordinaria (véase la Definición C.3) equivalente, de acuerdo con lo explicado en la Sección C.2.1. Sin embargo, si se usa una RdP ordinaria el aprendizaje puede ser más complejo. Por otro lado, la implementación de un motor de interpretación de una RdP generalizada sería más complejo —lo habitual es que se interpreten RdP ordinarias.

capas (véase la Sección 14.1.6) y el de ITOCA [Ridao et al., 2005], con una arquitectura híbrida basada también en tres capas (véase la Sección 14.1.6 y Figura 12.6).

Definición 12.3 (Tarea). *Una tarea es un módulo de especificación de una misión en el que se define una o varias acciones que debe realizar el vehículo que ejecuta la misión. Una tarea puede tener diferentes parámetros configurables para poder especificar de qué forma concreta debe realizarse la tarea. También puede acompañarse de una serie de condiciones o restricciones de diversa índole para indicar esencialmente cuándo y dónde debe realizarse la tarea; esto es lo que se conoce como disparador (véase la Definición 13.1 y Sección 13.2.1 para ver un ejemplo de especificación).*

En [Roberts et al., 2003] la capa de más alto nivel de la arquitectura —que es la que aquí nos interesa— es la de aplicación y está formada por aplicaciones software, un integrador de aplicaciones y módulos de subtareas. En el caso de ITOCA esta capa se denomina de planificación y dispone igualmente de una *GUI*. Es común que las arquitecturas basadas en tareas dispongan de herramientas que faciliten la especificación de las mismas y su integración en la misión. [Roberts et al., 2003] concretamente incluye una interfaz de usuario, un intérprete del lenguaje de descripción de tareas y un algoritmo de supervisión para el submarino.

En esta arquitectura no se dispone de ningún formalismo —a diferencia de las arquitecturas basadas en RdP (véase la Sección 12.1.1)— y tanto la especificación como la representación de la misión suelen ser propias de cada arquitectura concreta. No obstante, siempre se sigue un mismo esquema general: la misión es descompuesta en una secuencia de tareas o sub-objetivos. La Figura 12.7 es representativa de este hecho, pues muestra las diferentes tareas que intervienen en una misión de exploración (*survey*); la propia exploración también puede considerarse como una tarea —incluso contenida en otra—, lo que permite la definición de una misión bastante modular y flexible (véase el análisis de los criterios de calidad de las arquitecturas de especificación de misiones basadas en tareas o sub-objetivos de las siguientes líneas).

En esta arquitectura la misión se especifica mediante la descripción de las tareas a realizar, de modo que la arquitectura del sistema debe integrar los diferentes módulos de tareas y permitir la gestión de las mismas (véase la Sección 14.1.6). En la Figura 12.6 se muestra como el sistema de ITOCA dispone de módulos de tareas, o *Task Module* en la figura, encargados de la ejecución de éstas. El lenguaje empleado para la descripción de la misión suele ser un DSL (véase el Apéndice B) basado en tareas. Normalmente se adapta el lenguaje TDL (Lenguaje de Descripción de Tareas) [Simmons and Apfelbaum, 1998], que fue diseñado para simplificar el desarrollo de programas de control para robots como una extensión de C++ sobre una biblioteca de comunicaciones denominada TCM (Manejo de Tareas de Control). El lenguaje TDL incorpora descomposición de tareas, monitorización de la ejecución, sincronización entre tareas concurrentes y manejadores de excepciones. En [Simmons and Apfelbaum, 2003] se ofrece el manual de referencia y el compilador de TDL (TDLC). En [Roberts et al., 2003] se emplea un lenguaje basado en TDL, denominado STDL (SAUVIM TDL) y que se implemente en Basic.

De acuerdo con los criterios de calidad de la arquitectura de la misión enumerados en la Sección 12.1, el análisis de esta arquitectura es el siguiente:

Modularidad Los módulos de la arquitectura son las tareas —habitualmente descritas con un DSL. No existe una modularización de un nivel superior, como pudieran

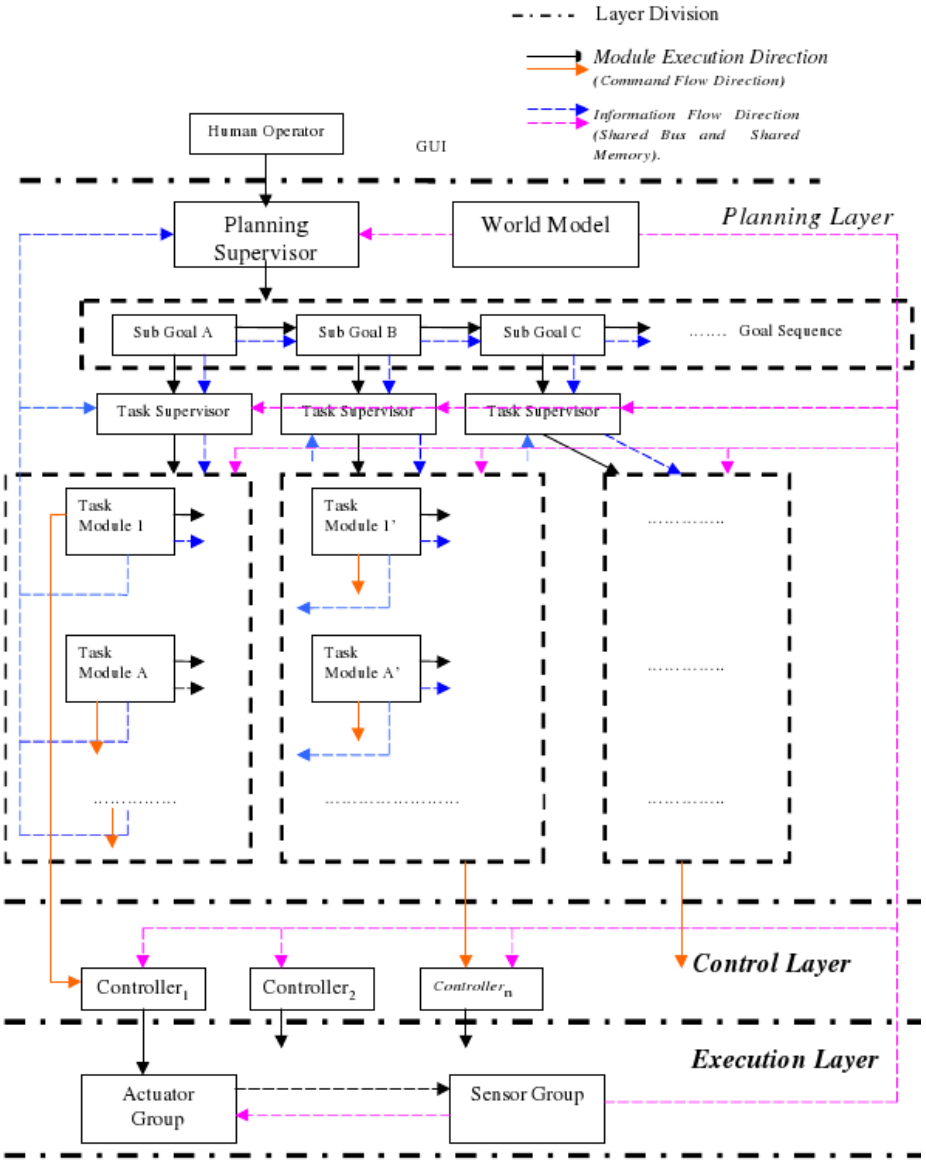


Figura 12.6: Arquitectura ITOCA. Módulos de gestión para tareas especificables en la misión

ser los planes de la misión (véase la Sección 12.2). Es posible conjuntar tareas para conseguir diferentes tipos de misión o la combinación de varias según la tipología a la que pertenezcan las tareas (véase la Sección 11.7). El grado de modularización se considera medio.

Flexibilidad La misión se describe a partir de tareas básicas o primitivas. Dependiendo del tipo de primitivas disponibles o desarrolladas, la especificación de la misión podrá ser más o menos flexible, i. e. si existe alguna funcionalidad básica no implementada, no será posible desarrollar las misiones que la requieran. Además, modificar alguna funcionalidad requiere modificaciones importantes en el software.

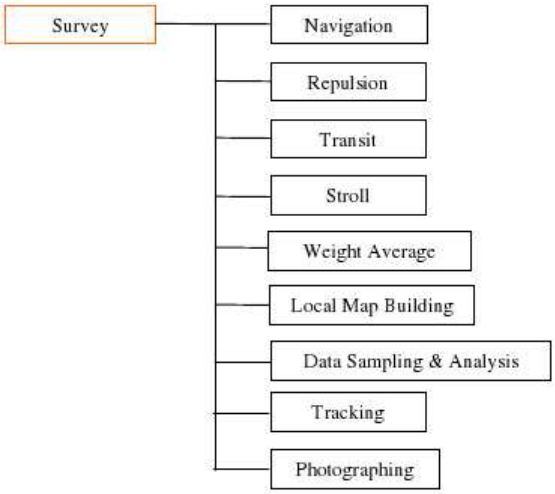


Figura 12.7: Ejemplo de tareas involucradas en una misión de exploración (*survey*), según la arquitectura ITOCA

En estos casos la flexibilidad se resiente, pero siempre que esto no ocurra el grado de flexibilidad será alto.

Monitorización La estabilidad y controlabilidad son fácilmente verificables; se dispone de la capacidad de evaluación del grado de realización de la misión, i. e. se puede evaluar la realización de cada una de las tareas que la componen o si ha ocurrido algún error con alguna. El nivel de realización interno de cada tarea dependerá de la implementación de la misma, por lo que el usuario no podrá modificarlo. A nivel de la misión el estado en que se encuentra cada tarea se modelará igual para cualquiera de ellas. Esto resta cierta versatilidad en la monitorización, de modo que se considera que se cubre con un grado medio.

Control remoto El control remoto se conseguirá mediante tareas encargadas de tomar información de control del exterior e inyectarla al sistema o a otras tareas de la misión. Esto permite integrar fácilmente el control remoto en la misión. No obstante, en este tipo de arquitectura el sistema suele proporcionar herramientas específicas para ello (véase la Sección 14.1.6).

Facilidad de definición La definición de la misión requiere el dominio del DSL que proporcione la arquitectura, aunque prácticamente siempre se dispone de una GUI que facilita la tarea de definición de la misión y sus elementos —i. e. las tareas en este caso— (véase la Apéndice A). El nivel de abstracción de las tareas es próximo a la semántica de las misiones de AUVs en general, por lo que el grado de facilidad de definición es alto siempre que se acompañe de herramientas de apoyo al estilo de la ya mostrada en la Figura 12.1 —que aunque pertenece a una arquitectura basada en RdP es común a prácticamente la totalidad de arquitecturas.

Portabilidad La misión se suele definir en un lenguaje específico como un DSL (véase el Apéndice B), v. g. STDL en el caso de [Roberts et al., 2003]. Para interpretar la misión se requiere un intérprete del DSL. Al generar la misión también se requiere

de herramientas que representen la misma en el DSL. Esto hace que el grado de portabilidad sea bajo si consideramos el proceso desde la definición hasta la interpretación de la misión; considerando sólo la definición podría decirse que es medio —supeditado a la disponibilidad de herramientas que faciliten la definición de la misión.

Reconfiguración y Aprendizaje Es posible reconfigurar las tareas que componen la misión. Se permite la creación y destrucción de tareas, así como el control de la sincronización y comunicación entre ellas. Igualmente, si el DSL lo permite, se podrán modificar determinados parámetros de configuración de tareas concretas; no todas las arquitecturas de tareas soportan esta característica, tanto a nivel del DSL de especificación de la misión como en el sistema. La arquitectura no soporta aprendizaje, de modo que tendría que implementarse como un elemento adicional en la arquitectura del sistema (véase la Sección 14.1.6). El grado de reconfiguración y aprendizaje se considera bajo en su conjunto.

12.1.3. Comportamientos

Es posible realizar la especificación de misiones mediante comportamientos (véase la Definición 12.4). La misión estará formada por un conjunto de comportamientos que modelan las acciones que debe realizar el vehículo. De acuerdo con los diferentes tipos de misiones del Capítulo 11 se podrán especificar comportamientos para cada uno de ellos. Es común que se definan comportamientos diferentes para un mismo ámbito —v. g. navegación— que se tendrán que coordinar por el sistema (véase la Sección 14.1.7).

Definición 12.4 (Comportamiento). *Un comportamiento es un patrón o conjunto de acciones que realiza un ente en relación con su entorno o los estímulos que percibe del mismo.*

En [Carreras Pérez, 2003] se realiza un estudio de diferentes arquitecturas de sistemas de control y se propone una arquitectura de control híbrida —i. e. deliberativa y reactiva— basada en comportamientos (véase la Sección 14.1.7). La arquitectura de especificación de misiones para este tipo de sistema se podrá modelar en base a la especificación de los diferentes comportamientos que puede realizar el vehículo.

En cierto modo, la especificación de un comportamiento concreto será un elemento configurable muy similar a una tarea (véase la Definición 12.3). Sin embargo —a diferencia de lo que ocurre con las tareas—, en la especificación de la misión se tendrá que especificar de qué forma se coordinan los comportamientos (véase la Figura 14.3). Un componente denominado **coordinador** es el encargado de este proceso de coordinación, que puede usar métodos competitivos o cooperativos, aunque se suele adoptar una arquitectura híbrida para aprovechar las ventajas de ambos (véase Sección 14.1.2).

La arquitectura propuesta por [Carreras Pérez, 2003] dispone de Aprendizaje por Refuerzo (RL), concretamente el algoritmo *Semi-Online Neural-Q_learning* (SONQL), que es una adaptación del algoritmo de Diferencias Temporales (TD) Q_learning, diseñado para aprender con un espacio continuo de estados y acciones. En la Figura 12.8 puede verse el diagrama de coordinación de comportamientos y aprendizaje adoptado. En [Roberts et al., 2003] también se emplea una arquitectura basada en comportamientos, a los cuales se les asignan diferentes pesos, de modo que el coordinador escoge aquellos

comportamientos con mayor peso. Todo este tipo de arquitecturas puede beneficiarse de una especificación de misiones basada también en comportamientos, ya que se ajustaría perfectamente a la arquitectura del sistema. No obstante, esto no es una imposición y puede usarse cualquier otro tipo de especificación diferente.

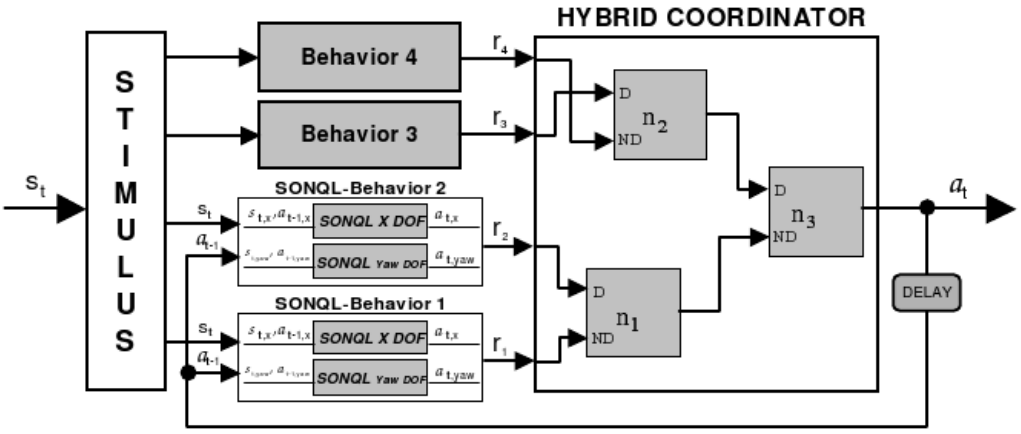


Figura 12.8: Control basado en comportamientos. Ejemplo de coordinación de varios comportamientos

De acuerdo con los criterios de calidad de la arquitectura de la misión enumerados en la Sección 12.1, el análisis de esta arquitectura es el siguiente:

Modularidad Los módulos de la arquitectura son los comportamientos. El grado de modularidad de la especificación de la misión dependerá del mecanismo de coordinación entre comportamientos que se adopte —igual que ocurre con la arquitectura del sistema (véase la Sección 14.1.7).

Si se emplea un método competitivo los comportamientos serán modulares, pero si es cooperativo no lo serán porque deben desarrollarse de forma dependiente entre ellos para que la aportación de cada comportamiento sea la apropiada a la salida final del coordinador.

El lenguaje de especificación de los comportamientos será *ad hoc*, como ocurría con las tareas (véase la Sección 12.1.2), i. e. un DSL. No obstante, también podrían modelarse con RdP[6]. En cualquiera de los casos, el factor con más peso en la modularidad de la arquitectura es el mecanismo de coordinación.

Incluso con un coordinador híbrido —como el de la Figura 14.3 (c)— habrá comportamientos que deban coordinarse de forma cooperativa; la modularidad se verá resentida en estos casos. En suma se considera que el grado de modularidad es medio.

Flexibilidad La misión se describe mediante comportamientos. Se dispondrá de una serie de comportamientos parametrizables que cubrirán las necesidades de los tipos de misiones estudiados en Capítulo 11. Si se da cobertura a toda la tipología de

[6]Si los comportamientos se definen mediante RdP será necesario desarrollar un motor de interpretación de RdP, al menos. Por este motivo, normalmente se suele optar por un DSL de fácil definición e interpretación.

misiones el grado de flexibilidad será alto, pero si falta alguna funcionalidad será necesario desarrollarla desde cero.

Monitorización La coordinación de varios comportamientos en el sistema dificulta la monitorización, especialmente en el caso del método cooperativo de coordinación; en el método competitivo hay un solo comportamiento actuando sobre el sistema. Será posible monitorizar el nivel de realización interno de cada comportamiento de forma independiente, pero se requerirán mecanismos adicionales para determinar el estado del sistema según la intervención del coordinador. En principio, se considera que el grado de monitorización es bajo.

Control remoto El control remoto se conseguirá mediante comportamientos o componentes internos del sistema encargados de tomar información de control del exterior e inyectarla al sistema. El sistema debe proporcionar herramientas específicas para ello (véase la Sección 14.1.7), permitiendo influir en los comportamientos individualmente o a nivel de coordinación de los mismos.

Facilidad de definición La definición de la misión requiere el dominio del DSL que proporcione la arquitectura para especificar los comportamientos. Se suele disponer de una GUI que facilita esta tarea (véase la Apéndice A). La especificación en base a comportamientos es próxima a la semántica de las misiones de vehículos robóticos, por lo que el grado de facilidad de definición es alto; más aún si se acompaña de herramientas software. Sólo en el caso de la definición de comportamientos que se coordinarán cooperativamente habrá dificultades debidos a las dependencias entre comportamientos.

Portabilidad Al definir los comportamientos de la misión con un DSL se requiere un intérprete del mismo y herramientas de representación de la misión especificada. El grado de portabilidad es bajo si consideramos el proceso desde la definición hasta la interpretación de la misión, pero considerando sólo la definición podría decirse que es medio; ocurre igual que con la arquitectura basada en tareas (véase la Sección 12.1.2).

Reconfiguración y Aprendizaje Los comportamientos serán reconfigurables, modificando los parámetros que los definan. Además, el sistema coordina los comportamientos *per se* y dispone de potentes mecanismos de aprendizaje integrados [Carreras Pérez, 2003] (véase la Sección 14.1.7). El grado de reconfiguración y aprendizaje es muy alto.

12.1.4. Conclusiones

El Cuadro 12.1 muestra los resultados de la evaluación de las arquitecturas estudiadas, en base a los criterios enumerados al inicio de la Sección 12.1. De estos resultados se concluye que en general todas las arquitecturas resultan apropiadas para la especificación de misiones, aunque no satisfagan con garantías algunos de los criterios de evaluación.

La arquitectura basada en RdP obtiene la mejor nota y esto es debido especialmente a los siguientes motivos:

	Red de Petri	Tareas	Comportamientos
Modularidad	★★★★★	★★★	★★★
Flexibilidad	★★★★★	★★★★	★★★★
Monitorización	★★★★	★★★	★★
Control remoto	★★★★	★★★★	★★★★
Facilidad de definición	★★★	★★★★	★★★★
Portabilidad	★★★	★★★	★★★
Reconfiguración y Aprendizaje	★★★★	★★	★★★★★
Total	★★★★	★★★	★★★

Cuadro 12.1: Arquitecturas de especificación de la misión. Evaluación de los criterios de calidad

1. Las RdP son un formalismo *per se*. Debido a esto hay aspectos que se cubren con garantías, ya que así lo constata la bibliografía (véase el **??**). Esto será especialmente importante en el sistema, al que ofrecerá robustez y fiabilidad (véase la Sección 14.1.5) —v. g. la flexibilidad de definición es muy alta gracias a la potencialidad de representación de las RdP, la monitorización se puede realizar en base a la marcación de la red, etc.

2. El uso de RdP modulares (véase la Sección C.2.2) ofrece un nivel de modularidad muy alto.

3. Al constituir una representación gráfica y disponerse de herramientas software para la construcción y simulación, aspectos como la portabilidad y la facilidad de definición se cubren con solvencia.

4. La reconfiguración y aprendizaje pueden incluirse con cierta facilidad en la propia red, aunque no sea algo trivial.

A pesar de tratarse de la alternativa con mejor nota, las RdP imponen el uso de una serie de herramientas y la necesidad de manejar correctamente su formalismo. Por este motivo, aunque las otras arquitecturas estén peor valoradas, la mejora en los aspectos donde flaquean puede permitir una especificación de misiones apropiada, como la propuesta en la Sección 12.2, la cual está basada en la arquitectura de tareas.

En lo relativo a las arquitecturas basadas en tareas su principal punto débil radica en la necesidad de desarrollo y manejo de un DSL para la especificación de la misión (véase el Apéndice B); se resiente especialmente la portabilidad, dado que hay que disponer de herramientas de definición e interpretación propias. Al margen de esto, el resto de criterios se cubren con solvencia, a excepción de los siguientes:

1. La modularidad sólo es a nivel de tareas, por lo que puede mejorarse como en el caso del uso de los planes (véase la Sección 12.2).

2. La monitorización es a nivel del estado de realización de las tareas, cuyo detalle está estrechamente ligado a cada tarea y no puede modificarse sin reprogramar el sistema y la especificación concreta de la tarea en cuestión.

3. El aprendizaje no está soportado por la misión y las posibilidades de reconfiguración se reducen a los parámetros de configuración de las tareas.

A pesar de estos carencias, es posible mejorar sustancialmente la arquitectura incidiendo precisamente en estos aspectos, tal y como se muestra en la Sección 12.2.

Las arquitecturas basadas en comportamientos tienen una valoración similar a las de tareas, tanto a nivel general como en la mayoría de criterios. Sólo existen dos criterios de evaluación donde difieren:

1. El grado de monitorización es peor debido a la dificultad que impone la coordinación de comportamientos, especialmente si se hace uso de un coordinador cooperativo —lo cual es bastante habitual— (véase la Sección 12.1.3).

2. Las arquitecturas basadas en comportamientos incorporan mecanismos de aprendizaje en el sistema —en especial [Carreras Pérez, 2003]— (véase la Sección 14.1.7). Por este motivo se consigue un alto grado de aprendizaje, que afectará directamente sobre la coordinación de los comportamientos especificados en la misión.

Al enfrentar comparativamente las arquitecturas basadas en tareas y comportamientos se estima la especificación mediante tareas como la más apropiada, por resultar semánticamente más fáciles de definir y la monitorización del sistema durante la ejecución de la misión es más sencilla y fiable —más aún si se mejora la arquitectura como se muestra en la Sección 12.2.

El estudio de la bibliografía consultada sobre arquitecturas de especificación de misiones (véase la Sección 12.1) muestra como de forma general se aplican arquitecturas basadas en tareas. Suele ser la arquitectura con un tiempo de desarrollo más corto a pesar de tener que desarrollar interfaces para la especificación de la misión y traducción al DSL con el que se representan las tareas de la misión, así como intérpretes de las mismas. En el caso de las arquitecturas basadas en RdP ya se dispone de este tipo de software (véase la Sección C.4), pero requiere de su adaptación al ámbito de especificación de misiones. Por ello, como la especificación basada en tareas es una solución *ad hoc*, suele ser la más adoptada. No obstante, hay varios proyectos que destacan por la aplicación de las RdP [Barrouil and Lemaire, 1998, Ramalho Oliveira et al., 1996]. Respecto a las arquitecturas basadas en comportamientos, éstas resultan una aproximación al problema bastante común en sistemas robóticos y que combinadas con mecanismos de aprendizaje como en [Carreras Pérez, 2003] donde resulta interesante el estudio de su aplicabilidad para vehículos submarinos.

Al consultar la bibliografía solemos encontrarnos con estudios o propuestas de arquitecturas orientadas al sistema del vehículo o, más concretamente, de arquitecturas de control. No aparece un análisis o explicación de arquitecturas de especificación de misiones propiamente dicho, sino ligeras reseñas a la solución adoptada, quedando relegado a un segundo plano este aspecto.[7] En cualquier caso, la arquitectura empleada para especificar las misiones es bastante importante, pues determina la tipología de misiones que podrá realizar el sistema integrado en el vehículo.

12.2. Planes de la Misión. Arquitectura Propuesta

En la arquitectura propuesta la misión se definirá de forma separada en varios planes (véase la definición de plan en la Definición 12.5). Se reducirá al mínimo las interde-

[7] En el caso de las arquitecturas basadas en RdP, la estrecha relación entre la especificación de la misión y el sistema hace que se deduzcan más fácilmente las características de esta arquitectura.

Plan de	Abreviatura	Descripción
Almacenamiento	PdA	Almacenamiento de datos y gestión del inventario
Comunicación	PdC	Comunicación de datos
Medición	PdM	Toma de muestras de medidas
Navegación	PdN	Navegación: seguimiento de ruta, exploración de área o seguimiento de medida
Supervisión	PdS	Supervisión del sistema y actuación frente a excepciones

Cuadro 12.2: Planes de la misión

pendencias entre los planes, favoreciendo la posibilidad de intercambiar los planes entre distintas misiones; esto se conseguirá gracias a que el sistema del AUV (véase la Parte III) dispondrá de la potencialidad y capacidad de decisión suficiente para integrar correctamente los planes sin que éstos sean dependientes entre sí. En cualquier caso siempre se realizarán chequeos de integridad entre los planes, como la ya comentada validación de la misión (véase la Apéndice A y Sección 12.3).

Definición 12.5 (Plan). *Definiremos un plan de la misión como un elemento donde se especifica un conjunto de tareas relacionadas entre sí, dentro de un ámbito concreto, común e independiente del resto de planes.*

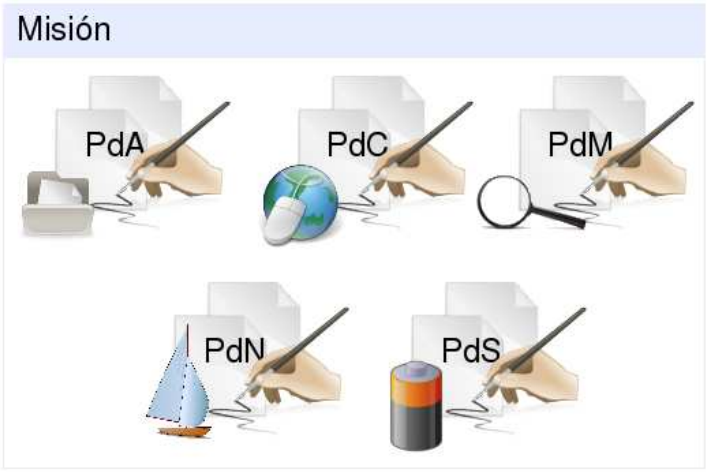


Figura 12.9: Arquitectura de especificación de la misión mediante planes

La arquitectura propuesta para especificar la misión constará de los siguientes planes (véase el Cuadro 12.2 para una descripción breve y la Figura 12.9 para una representación gráfica resumida de la arquitectura):

Plan de Almacenamiento (PdA) También se puede considerar como la configuración de la *Caja Negra*, Registro o *Log* del sistema. Contiene las tareas de almacenamiento de datos y la gestión del inventario de datos (véase el Sección 15.3). Los datos almacenables serán las medidas muestreadas por los sensores y las variables observables del sistema del AUV (véase la Sección 15.7 para conocer los

detalles de la gestión y el tipo de elementos almacenables, para los que se plantea un tratamiento homogéneo).

Plan de Comunicación (PdC) Contiene las tareas de comunicación a realizar durante la ejecución de la misión. Las capacidades de comunicación del vehículo están estrechamente ligadas al equipamiento que posee (véase la Parte I). Esto influirá inevitablemente en las posibles tareas de comunicación realizables (véase el Ejemplo 12.1).
Durante el proceso de validación de la misión será posible detectar incompatibilidades (véase la Definición 12.8 y Sección 12.3.1) entre el PdC y el equipamiento del AUV. Igualmente, puede ocurrir que la realización de las tareas de comunicación sea incompatible con las tareas del resto de planes; en el Ejemplo 12.2 se muestra el caso más típico de incompatibilidad, que se puede producir entre el PdC y el PdN, en función del equipamiento de comunicación.

Plan de Medición (PdM) También puede denominarse Plan de Medidas. Contiene las tareas de medición de medidas proporcionadas por el AUV según las capacidades de su equipamiento. Permite indicar las medidas a muestrear sin necesidad de seleccionar un sensor concreto; en este caso el sistema del AUV (véase la Sección 15.7) se encargará de seleccionar automáticamente el sensor apropiado para medir de forma transparente para el usuario. También se permite indicar el sensor concreto o un conjunto de sensores que pueden usarse para tomar la medida; en el caso de indicarse un conjunto se procederá igual que al no indicar ninguno, pero la elección automática del sensor estará limitada al conjunto indicado.

Plan de Navegación (PdN) Contiene las tareas de navegación que indican por dónde debe ir el AUV. Esto se puede indicar con cualquiera de los tipos de misión soportados, comentados en la Sección 11.2[8], Sección 11.3 y Sección 11.4, i. e. seguimiento de ruta, exploración de área y seguimiento de medidas, respectivamente, o la combinación de ellas.

Plan de Supervisión (PdS) Contiene las tareas a realizar en caso de producirse problemas en la ejecución de la misión o en el sistema del AUV. Las tareas de este plan determinan las acciones a realizar en el caso de producirse determinados eventos o excepciones, si bien el soporte está generalizado a cualquier tipo de disparador (véase el Sección 13.2.1). El sistema controlará la ejecución de los anteriores planes de la misión usando el PdS para determinar las acciones a tomar en caso de ser necesario y poder continuar ejecutando la misión. Las tareas indicadas en el PdS serán acciones de cualquier tipo de plan, pero con una especificación homogénea basada en comandos (véase la Definición 15.1 y Sección 15.1). También se da soporte a la ejecución de planes de contingencia (véase la Sección 13.7 para más información).

La arquitectura propuesta está basada principalmente en las arquitecturas de tareas y sub-objetivos, estudiadas en la Sección 12.1.2. Con los planes de la misión se pretende mejorar la especificación de misiones basada en tareas aportando varias características deseables que no se encuentran presentes en éstas. Los planes de la misión se definirán

[8]El tipo de misión de toma de muestras —boya— (véase la Sección 11.1) queda incluido en el de seguimiento de ruta (véase la Sección 11.2).

con la sintaxis comentada en el Capítulo 13, donde se detallan las características de cada plan individualmente.

En el Cuadro 12.1 se observa que las arquitecturas de especificación de misiones basadas en tareas y sub-objetivos no son la mejor opción, en comparación con otras, de acuerdo con la valoración realizada en la Sección 12.1.2. No obstante, se trata de una alternativa que recibe una puntuación media —frente a una puntuación alta de las RdP, por ejemplo— y no es una mala opción (véase la Sección 12.1.4).

Con los planes de la misión se aportan las siguientes características adicionales para mejorar las arquitecturas basadas en tareas estudiadas:

1. Los planes de la misión tienen la finalidad de agrupar tareas que sean de una tipología bien diferenciada, como se observa en la división de los planes realizada y explicada anteriormente. Esto ofrece una especificación modular de la misión, que permite reaprovechar e intercambiar planes entre diferentes misiones.

2. Los planes de la misión no sólo permiten que la misión se defina de forma modular, sino también que se interprete y gestione modularmente. Por este motivo, la propia arquitectura del sistema puede beneficiarse de ello a la hora de diseñarse. Éste es el caso de *Sick***AUV**, donde varios de los subsistemas en que se subdivide disponen de intérpretes específicos de cada plan de la misión, como muestra el Cuadro 12.3. Esto desacopla la ejecución de cada plan de la misión, repartiendo mejor la carga de proceso de toda la misión, amén de otras bondades comentadas en el Capítulo 15.

3. Como lenguaje de especificación de la misión se usa XML como base. La sintaxis final del lenguaje se define mediante esquemas XML —XSD— (véase la Sección E.5), que pueden verse en el Capítulo 13. El uso de XML como lenguaje se justifica con los siguientes motivos:

   a) Es un meta-lenguaje basado en etiquetas, por lo que su lectura es bastante cómoda, tanto para un programador como para un experto en la materia representada, i. e. las misiones de vehículos submarinos en este caso. Esto es posible porque se puede conseguir un lenguaje con una semántica muy cercana al dominio representado.

   b) XML está muy extendido hoy en día y existe gran variedad de herramientas software para trabajar con él (véase la Sección E.7) en diversas plataformas. Esto reduce enormemente la necesidad de programación, al mismo tiempo que disponemos de gran portabilidad para la especificación, validación e interpretación de las misiones (véase la Sección 12.3). La mayoría de aplicaciones ya están realizadas, especialmente los analizadores sintácticos (*parser*), pero también las librerías para construcción de ficheros XML.

   c) La especificación del lenguaje creado en base a XML se puede indicar en ficheros XSD, que se denominan esquemas XML. Esto permite validar o analizar sintácticamente los planes de la misión directamente. Además, la modificación del lenguaje es bastante simple y se puede distribuir fácilmente.

4. Aparte de los planes, la misión se complementa con una lista de parámetros configurables. Estos parámetros de la misión facilitan la modificación de un determinado aspecto de la misma, incluso durante su ejecución. Se usa XPath (véase la Sección E.6) como lenguaje de especificación de los elementos a modificar en el

Plan de	Subsistema
Almacenamiento	de Almacenamiento
Comunicación	de Comunicación
Medición	Sensorial
Navegación	de Guiado
Supervisión	de Supervisión

Cuadro 12.3: Subsistemas de *Sick***AUV** encargados de interpretar los planes de la misión

lenguaje utilizado. Los parámetros de la misión son una lista de atajos de consultas de XPath, ya que en realidad se podrá modificar cualquier elemento de la misión usando esta tecnología.

5. La especificación de la misión basada en planes se complementa con una GUI para definirlos y validarlos, que forma parte de un proyecto relacionado con éste (véase la Apéndice A). Esto ofrece mayor facilidad a la especificación de la misión, mediante herramientas orientadas a cada plan. Además, gracias al proceso de validación (véase la Definición 12.7) de la misión durante la carga e interpretación de la misma dentro del sistema, la lógica será más simple, pues muchos chequeos ya no serán necesarios —v. g. chequeos de tipos de datos, atributos o elementos XML obligatorios, etc.

De acuerdo con los criterios de calidad de la arquitectura de la misión enumerados en la Sección 12.1, el análisis de la arquitectura propuesta es el siguiente:

Modularidad Los módulos de la arquitectura son los planes. Varios de los planes se describen internamente en base a tareas, pero no se trata de una imposición a todos los planes. De hecho, el PdN no se define en base a tareas, pues resulta más adecuado una especificación *ad hoc* de acuerdo con la tipología de misiones de navegación (véase la Sección 11.2, Sección 11.3 y Sección 11.4); en el Capítulo 13 puede verse la especificación adoptada, más cercana a la semántica de esta tipología de misiones que las tareas. No obstante, los tipos de navegación posibles —e.g. ruta, área, seguimiento de una medida—, pueden considerarse tareas, aunque su especificación e interpretación sería secuencial y no paralela, como ocurre con el resto de planes de la misión.
La modularidad de las tareas combinada con la de los planes de la misión permite un alto grado de modularidad.

Flexibilidad Los planes de la misión se describen mediante tareas, salvo en el caso del PdN, donde se usa una especificación *ad hoc* (véase la Sección 13.6). La forma en que están definidas las tareas está orientada a cubrir toda la tipología de misiones del Capítulo 11 al mismo tiempo que se intenta dar la mayor flexibilidad posible con los parámetros de configuración de las mismas (véase el Capítulo 13). Con esto se obtiene un grado de flexibilidad muy alto, pues se cubre prácticamente todo el universo de posibles misiones especificables para vehículos submarinos.

Monitorización Se mantiene la estabilidad y controlabilidad de las arquitecturas de tareas y sub-objetivos. Es posible monitorizar el grado de realización de la misión, los planes y sus tareas. El sistema subyacente ofrece las herramientas necesarias

para monitorizar el grado de realización de las diferentes acciones o comandos (véase la Definición 15.1) que componen una tarea mediante servicios (véase la Definición 15.2 y Capítulo 15 para conocer los detalles del sistema *Sick***AUV** propuesto). No obstante, el modelo de gestión de todas las tareas —de los comandos que contienen y se tramitan como servicios— será equivalente, restando versatilidad a la monitorización en casos muy concretos. Como estos casos serán muy pocos, dado que se da soporte para toda la tipología de misiones del Capítulo 11, el grado de monitorización es alto.

Control remoto El control remoto se conseguirá mediante módulos específicos para ello, como parte del subsistema de comunicación (véase la Sección 15.4). Esto libera a la misión de la especificación de la activación o configuración del control remoto. Desde el punto de vista de la especificación de la misión, ésta permite el control remoto sin "efectos secundarios" sobre el sistema. Para ello, las tareas —o sus acciones— se pueden inhibir o desactivar fácilmente, con el fin de evitar interferencias con el control remoto; el sistema se encargará de esto de forma transparente, afectando sólo a las tareas o acciones problemáticas.

Facilidad de definición La definición de la misión se representará textualmente en XML, tal y como se observa en el Capítulo 13. El uso de XML facilita especialmente la edición de misiones (véase el Sección E.3), pero aún así se dispondrá de varias GUI para la especificación de misiones (véase la Apéndice A).
Como el nivel de abstracción de las tareas es próximo a la semántica de las misiones de AUVs, el grado de facilidad de definición es alto siempre que se acompañe de herramientas de apoyo (véase la Sección 12.1.2 y Figura 12.1).

Portabilidad La misión se define en XML, con una sintaxis concreta que puede consultarse en el Capítulo 13. Cada plan de la misión dispone de un esquema XML (XSD) propio, aunque en general todos se basan en tareas (véase el Sección 13.2) —salvo el PdN. El uso de XML como base para construir el lenguaje de especificación de la misión nos permite aprovechar multitud de herramientas software para la validación e interpretación de la misión —concretamente de cada uno de sus planes— (véase la Sección 12.3 y Sección E.7 para más detalles sobre el software disponible y el utilizado en *Sick***AUV**). Este tipo de herramientas están muy extendidas y son —en su mayoría— multiplataforma, por lo que el grado de portabilidad es muy alto.

Reconfiguración y Aprendizaje Las tareas de los planes de la misión y los elementos de especificación del PdN —que no usa tareas— están elaboradas para facilitar la reconfiguración de la misión. En la Sección 13.9 se explica la sintaxis de especificación de parámetros de la misión. Se trata de una herramienta basada en XPath (véase el Sección E.3) que permite definir nombres para parámetros reconfigurables de la misión. No obstante, se pueden usar construcciones de XPath directamente para modificar cualquier especificación de los planes de la misión.
El aprendizaje se implementará como parte del sistema, pero podrá beneficiarse de esta capacidad de reconfiguración de la misión para la adaptación de la misión. El grado de reconfiguración y aprendizaje se considera alto en su conjunto, aunque hay que destacar que las capacidades de reconfiguración no determinan el grado

	Nota	Comparativa
Modularidad	★★★★★	Se mejora la arquitectura basada en tareas con la incorporación de los planes. Se alcanza el nivel de la arquitectura basada en RdP.
Flexibilidad	★★★★★	Ligera mejora respecto a la arquitectura basada en tareas gracias a la gran capacidad de configuración de las tareas y la especificación *ad hoc* del PdN.
Monitorización	★★★★	Ligera mejora respecto a la arquitectura basada en tareas gracias a la definición de tareas compuestas por acciones que el sistema *Sick***AUV** gestiona mediante servicios.
Control remoto	★★★★	Se mantiene la misma nota que la arquitectura basada en tareas —igual que la de RdP.
Facilidad de definición	★★★★	Se mantiene la misma nota que la arquitectura basada en tareas.
Portabilidad	★★★★★	Se mejora la arquitectura basada en tareas debido al uso de XML como base para el lenguaje de representación de la especificación de la misión.
Reconfiguración y Aprendizaje	★★★★	Se mejora la arquitectura basada en tareas —alcanzando la misma nota que la de RdP— al introducir los parámetros de la misión reconfigurables, soportados por XPath.
Total	★★★★	En comparación con la arquitectura basada en tareas se consigue una mejora importante, de modo que la arquitectura de planes de la misión es tan alta como las otras arquitecturas —i. e. RdP y comportamientos.

Cuadro 12.4: Valoración y comparativa de la arquitectura de especificación de misiones basada en los planes de la misión frente al resto de arquitecturas estudiadas en la Sección 12.1. Mejoras respecto a la arquitectura basada en tareas y sub-objetivos (véase la Sección 12.1.2), en la cual está basada la arquitectura de planes de la misión

de aprendizaje, sino que simplemente permiten que el aprendizaje pueda realizarse sobre la especificación de la misión.

En el Cuadro 12.4 se muestra el resumen de la valoración de la arquitectura realizada, donde la arquitectura se considera alta en general.

12.3. Ciclo de Vida. Tareas aplicables a una misión

La especificación de la misión está sujeta a un ciclo de vida a lo largo del cual se realizan diferentes tareas con la misión (véase la Definición 12.6). Este ciclo de vida es aplicable a cualquier arquitectura de especificación de misiones. Sin embargo, según la arquitectura es posible que determinadas fases sean innecesarias.

Definición 12.6 (Ciclo de vida (misión)). *El ciclo de vida de una misión hace alusión a las diferentes fases por las que pasa una especificación de la misión, desde su creación hasta la finalización de la ejecución de la misma en el vehículo.*

Definición 12.7 (Validación). *Dentro de la terminología de computadores, la validación se refiere al proceso de comprobación del cumplimiento de una serie de especificaciones.*
En el caso concreto de la validación XML se trata de la comprobación de que un documento en lenguaje XML está bien formado y se ajusta a una estructura definida —v. g. DTD, esquema XML o XSD— (véase el Sección E.3).

A continuación se muestran secuencialmente las fases del ciclo de vida de la arquitectura de planes de misión de la Sección 12.2, de acuerdo con sus características y sintaxis (véase el Capítulo 13).[9]

1. **Creación:** Proceso de creación o edición de la misión. En el caso de la arquitectura basada en los planes de la misión se dispondrá de un planificador (véase la Apéndice A) con una GUI especializada en la especificación de cada plan. Durante la edición de la misión de forma gráfica, internamente se manejará una representación textual —v. g. DSL, representación de RdP, DSL basado en XML en el caso de la arquitectura de planes de misión (usando la sintaxis comentada en el Capítulo 13), etc.

2. **Validación:** Cuando toda la misión está definida se puede realizar un proceso de validación mediante el propio planificador de la misión (véase la Apéndice A). Éste hará uso de la representación textual interna que maneja de la especificación de la misión para analizar si hay errores o incompatibilidades (véase la Definición 12.8) en la misión o sus componentes, o entre la misión y el equipamiento que llevará el vehículo (véase la Parte I).
   Dependiendo de la arquitectura usada, las técnicas de validación serán diferentes —v. g. para las RdP se dispone de un formalismo que permite verificar la corrección de la red (véase el Apéndice C), para los planes de la misión se utiliza XML y esquemas XML (XSD) para su validación —la cual puede subdividirse en varias fases (véase la Sección E.4). Además, el proceso de validación a nivel de la misión incluirá todo tipo de chequeos semánticos que no estén cubiertos por las herramientas de validación base —v. g. validación XML.

3. **Envío:** Tras las aplicación de las dos fases anteriores la misión estará lista para su ejecución. Sin embargo, hasta ahora se ha trabajado desde un entorno computacional configurado para la definición y planificación de misiones, y es necesario disponer de la misión empaquetada para enviarla al vehículo que la ejecutará. Se incluye, por tanto, el envío como una fase más del ciclo de vida de la misión, en la que el planificador se comunicará con el sistema del vehículo para transmitirle la misión; el vehículo recibirá la misión y ésta quedará almacenada en el equipo informático embebido.
   El proceso de comunicación que aquí se produce dependerá del planificador y el sistema del vehículo, por lo que variará de unas arquitecturas a otras y también dependerá del proceso que se haya preferido —v. g. comunicación TCP/IP, serie, etc. También es posible hacer uso de medios de almacenamiento extraíbles, en cuyo caso realmente esta fase no tendría lugar, ya que el proceso se realizaría manualmente.

4. **Ejecución:** Con la misión ya almacenada en el vehículo, cuando se indique que debe iniciarse la misión se iniciará la fase de ejecución. El sistema del vehículo realizará lo especificado en la misión; esta fase se puede dividir en:

   a) **Carga:** Cuando se arranque el sistema del vehículo éste cargará en memoria la misión almacenada en disco. La misión se representará en memoria mediante un modelo de datos de la misma. En función de la representación textual

[9] El ciclo de vida de la arquitectura de planes de misión consta de todas las fases que nos podemos encontrar en el resto de arquitecturas; si procede, si indicará si una determinada fase es innecesaria u opcional para alguna arquitectura de especificación de misiones.

de la misión, esta fase se realizará de forma diferente —v. g. las RdP suele codificarse en lenguaje Lisp y son interpretadas directamente, los planes de la misión están definidos en XML y se tienen que analizar con las herramientas apropiadas (véase la Sección E.7) al mismo tiempo que se va contruyendo la representación del modelo de datos de la misión en memoria, para facilitar la posterior interpretación y gestión. La arquitectura del sistema también afectará en este aspecto —v. g. el sistema que trabaje con RdP dispondrá de un intérprete de RdP, para las arquitecturas basadas en comportamientos se tendrá un coordinador, para los planes de la misión se emplea el sistema *Sick***AUV** —que emplea un modelo de datos orientado a objetos— (véase la Sección 14.1.5, Sección 14.1.7 y Capítulo 15, respectivamente).

b) **Interpretación:** Una vez cargada la misión el sistema trabajará con la misma. En primer lugar se configurará el sistema para que se realice lo especificado en la misión y se irá interpretando la misión para realizar lo indicado. En el caso de producirse la reconfiguración o aprendizaje sobre la misión, los cambios que surjan en ésta —que está siendo simultáneamente interpretada— serán detectados y el sistema se autoconfigurará consecuentemente.
Si se produce cualquier tipo de incompatibilidad durante la ejecución de la misión, ésta se registrará y el sistema actuará en consecuencia. Este tipo de incompatibilidades son no anticipables, ya que no pudieron predecirse en la fase de validación (véase la Sección 12.3.1).

c) **Finalización:** Cuando se haya realizado todo lo indicado en la especificación de la misión, ésta finalizará. Se considera la fase de finalización como el proceso de desconfiguración o configuración del sistema para que éste quede en reposo; i. e. se liberarán todos los recursos que estaban usándose para la misión —v. g. la misión será descargada de la memoria (si se disponía de su modelo de datos en memoria), se apagarán los sensores y actuadores (véase la Parte I), etc.
Si durante la interpretación de la misión se ha producido reconfiguración o aprendizaje, se asume que el sistema se habrá encargado de registrar los cambios.

5. **Reproducción/Simulación:** Opcionalmente, tras finalizar la ejecución de la misión, es posible disponer de una representación del registro de todo lo realizado durante la misión para poder reproducir la misión con herramientas orientadas a tal fin. Con un simulador (véase la Apéndice A) es posible cargar toda la información generada por el sistema del vehículo durante la ejecución de la misión[10] y reproducir o recrear lo ocurrido. Esto permite el análsis *offline* de la misión realizada.
El simulador modelará un entorno y vehículos virtuales. De modo que se podrá configurar un vehículo virtual con un equipamiento igual que el usado en el real, para que las condiciones sean teóricamente equivalentes. También podrá aprovecharse el simulador para simular determinados elementos del entorno o el vehículo —v. g.

[10] Durante la ejecución de la misión se registrarán los hitos de realización de la misión y sus componentes —en base a una especificación (véase el Sección 13.8). También se habrán almacenado muestras sensoriales o valores de variables del sistema, según lo especificado en la misión (véase la Sección 13.3). Toda esta información será "grabada" al ejecutar la misión y luego podrá "reproducirse" con herramientas de simulación (véase el simulador comentado en la Apéndice A).

parámetros físico-químicos del entorno, como la temperatura del agua, virtualización de sensores, etc. En cualquiera de los casos, el sistema que se usará debe ser el mismo que el del vehículo real, para que la reproducción de la misión sea correcta. Para ello el sistema debe dar soporte a la virtualización de su hardware mediante el simulador (véase la Apéndice A). Esto se conseguirá mediante una HAL que permita conmutar fácilmente entre equipamiento real y virtualizado.

12.3.1. Incompatibilidades para la realización de la misión

Aunque en general los componentes de la especificación de misiones de las diversas arquitecturas estudiadas en la Sección 12.1 son bastante modulares, hay casos en los que puede haber dependencias. Por ello se necesita de un proceso de validación de la misión antes de que sea interpretada por el AUV (véase la Definición 12.7 y Sección 12.3). Este proceso se encarga de detectar inconsistencias en la especificación de la misión tanto a nivel de sus módulos como entre ellos. Es en este segundo caso en el que podrán detectarse incompatibilidades entre módulos. No obstante, también es posible que se produzcan incompatibilidades durante la ejecución de la misión. Por este motivo distinguimos entre dos tipos de incompatibilidades en la especificación de la misión:

Anticipables Se trata de incompatibilidades que son detectables analizando las especificaciones de las misiones, v. g. incompatibilidades entre los planes de la arquitectura propuesta (véase el Ejemplo 12.1 y Ejemplo 12.2). Esta tarea se realiza mediante un motor de validación integrado en el planificador (véase la Apéndice A).

No anticipables Se trata de incompatibilidades que no son detectables con el proceso de validación de la misión (véase la Definición 12.7). Este tipo de incompatibilidades surge durante la ejecución de la misión (véase el Ejemplo 12.3). El sistema embebido en el vehículo es el encargado de actuar en consecuencia, v. g. replanificar, priorizar unas tareas o planes frente a otros, lanzar una excepción, etc.

Los ejemplos de incompatibilidades (Ejemplo 12.1, Ejemplo 12.2 y Ejemplo 12.3) se basan en la arquitectura de especificación de misiones basada en planes de misión porque éstos son semánticamente fáciles de entender y muy representativos de las posibles incompatibilidades que pueden surgir entre ellos.

Definición 12.8 (Incompatibilidad). *Impedimento para realizar una determinada tarea especificada en la misión. Suele deberse a dependencias entre distintos elementos de la misión —v. g. planes, tareas, en el caso de la arquitectura propuesta—, pero también puede estar causada por impedimentos relativos al equipamiento disponible o al estado del sistema embebido en el vehículo que ejecuta la misión.*

Ejemplo 12.1 (Dependencia de comunicación y equipamiento). *A lo largo del Sección 8.3 se analizan los diferentes dispositivos de comunicación que pueden ir embarcados en un vehículo submarino. Cada uno dispone de unas características y entorno de aplicación diferentes, v. g. no todos los dispositivos pueden comunicarse sumergidos, algunos dispositivos dependen de factores externos para estar en línea (v. g. satélites a la vista, encontrarse en la zona de cobertura según el alcance de la señal de comunicación), etc.*
En el caso de una arquitectura basada en planes, si en el PdC se especifica una tarea para

comunicarse cuando se supere la profundidad de 100m*, pero el equipamiento de comunicaciones del vehículo no incluye ningún dispositivo capaz de comunicarse sumergido, queda de manifiesto la imposibilidad de realizar la tarea.*
Esta incompatibilidad es anticipable y, por tanto, detectable durante el proceso de validación de la misión al analizar la compatibilidad de la misión con el equipamiento definido para el vehículo que la ejecutará.

Ejemplo 12.2 (Incompatibilidad entre PdC y PdN). *Partiendo de un AUV equipado con un dispositivo de comunicación que no pueda operar sumergido —v. g. comunicador por satélite— nos podemos encontrar con una especificación del PdC y PdN incompatibles entre sí —en el caso de una arquitectura basada en planes.*
Si en el PdN se especifica el seguimiento de una ruta que obliga al AUV a ir sumergido a 100m *durante* 10km*, y al mismo tiempo en el PdC se especifica una tarea para comunicarse periódicamente —v. g. enviar todas las muestras tomadas de una determinada medida— tras recorrer* 1km*, se puede concluir que el PdC no podrá realizarse debido a la especificación impuesta en el PdN; tendría que incumplirse la especificación del PdN para poder realizar lo indicado en el PdC —v. g. si éste tuviera prioridad sobre el otro.*
Esta incompatibilidad es anticipable y, por tanto, detectable durante el proceso de validación de la misión. Así se evita que el AUV ejecute una misión que a priori *contiene tares no realizables por ser incompatibles entre ellas.*

Ejemplo 12.3 (Incompatibilidad no anticipable). *Una misión validada satisfactoriamente no tiene ningún tipo de incompatibilidad* a priori *—i. e. incompatibilidad anticipable. Por tanto, el sistema del vehículo ejecutará la misión con garantías. No obstante, durante la ejecución de la misma pueden surgir incompatibilidades no anticipables, debidas a factores ajenos a la propia especificación de la misión y el equipamiento.*
En el caso de una arquitectura basada en planes, si durante el seguimiento de una ruta que obliga al AUV a ir sumergido a 100m *y salir a superficie cada* 30min*, éste se encuentra con un corriente marina en contra que le hace tardar un* 25 % *más en salir a superficie, y al mismo tiempo en el PdC se especifica una tarea para comunicarse periódicamente cada* 30min*, el sistema detectará una incompatibilidad entre la ejecución de estos dos planes, ya que no podrá comunicarse estando aún sumergido.*
El sistema embebido en el vehículo deberá determinar la acción a realizar en este caso, v. g. continuar ejecutando el PdN —en cuyo caso no se cumpliría el PdC—, priorizar el PdC —en cuyo caso se interrumpiría el PdN. En cualquiera de los casos se produce una incompatibilidad entre los planes de la misión, la cual no es anticipable —i. e. detectable durante la validación de la misión.

Capítulo 13

Sintaxis de especificación formal de misiones

Vehicle Primitives and Mission Procedures can be developed and implemented using [...] specially developed software programming environments... (and can be built) on the theory of Petri nets...

— Paulo J. C. Ramalho Oliveira et al.
1996 (Assistant Professor (IST))

XML es un Lenguaje de Etiquetado Extensible muy simple, pero estricto que juega un papel fundamental en el intercambio de una gran variedad de datos.

— W3C
1994-2005 (World Wide Web Consortium)

Conforme a la arquitectura de especificación de misiones basada en planes, explicada en la Sección 12.2, se ha desarrollado un Lenguaje de Dominio Específico (DSL) basado en XML (véase el Apéndice B y Sección E.3) que soporta la definición basada en planes y satisface los criterios de calidad de la arquitectura en la medida apuntada en el análisis de los mismos. La sintaxis del lenguaje XML desarrollado para la especificación de los planes de la misión se materializa en la definición de una serie de esquemas XML (XSD). Los XSD definirán la estructura que puede usarse en la especificación de la misión con los ficheros XML, lo cual constituye el lenguaje adoptado.

Respecto a la elección del DSL usado para la especificación de la misión, podría haberse empleado otro lenguaje como base —en lugar de XML. Sin embargo, XML es el único que garantiza una portabilidad y disponibilidad de herramientas para su

gestión completa (véase la Sección E.7). Una posible alternativa a XML sería Prolog, que destacaría por los siguientes aspectos:

1. Realizar un intérprete de la misión sería simple, aunque habría que desarrollarlo.
2. Es posible formular autómatas de estados, útiles en la validación e interpretación de la misión.
3. Es un lenguaje de prototipado rápido, lo que reduce el tiempo de desarrollo, en el caso de usarse también para la implementación del sistema que interpretará las misiones.
4. Permite la programación basada en restricciones mediante el análisis de satisfacción de restricciones; el algoritmo símplex suele resultar útil en este paradigma de programación. Este aspecto se aprovecharía en la implementación de los disparadores de las tareas (véase la Definición 13.1).

Sin embargo, el caso de Prolog tiene un coste de diseño mayor que XML y no cumple con las necesidades de portabilidad y software disponible demandadas, aparte de otras ventajas que presenta XML en general y que se detallen en la Sección E.7. Además, el lenguaje de programación empleado en la implementación del sistema podrá ser diferente del usado en la especificación de la misión —v. g. C++ en *Sick***AUV**.

La misión se divide en varios planes y esto se verá reflejado en la especificación formal de la misma, así como en los ficheros que formarán una misión completa. Opcionalmente, la misión también podrá ir acompañada de la especificación de los parámetros de la misión (véase la Sección 13.9). En las secciones de este capítulo se detalla la sintaxis de especificación de cada uno de los planes de la misión (PdA, PdC, PdM, PdN y PdS), pero como la arquitectura de planes de misión es fundamentalmente una arquitectura basada en tareas (véase la Sección 12.1.2), todos los planes de la misión, salvo el PdN, pueden aprovechar una especificación común de las tareas, que se detalla en la Sección 13.2. En el Cuadro 13.1 se muestra la estructura de ficheros que contendrá la definición de la misión con toda su subdivisión en planes y los XSD. Se incluyen los XSD porque permitirán la validación de la misión en todo momento (véase la Definición 12.7 y Sección 12.3).

La especificación de la misión también incluye un plan de *log* (PdL) o registro del sistema, que se comenta en la Sección 13.8. Este plan se especifica de acuerdo con [Gülkü, 2002], en lugar de usar esquemas XML de desarrollo propio —como en el resto de planes de la misión—, porque el sistema usará el *framework log4j*, el cual dispone de todo lo necesario para un registro del sistema completo y sencillo (véase el Sección E.8 para más detalles sobre *log4j*). Aunque el PdL se incluye dentro de la misión, hay que señalar que se trata de una especificación orientada a los desarrolladores o programadores del sistema, y no a los usuarios del **planificador** de misiones —habitualmente oceanógrafos.

En las secciones siguientes se comenta la sintaxis adoptada siguiendo la metodología de especificación de esquemas XML y aprovechando su potencialidad para representar la arquitectura de especificación basada en planes de la forma más fiel posible a los criterios de evaluación que debe satisfacer [Bray et al., 2004, Fallside and Walmsley, 2004, Gayo, 2006, Costello, 2004]. Los diferentes elementos desarrollados se ilustran con ejemplos de especificación en XML, basados en los esquemas XML subyacentes según la correspondencia del Cuadro 13.2, cuya definición se relega al Sección E.13, donde se entra en detalles más técnicos mostrando la sintaxis XSD empleada para representar el formato

Directorio	Formato	Descripción
./esquemas	XSD	Esquemas XML que definen la sintaxis de los ficheros XML de especificación de la misión, sus planes y los parámetros
./esquemas/lib	XSD	Esquemas XML que definen la sintaxis de aspectos auxiliares que son comunes a todos los elementos de especificación de la misión
./misiones	XML (misión)	Misiones, donde cada una incluirá mediante referencias los planes y, opcionalmente, parámetros que la constituyen
./planes		Contiene los directorios con los diferentes planes de la misión
./planes/pda	XML (PdA)	Planes de almacenamiento
./planes/pdc	XML (PdC)	Planes de comunicación
./planes/pdm	XML (PdM)	Planes de medición
./planes/pdc	XML (PdC)	Planes de comunicación
./planes/pdn	XML (PdN)	Planes de navegación
./planes/pds	XML (PdS)	Planes de supervisión
./planes/pdl	XML (*log4j*)	Ficheros de configuración del registro del sistema, usando *log4j*
./tablasparametros	XML (parámetros)	Tablas de parámetros de la misión, a modo de variables de estado de la misma y que permiten su modificación dinámicamente

Cuadro 13.1: Estructura de ficheros o árbol de directorios que contiene la misión: planes, parámetros y esquemas XML (XSD)

Elemento	Esquema XML
Misión	**./esquemas/mision.xsd**
PdA	**./esquemas/pda.xsd**
PdC	**./esquemas/pdc.xsd**
PdM	**./esquemas/pdm.xsd**
PdN	**./esquemas/pdn.xsd**
PdS	**./esquemas/pds.xsd**
Acciones (Tareas)	En el esquema del plan al que pertenecen las acciones
Disparadores (Tareas)	**./esquemas/lib/disparadores.xsd**
Parámetros	**./esquemas/tablaparametros.xsd**

Cuadro 13.2: Esquemas XML (XSD) de los distintos elementos especificables en la misión

y las restricciones deseadas (véase la Sección E.5). Cabe hacer las siguientes aclaraciones generales:

1. La sintaxis de especificación de misiones aquí comentada no da soporte a la coordinación entre AUVs. No obstante, sirve de base para la comunicación que facilitaría la coordinación, mediante la especificación del PdC, al cual sólo sería necesario incorporar algunas acciones adicionales (véase el Capítulo 18).

2. Durante la exposición de la sintaxis se mostrarán y explicarán los elementos y atributos XML con los que se especifican los diferentes componentes de la misión (véase la Definición E.1 y Definición E.2, respectivamente). Los nombres de éstos se muestran en **negrita** y sin usar caracteres ASCII especiales —v. g. á, ñ— porque el léxico de los atributos y elementos XML no lo permite.

3. Se usa la versión **1.0** de XML y XSD porque es la versión final actualmente publicada por el W3C, el cual está trabajando en la elaboración de la especificación **1.1**, pero aún es un borrador (véase el Sección E.3).

4. Las listas de elementos XML que formen parte de los planes dispondrán de un atributo XML que permita la identificación unívoca de cada elemento —i. e. atributo **id** que se usa como identificador. La utilidad de este identificador radica en que facilita las búsquedas —especialmente si se usa una tabla *hash* en el modelo de datos (véase el Capítulo 15)— y permite la eliminación e inserción de elementos fácilmente, sin afectar al resto de la lista. Cuando en una lista no se usen identificadores, se usará el orden de aparición de los elementos en el fichero XML.

5. Determinada información especificable en los planes de la misión podrá ser opcional. El sistema debe encargarse de codificar la opcionalidad y proporcionar valores por defecto si procede, en el modelo de datos usado para la interpretación de la misión. Según sean opcionales los atributos o los elementos XML, esto tendrá connotaciones o problemas diferentes:

   Atributos XML Los atributos XML se modelarán mediante variables de tipos de datos primitivos del lenguaje de programación del sistema, por lo que bastará con asignarle el valor por defecto o un valor especial que indique que no se especificó ningún valor. Véase el caso de la acción **medir**, cuyo atributo **muestras** es opcional (véase la Sección 13.5), de modo que si no se indica se tomará un número indefinido de muestras, lo cual se podrá reflejar en el modelo de datos con un valor especial —v. g. -1— o con una variable adicional que indique si el número de muestras está limitado o no.

   Elementos XML Los elementos XML suelen modelarse mediante clases en el sistema, suponiendo Programación Orientada a Objetos (POO). Cuando un elemento es opcional, el sistema se encargará de crear una clase con valores por defecto de acuerdo con la gestión que se haga de cada plan de la misión. Véase el caso de la configuración de los sensores especificados en el PdM (véase la Sección 13.5) y los propios sensores, que pueden ser opcionales, en cuyo caso el sistema deberá poder gestionar sensores sin configuración —asignándoles una por defecto– y listas sin sensores para las acciones **medir**.

   En el caso de que haya algún elemento XML con identificador (**id**), que sea opcional, se usará un valor especial —v. g. -1— para hacer referencia a la representación interna del sistema.

6. Se recoge la posibilidad de que el sistema que ejecute la misión tenga capacidades de Análisis Dimensional, i. e. que realice cálculos numéricos considerando las unidades de las medidas involucradas en éstos. Por esta razón, la especificación de la misión siempre recoge las unidades de los valores que el **planificador** toma durante la fase de **creación** de la misión (véase la Sección 12.3).

7. En determinados atributos se usa como tipo de datos una simple cadena de caracteres por simplicidad y generalidad, v. g. los nombres de medidas (véase la Definición 13.7) no se definen en una lista como enumerados (véase la Sección E.5), sino que se permite cualquier nombre —i. e. ristra o cadena de caracteres— y se comprobará que el equipamiento soporta dicha medida, mediante el proceso de validación de la misión.

13.1. Misión

La misión es el elemento raíz de la especificación de la misión. Se trata de un simple contenedor con referencias a los planes y parámetros de la misión (véase el Algoritmo 13.1). A la misión se le asigna un identificador y un nombre en lenguaje natural para facilitar la gestión de la misma. Los planes que forman parte de la misión son obligatorios y en principio únicos —i. e. debe indicarse un único plan de cada tipo, si bien podría eliminarse esta restricción y permitir varios planes de un mismo tipo (véase el **??**). Los parámetros de la misión son opcionales y se especificarán mediante tablas (véase la Sección 13.9). Se pueden proporcionar varias tablas de parámetros, lo que permite reaprovecharlas entre misiones. Lo mismo ocurre con los planes, que son módulos independientes de la misión que pueden reutilizarse.

```
<?xml version="1.0"?>
<mision id="1" nombre="Mision 1"
                xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                xmlns="http://www.mision.com"
                xsi:schemaLocation="http://www.mision.com ../esquemas/mision.xsd">

  <pda fichero="../planes/pda/pda.xml"/>
  <pdc fichero="../planes/pdc/pdc.xml"/>
  <pdm fichero="../planes/pdm/pdm.xml"/>
  <pdn fichero="../planes/pdn/pdn.xml"/>
  <pds fichero="../planes/pds/pds.xml"/>
  <pdl fichero="../../../log/cfg/auv.xml"/>

  <tablaParametros fichero="../tablaparametros/tablaparametros.xml"/>
</mision>
```

Algoritmo 13.1: Misión: Planes y Parámetros

Para garantizar que la misión no presenta incompatibilidades se someterá a un proceso de validación, pues pueden existir determinadas dependencias puntuales (véase la Definición 12.8, Definición 12.7 y Sección 12.3). Si la misión contiene varias tablas de parámetros, el proceso de validación de la misión comprobará que no haya parámetros redefinidos —en una misma tabla o entre tablas diferentes.[1]

Tanto cada uno de los planes como las tablas de parámetros —si las hubiere— se indican mediante la ruta relativa del fichero XML donde se definen (véase el Algoritmo 13.1). Las rutas serán relativas a la ubicación del fichero XML donde se define la misión. El planificador (véase la Apéndice A) debe conocer o compartir el árbol de directorios que se usa para almacenar la misión en el sistema del vehículo que ejecuta la misión. El campo **Directorio** del Cuadro 13.1 es representativo del árbol de directorios que se usará habitualmente[2].

La especificación de la misión, todos los planes de la misión —salvo el PdL— y las tablas de parámetros comparten los atributos XML indicados en el Cuadro 13.3 (véase el Algoritmo 13.1, Algoritmo 13.9, Algoritmo 13.11, Algoritmo 13.13, Algoritmo 13.15, Algoritmo 13.22 y Algoritmo 13.26). Se trata de datos identificativos que permiten hacer referencia a los mismos de forma unívoca, v. g. especificación de atributos a modificar en los planes de la misión, como se comenta en la Sección 13.9.

[1] En el caso de que un parámetro de la misión se haya redefinido, en lugar de avisar de la incompatibilidad se fusionarán ambas especificaciones. Esto se traduce en que el parámetro apuntará a los elementos indicados en cada una de ellas.

[2] Los diferentes ficheros que definen la misión al completo pueden estructurarse en el árbol de directorios que se prefiera. No obstante, el árbol de directorios propuesto en el Cuadro 13.1 se mantiene como compromiso o estándar para garantizar la compatibilidad entre *Sick***AUV** y los proyectos complementarios **planificador** y **simulador**.

Atributo	Descripción
id	Identificador unívoco del plan o de la misión
nombre	Nombre del plan o la misión, que suele ser representativo de la utilidad

Cuadro 13.3: Atributos comunes a todos: la misión, los planes de la misión y las tablas de parámetros

13.2. Planes de la misión. Estructura general: Tareas

Todos los planes de la misión —salvo el PdN y el registro del sistema (véase la Sección 13.6 y Sección 13.8)— comparten una estructura general basada en tareas (véase la Definición 12.3 y Sección 12.1.2). La especificación de estos planes está constituida por una lista de tareas que se gestionarán concurrentemente. Las tareas están formadas por un conjunto de disparadores y otro de acciones, de forma que cuando se cumplen todos los disparadores se ejecutan todas las acciones (véase la Sección 15.1). También dispondrán de un período de inhibición que determina cada cuánto tiempo deben comprobarse los disparadores.

Definición 13.1 (Disparador). *Especificación que determina* grosso modo *el momento o lugar para ejecutar una tarea. Se realiza un cómputo sobre una medida del sistema (véase la Definición 13.7) y en el caso de cumplirse o verificarse se dice que se* ***dispara****. Cuando se cumplen todos los disparadores de una determinada tarea se genera una notificación que provoca la activación o ejecución de todas sus acciones (véase la Sección 15.1 para conocer cómo se gestiona en* Sick***AUV****).*
Se distinguen varios tipos de disparadores: ***condición****,* ***intervalo*** *y* ***excepción*** *(véase la Definición 13.4, Definición 13.5 y Definición 13.6, respectivamente).*

Definición 13.2 (Acción). *Especificación del nombre y los parámetros de un comando soportado por el sistema, que se ejecutará bajo su supervisión y monitorización (véase la Definición 15.1, Definición 15.2 y Sección 15.1).*

Definición 13.3 (Período de Inhibición). *Especificación temporal que indica cada cuánto tiempo se comprobarán los disparadores de una tarea, contando desde el momento en que éstos se cumplan (véase la Definición 13.1).*
El chequeo o comprobación del estado de los disparadores se realizará periódicamente usando el período definido. Este proceso inhibe la comprobación con la finalidad de garantizar que las acciones se mantienen en ejecución el tiempo que se estima necesario en la misión (véase la Sección 15.1 para más detalles sobre la gestión que realiza Sick***AUV*** *de las tareas y sus disparadores).*

Cada plan de la misión que se defina mediante tareas dispondrá de una lista de tareas de **0 a N**; lo habitual será que se definan varias tareas pero también es posible que no exista ninguna tarea de un determinado tipo de plan. A toda tarea se le asignará un nombre e identificador único dentro del plan al que pertenezca. La tarea constará de los siguientes elementos (véase el Algoritmo 13.2):

Disparadores Lista de disparadores que tienen que cumplirse para que se ejecuten las acciones de la tarea (véase la Definición 13.1 y Sección 13.2.1).

Acciones Lista de acciones a ejecutar mientras se cumplan los disparadores de la tarea (véase la Definición 13.2 y Sección 13.2.2). Las acciones que puede contener la tarea dependerán del tipo de plan al que pertenezca ésta.

Período de Inhibición Indica cada cuánto tiempo se comprueban los disparadores de la tarea. Se denomina período de inhibición porque define el tiempo que queda inhibida la comprobación de los disapradores desde que éstos se cumplen o disparan —momento en el que se ejecutan todas las acciones. Esto permite que las acciones se mantengan en ejecución al menos durante el tiempo indicado en el período de inhibición, aunque los disparadores dejen de cumplirse inmediatamente (véase la Definición 13.3).

```
<tarea id="1" nombre="AlmacenamientoTemperatura">
  <disparadores:disparadores>
    <!-- ... -->
  </disparadores:disparadores>
  <acciones>
    <!-- ... -->
  </acciones>
  <periodoInhibicion valor="10" unidad="minuto" />
</tarea>

<tarea id="2" nombre="AlmacenamientoPosicion">
  <disparadores:disparadores>
    <!-- ... -->
  </disparadores:disparadores>
  <acciones>
    <!-- ... -->
  </acciones>
  <periodoInhibicion valor="30" unidad="segundo" />
</tarea>

<!-- ... -->
```

Algoritmo 13.2: Esqueleto de tareas

13.2.1. Disparadores

Todas las tareas, con independencia del tipo de plan de la misión al que pertenezcan, dispondrán de una lista de disparadores con un formato de especificación común. Una tarea podrá tener una lista de disparadores de **0 a N**. Una tarea se ejecutará cuando se cumplan todos sus disparadores, de modo que si no tiene ningún disparador se ejecutará siempre, a partir del inicio de la misión— (véase la Sección 15.1 para más detalles sobre la gestión de los disparadores en *Sick***AUV**).

Definición 13.4 (Condición (Disparador)). *Disparador que define una comparación entre el valor de las muestras de una determinada medida y un valor indicado aplicando una operación relacional. Si la comparación se cumple se produce el disparo.*

Definición 13.5 (Intervalo (Disparador)). *Disparador que define un intervalo o vector de valores indicando un inicio, incremento o período y un fin. Si el valor de las muestras de una determinada medida está contenido en el intervalo se produce el disparo.*

Definición 13.6 (Excepción (Disparador)). *Disparador que se define indicando el nombre de una excepción del sistema. Si se genera dicha excepción se producirá el disparo.*

Los tipos de disparadores soportados son:

Condición Se explica en la Definición 13.4 e ilustra con los ejemplos del Algoritmo 13.3:

1. Comprobar si la temperatura exterior es igual a 10°C.
2. Comprobar si se ha alcanzado un *waypoint* anterior (estrictamente menor) al 3.

Los operadores relacionales que pueden usarse son: $\neq$, $<$, $\leq$, $=$, $\geq$ y $>$. En la especificación XML se evita el uso de símbolos, de modo que se usa **ne**, **lt**, **le**, **eq**, **ge** y **gt**, respectivamente (véase el Capítulo 18 para más información sobre posibles operadores adicionales).

Intervalo Se explica en la Definición 13.5 e ilustra con los ejemplos del Algoritmo 13.4:

1. Intervalo formado por los instantes temporales —contados desde el momento en que se inicia la misión— que van desde el minuto 1 al 100 con un incremento de 10. Por tanto, se tendrá el siguiente vector de valores:

$$[1, 11, 21, 31, 41, 51, 61, 71, 81, 91]\text{min}$$

2. Intervalo de presiones que va desde 2 a 20 con un incremento de 4, todo ello medido en atmósferas. Por tanto, se tendrá el siguiente vector de valores:

$$[2, 6, 10, 14, 18]\text{atm}$$

La especificación del intervalo es equivalente a la definición de vectores en el lenguaje **MATLAB®**, donde se indica un valor inicial, el período o incremento y el valor final con la siguiente sintaxis: **inicio:periodo:final** —v. g. **1:10:100** para el primer ejemplo del Cuadro 13.5. No obstante, en la especificación de la misión los intervalos también pueden ser abiertos, i. e. que no se defina el **inicio** o el **final**. Esto permite especificar series numéricas infinitas, en lugar de vectores —que tienen una cantidad finita de valores. Para ello se podrá usar el valor **-INF** o **INF** en los atributos **inicio** y **final**, respectivamente, en el caso de un **periodo** positivo; otra alternativa consiste en no indicar el **inicio** o el **final**, pues serán atributos opcionales. En el Algoritmo 13.5 se muestra un ejemplo de especificación de un intervalo infinito, donde se tendrá la siguiente serie de valores:

$$25, 50, 75, 100, 125, \ldots \text{m}$$

La implementación de este tipo de intervalos puede ser compleja, especialmente si no se indica el **inicio**. En cualquier caso, la especificación recoge cualquier tipo de intervalo infinito (véase la Sección 15.1 para consultar el soporte que ofrece *Sick***AUV**para este tipo de intervalos).

Excepción Se explica en la Definición 13.6 e ilustra con el ejemplo del Algoritmo 13.6:

1. Excepción que indica que se han agotado las baterías, la cual tiene el nombre **agotamientoBaterias** (véase la Sección 15.8 para consultar la gestión de las excepciones del sistema).

Atributo	Descripción
id	Identificador unívoco del disparador dentro de la lista a la cual pertenece
medida	Nombre de la medida (véase la Definición 13.7) cuyas muestras se usarán para evaluar la condición
operador	Operador relacional utilizado en la condición
valor	Valor que determina el umbral con el que se compararán las muestras de la medida de la condición. Debe pertenecer al dominio de la medida —i. e. tipado, rango de valores, unidad, etc.
unidad	Unidad con la que se expresa el **valor** de la condición

Cuadro 13.4: Atributos usados para especificar un disparador de tipo condición

La lista de excepciones del sistema formará parte de la especificación del equipamiento del vehículo que ejecutará la misión. Esto permite validar que las excepciones especificadas en la misión están soportadas por el vehículo (véase la Definición 12.7 y Sección 12.3) y ofrece al **planificador** de la misión la posibilidad de gestionar las excepciones que pueden generarse.

```
<disparadores:condicion id="1" medida="temperatura exterior"
                        operador="eq" valor="10" unidad="grado centigrado"/>
<disparadores:condicion id="2" medida="waypoint"
                        operador="lt" valor="3" unidad="waypoint"/>
```

Algoritmo 13.3: Condición (Disparador)

```
<disparadores:intervalo id="1" medida="tiempo"
                        inicio="1" fin="100" periodo="10" unidad="minuto"/>
<disparadores:intervalo id="2" medida="presion"
                        inicio="2" fin="20" periodo="4" unidad="atm"/>
```

Algoritmo 13.4: Intervalo (Disparador)

```
<disparadores:intervalo id="1" medida="distancia recorrida"
                        inicio="25" periodo="25" unidad="metro"/>
```

Algoritmo 13.5: Intervalo infinito (Disparador)

```
<disparadores:excepcion id="1" nombre="agotamientoBaterias"/>
```

Algoritmo 13.6: Excepción (Disparador)

Durante el proceso de validación de la misión se comprobará que no haya disparadores repetidos dentro de la lista de una tarea concreta (véase la Sección 12.3). En caso de que haya repiticiones, se eliminarán para dejar una sola ocurrencia del disparador. Esto libera al intérprete de los planes de la misión —integrado en el sistema— de realizar comprobaciones para optimizar la gestión de los disparadores.[3]

El sistema encargado de ejecutar la misión se autoconfigurará de acuerdo con los disparadores de las tareas de todos los planes de la misión. Esto forma parte del proceso de configuración del sistema para que pueda realizarse la misión. En lo relativo a los disparadores, esto se materializa en la activación de la toma de muestras de las medidas usadas por los disparadores (véase la Sección 15.1 para más detalles).

[3]Si hubiera una lista con disparadores repetidos la gestión de los mismos en el sistema funcionaría perfectamente, pero de forma menos eficiente.

Atributo	Descripción
id	Identificador unívoco del disparador dentro de la lista a la cual pertenece
medida	Nombre de la medida (véase la Definición 13.7) cuyas muestras se usarán para determinar si están contenidas en el vector de valores que define el intervalo
inicio	Valor inicial del intervalo; se incluye en el intervalo
fin	Valor final del intervalo; se incluye en el intervalo, siempre que no se exceda al aplicar el **periodo**
periodo	Valor del período o incremento que determina los valores del vector que define implícitamente el intervalo, i. e. desde **inicio** hasta **fin** con un incremento igual al valor del **periodo**; el **inicio** y **fin** están incluidos, si bien no se toma el **fin** si se excede
unidad	Unidad con la que se expresan el **inicio**, **fin** y **periodo** del intervalo

Cuadro 13.5: Atributos usados para especificar un disparador de tipo intervalo

Atributo	Descripción
id	Identificador unívoco del disparador dentro de la lista a la cual pertenece
nombre	Nombre de la excepción que si es generada por el sistema hará que se cumpla el disparador

Cuadro 13.6: Atributos usados para especificar un disparador de tipo excepción

13.2.1.1. Combinación lógica de disparadores

Los disparadores —sean del tipo que sean— podrán combinarse entre sí gracias a las listas de tareas dentro del plan y las listas de disparadores dentro de cada tarea. Hablamos de combinación lógica porque todos los disparadores se caracterizan por un estado lógico, i. e. se cumplen o no. Por tanto, la sintaxis de la especificación de la misión desarrollada soporta las siguientes combinaciones:

Y lógico Todos los disparadores de una lista perteneciente a una tarea se evaluarán como un **y** lógico —i. e. deben cumplirse todos los disparadores de la lista para que se ejecuten las acciones de la tarea. El Algoritmo 13.7 muestra un ejemplo de combinación con **y** lógico que en lenguaje natural puede expresarse así:

> Realizar las acciones de la tarea si:
>
> La temperatura interna es estrictamente menor de 15°F
>
> **y**
>
> El vehículo está en uno de los *waypoints* del intervalo definido por el vector $[1, 3, 5, 7, 9]$ —i. e. **inicio** en el 1, **fin** en el 10 y **periodo** de 2.

O lógico La sintaxis de la especificación de la misión permite combinar disparadores aplicando un **o** lógico mediante la definición de varias tareas con las mismas acciones, pero cada una de ellas con los distintos disparadores que se combinarán como

un **o** lógico. Esto es posible gracias al mecanismo de gestión de tareas que implementa el sistema (véase la Sección 15.1). El Algoritmo 13.8 muestra un ejemplo de combinación con **o** lógico que en lenguaje natural puede expresarse así:

> Enviar un aviso de SOS al **planificador** —acción definida en las tareas— si:
>
> La temperatura exterior es menor o igual que 2°C
> **o**
> La profundidad es mayor o igual que 100m.

```
<disparadores:disparadores>
  <disparadores:condicion id="1" medida="temperatura interna"
                          operador="lt" valor="15" unidad="grado fahrenheit"/>
  <disparadores:intervalo id="2" medida="waypoint"
                          inicio="1" fin="10" periodo="2" unidad="waypoint"/>
</disparadores:disparadores>
```

Algoritmo 13.7: Y lógico entre disparadores

```
<tarea id="1" nombre="SOSTemperaturaLimite">
  <disparadores:disparadores>
    <disparadores:condicion id="1" medida="temperatura exterior"
                            operador="le" valor="2" unidad="grado centigrado"/>
  </disparadores:disparadores>
  <acciones>
    <enviarMedida id="1" medida="SOS" destinatario="planificador"/>
  </acciones>
  <periodoInhibicion valor="10" unidad="segundo"/>
</tarea>

<tarea id="2" nombre="SOSPresionLimite">
  <disparadores:disparadores>
    <disparadores:condicion id="1" medida="profundidad"
                            operador="ge" valor="100" unidad="metro"/>
  </disparadores:disparadores>
  <acciones>
    <enviarMedida id="1" medida="SOS" destinatario="planificador"/>
  </acciones>
  <periodoInhibicion valor="20" unidad="segundo"/>
</tarea>
```

Algoritmo 13.8: O lógico entre disparadores

13.2.2. Acciones

El tipo de acciones que puede contener una tarea dependerá del plan al que ésta pertenece. La especificación de cada plan de la misión tendrá, por tanto, su propia tipología de acciones, que se muestra al presentar cada uno de los planes de la misión. Una tarea podrá tener una lista de acciones de **0 a N**.[4]

Aunque las acciones serán diferentes según el plan al que pertenezcan, todas comparten las siguientes características:

1. Identificador único para cada tarea dentro de la tarea a la que pertenecen.
2. Las acciones tendrán un nombre identificativo del comando que la implementa o el servicio que se ofrece en el sistema (véase la Definición 15.1 y Definición 15.2).

[4]Se permite no especificar ninguna acción dentro de una tarea por completitud en la definición y porque puede resultar útil en las pruebas de los diferentes módulos del sistema (véase la Sección 15.1).

Atributo	Descripción
id	Identificador unívoco de la acción dentro de la lista a la cual pertenece
medida	Nombre de la medida cuyas muestras serán almacenadas. El nombre del dato coincidirá con el de la medida, pero haciendo referencia al inventario de datos almacenados en disco (véase la Sección 15.3)
muestras	Indica el número de muestras que deben almacenarse de la medida. Con el valor **INF** se indica que se tome un número ilimitado de muestras. El atributo **muestras** es opcional y su valor por defecto es **INF**

Cuadro 13.7: Atributos usados para especificar la acción **almacenar**

Según el tipo de acción se dispondrá de un determinado número y tipo de parámetros *ad hoc* para configurar la acción o proporcionarle los datos que ésta necesita para su correcta ejecución.

13.3. Plan de Almacenamiento

El PdA es un plan que se especifica en base a tareas. Su estructura básica es la mostrada en la Sección 13.2 y sus tareas sólo admiten acciones relacionadas con el almacenamiento de datos en el sistema. El Algoritmo 13.9 muestra el esqueleto del PdA, donde las acciones de almacenamiento soportadas por sus tareas son:

almacenar Especifica el nombre de la medida cuyas muestras se almacenarán. Opcionalmente se puede indicar el número máximo de muestras a almacenar; en caso contrario, se almacenarán todas las muestras, de forma ilimitada. En el Cuadro 13.7 se describen los atributos XML que definen una acción de tipo **almacenar**.

```
1 <planDeAlmacenamiento id="1" nombre="pda"
2                       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3                       xmlns:disparadores="http://www.disparadores.com"
4                       xmlns="http://www.pda.com"
5                       xsi:schemaLocation="http://www.pda.com ../../esquemas/pda.xsd">
6   <!-- ... -->
7 </planDeAlmacenamiento>
```

Algoritmo 13.9: Esqueleto del PdA

De acuerdo con el criterio de modularidad que deben satisfacer los planes de la misión (véase la Definición F.1 y Sección 12.1), las acciones que ejecute este plan no forzarán la realización de acciones que se especifican en otros planes —v. g. la acción **almacenar** no forzará a **medir**, que es una acción propia del PdM (véase la Sección 13.5). Tampoco se encargará de realizar acciones que no son responsabilidad suya. En este sentido, el PdA se encarga exclusivamente del almacenamiento de datos que genera el sistema como parte de la ejecución de la misión, de forma que no almacenará información de depuración o estado del sistema, lo cual es responsabilidad del registro del sistema (véase la Sección 13.8).

Los datos almacenados serán registrados en un inventario (véase la Sección 15.3.2) usando el mismo nombre que la medida especificada en las acciones de **almacenar**; los datos no son más que un conjunto de muestras de una medida (véase la Definición 13.7 y Definición 13.8 para más detalles sobre las diferencias entre medida y dato), que suelen

almacenarse en un fichero en disco y son gestionados mediante el inventario. Las muestras de las medidas son generadas normalmente por un subsistema sensorial que gestiona el equipamiento sensorial del vehículo (véase la Sección 15.7 y Sección 8.1). Para poder almacenar cualquier tipo de información no sensorial —v. g. variables u observables del sistema, como las estimaciones de la posición y velocidad, etc.— el sistema debe disponer de sensores virtuales que proporcionen esta información como si de una medida se tratase. Al modelar o representar la información mediante muestras se consigue homogeneizar su tratamiento en el sistema. La lista de medidas disponibles vendrá especificada en el equipamiento del vehículo (véase el Capítulo 9), lo que permitirá validar que el plan no presenta incompatibilidades (véase la Definición 12.7).

El Algoritmo 13.10 muestra los siguientes ejemplos de tareas con acciones de almacenamiento:

1. La tarea 1 se encarga del almacenamiento de la **temperatura exterior** y la **profundidad**, lo cual se especifica respectivamente en las acciones 1 y 2 de esta tarea. Para ambas acciones el número de muestras que se tomará de estas medidas no está limitado.

2. La tarea 2 se encarga del almacenamiento de la **salinidad**, tomando un máximo de 500 muestras; se tomarán menos muestras si dejan de cumplirse los disparadores de la tarea antes de alcanzar el número de muestras máximo especificado.

```
<tarea id="1" nombre="almacenarTemperatura">
  <disparadores:disparadores>
    <!-- ... -->
  </disparadores:disparadores>
  <acciones>
    <almacenar id="1" medida="temperatura exterior"/>
    <almacenar id="2" medida="profundidad"/>
  </acciones>
  <periodoInhibicion valor="5" unidad="minuto"/>
</tarea>

<tarea id="2" nombre="almacenarSalinidad">
  <disparadores:disparadores>
    <!-- ... -->
  </disparadores:disparadores>
  <acciones>
    <almacenar id="1" medida="salinidad" muestras="500"/>
  </acciones>
  <periodoInhibicion valor="1" unidad="hora"/>
</tarea>
```

Algoritmo 13.10: Acciones del PdA

13.4. Plan de Comunicación

El PdC es un plan que se especifica en base a tareas. Su estructura básica es la mostrada en la Sección 13.2 y sus tareas sólo admiten acciones relacionadas con la comunicación de datos (véase la Sección 11.5 para más detalles sobre la tipología de misiones de comunicación). El Algoritmo 13.11 muestra el esqueleto del PdC, donde las acciones de comunicación soportadas por sus tareas son:

enviarMedida Especifica el nombre de la medida cuyas muestras se enviarán al destinatario indicado. En el Cuadro 13.8 se describen los atributos XML que definen la acción **enviarMedida**.

Atributo	Descripción
id	Identificador unívoco de la acción dentro de la lista a la cual pertenece
medida	Nombre de la medida cuyas muestras serán enviadas
destinatario	Dirección que identifica el participante del proceso comunicativo que recibirá las muestras enviadas por el vehículo. Este atributo es opcional, de modo que si no se especifica se interpretará que el destinatario son todas las entidades de la red —i. e. se hará *broadcast* o multidifusión

Cuadro 13.8: Atributos usados para especificar la acción **enviarMedida**

enviarDato Especifica el nombre del dato que se enviará al destinatario indicado. El inventario de datos proporcionará el fichero almacenado en disco con las muestras de la medida de igual nombre que el dato. En el Cuadro 13.9 se describen los atributos XML que definen la acción **enviarDato**.

recibirMedida Especifica el nombre de la medida cuyas muestras se recibirán de la fuente u origen indicado. Por lo general, estas muestras se usarán para el control remoto de la navegación —v. g. velocidad, ángulo de giro— (véase la Sección 15.4). En el Cuadro 13.10 se describen los atributos XML que definen la acción **recibirMedida**.

recibirDato Especifica el nombre del dato que se recibirá y la fuente u origen que lo enviará al vehículo. El dato se almacenará en el inventario una vez recibido (véase la Sección 15.3.2). En el Cuadro 13.11 se describen los atributos XML que definen la acción **recibirDato**.

```
1 <planDeComunicacion id="1" nombre="PdC"
2                     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3                     xmlns:disparadores="http://www.disparadores.com"
4                     xmlns="http://www.pdc.com"
5                     xsi:schemaLocation="http://www.pdc.com ../../esquemas/pdc.xsd">
6   <!-- ... -->
7 </planDeComunicacion>
```

Algoritmo 13.11: Esqueleto del PdC

Definición 13.7 (Medida). *Elemento del sistema del cual se obtienen muestras (véase la Definición 15.3). Se trata de una unidad de información atómica —v. g. temperatura exterior, posición estimada, profundidad, etc.*
El subsistema sensorial se encarga de obtener las muestras mediante el sensor correspondiente (véase la Sección 15.7) —incluyendo los sensores virtuales.

Definición 13.8 (Dato). *Elemento del sistema almacenado en disco y registrado en el inventario (véase la Sección 15.3.2), el cual hace transparente la asociación entre el dato y el fichero que contiene su información. El nombre del dato coincide con el de la medida cuyas muestras almacena.*

A pesar del paralelismo entre las acciones de envío (**enviarMedida** y **enviarDato**), las de recepción (**recibirMedida** y **recibirDato**), las que manejan medidas

Atributo	Descripción
id	Identificador unívoco de la acción dentro de la lista a la cual pertenece
dato	Nombre del dato del inventario que será enviado
destinatario	Dirección que identifica el participante del proceso comunicativo que recibirá las muestras enviadas por el vehículo. Este atributo es opcional, de modo que si no se especifica se interpretará que el destinatario son todas las entidades de la red —i. e. se hará *broadcast* o multidifusión

Cuadro 13.9: Atributos usados para especificar la acción **enviarDato**

Atributo	Descripción
id	Identificador unívoco de la acción dentro de la lista a la cual pertenece
medida	Nombre de la medida cuyas muestras serán recibidas
fuente	Dirección que identifica el participante del proceso comunicativo que enviará las muestras recibidas por el vehículo. Este atributo es opcional, de modo que si no se especifica se interpretará que la fuente o remitente es anónimo, en cuyo caso la información se recibirá de cualquiera

Cuadro 13.10: Atributos usados para especificar la acción **recibirMedida**

Atributo	Descripción
id	Identificador unívoco de la acción dentro de la lista a la cual pertenece
dato	Nombre del dato que será recibido y almacenado en el inventario
fuente	Dirección que identifica el participante del proceso comunicativo que enviará las muestras recibidas por el vehículo. Este atributo es opcional, de modo que si no se especifica se interpretará que la fuente o remitente es anónimo, en cuyo caso la información se recibirá de cualquiera

Cuadro 13.11: Atributos usados para especificar la acción **recibirDato**

(**enviarMedida** y **recibirMedida**) y las que manejan datos (**enviarDato** y **recibirDato**), cada una se modela con una acción independiente para recoger las pequeñas diferencias entre ellas. En el caso de las acciones de recepción, el sistema seguirá la estrategia de recepción activa (véase la Sección 11.5.2) —i. e. se realizará la petición[5] de la información a recibir—, pues se trata de la más robusta —frente a la recepción pasiva. Si no se especifica ninguna acción en alguna tarea del PdC, el sistema lo interpretará como la necesidad de ponerse en línea (véase la Sección 11.5.2); no obstante, se podría indicar con una acción específica para ello (véase el Capítulo 18 para más detalles sobre posibles extensiones al conjunto de acciones soportadas), lo cual sería más apropiado al facilitar un tratamiento más homogéneo de los planes.

Tanto el destinatario en los envíos, como la fuente en las recepciones, se codificarán como una dirección que identifique unívocamente a una entidad dentro de la red a la

[5] En el caso de que la fuente sea anónima, la petición se hará mediante *broadcast* o multidifusión.

cual se conectará el vehículo que ejecute la misión. Se propone la siguiente sintaxis para su especificación (véase la Sección E.13.1 para consultar la especificación formal con expresiones regulares y la gramática BNF):

1. Se puede indicar la dirección IP y opcionalmente el puerto[6], separado por dos puntos (**:**) —v. g. **192.168.1.33**, **192.168.1.34:21**.

2. Se puede indicar un nombre de dominio que se resolverá por el DNS de la red o mediante la lista de nombres del sistema —v. g. fichero del sistema UNIX */etc/hosts*. Los nombres serán representativos del rol comunicativo de cada entidad (véase la Definición 11.9, Sección 11.5.1 y Apéndice A) y opcionalmente también podrá indicarse el puerto, separado por dos puntos (**:**) —v. g. **planificador**, **simulador**, **AUV1:23**, etc.

3. Tanto la especificación mediante direcciones IP —con o sin puerto—, como con nombres de dominios podrán combinarse, pudiendo especificarse una lista de direcciones separadas por coma (**,**) —v. g. **planificador, 192.168.1.35:21**.

El Algoritmo 13.12 muestra los siguientes ejemplos de tareas con acciones de comunicación:

1. La tarea 1 se encarga de la comunicación de las medidas de **posicion** y **velocidad**. Cuando se cumplan los disparadores de la tarea se enviarán las muestras de la **posicion** al **planificador** y al mismo tiempo se recibirán las muestras de la **velocidad** —a la que se desea que se desplace el vehículo— del **planificador**.

2. La tarea 2 se encarga de la comunicación de los datos de **temperatura** y **batimetria**. Cuando se cumplan los disparadores de la tarea se hará, por orden de **id**:

   a) El envío del estado de la misión —i. e. muestras de la medida **estadoMision**— al **planificador**.

   b) El envío de todas las muestras almacenadas en el dato **temperatura** a la dirección IP **192.168.1.33**.

   c) La recepción de la batimetría —i. e. la información batimétrica se almacenará como el dato **batimetria**—, que la enviará el **servidorBatimetria**.

```
<tarea id="1" nombre="comunicarPosicionVelocidad">
  <disparadores:disparadores>
    <!-- ... -->
  </disparadores:disparadores>
  <acciones>
    <enviarMedida id="1" medida="posicion" destinatario="planificador"/>
    <recibirMedida id="2" medida="velocidad" fuente="planificador"/>
  </acciones>
  <periodoInhibicion valor="3" unidad="minuto"/>
</tarea>

<tarea id="2" nombre="comunicarTemperaturaBatimetria">
  <disparadores:disparadores>
    <!-- ... -->
  </disparadores:disparadores>
  <acciones>
    <enviarMedida id="1" medida="estadoMision" destinatario="planificador"/>
```

[6] Los puertos se indicarán en forma numérica, pero también podrá incluirse una traducción de los nombres de los puertos más conocidos —v. g. **ftp** = **21**— (véase la **??**).

```
18      <enviarDato id="2" dato="temperatura" destinatario="192.168.1.33"/>
19      <recibirDato id="3" dato="batimetria" fuente="servidorBatimetria"/>
20    </acciones>
21    <periodoInhibicion valor="1" unidad="hora"/>
22  </tarea>
```

Algoritmo 13.12: Acciones del PdC

13.5. Plan de Medición

El PdM es un plan que se especifica en base a tareas. Su estructura básica es la mostrada en la Sección 13.2 y sus tareas sólo admiten acciones relacionadas con la medición de muestras de medidas (véase la Sección 11.1 para más información sobre las misiones de toma de muestras). El Algoritmo 13.13 muestra el esqueleto del PdM, donde las acciones de comunicación soportadas por sus tareas son:

medir Especifica el nombre de la medida de la que tomar muestras e indica qué frecuencia y resolución de muestreo usar. De acuerdo con lo comentado en la Sección 11.1.1, existen diversas posibilidades a la hora de especificar la toma de muestras, respecto a la selección o elección del sensor que se usará:

1. Indicar sólo la medida y el sistema seleccionará el sensor de forma transparente, en base a las características de la medición especificada —i. e. frecuencia y resolución de muestreo, fundamentalmente— y la disponibilidad y estado del equipamiento sensorial.
2. Indicar el sensor con el que tomar las muestras de la medida. Se podrá indicar un solo sensor o varios. En este último caso, el sistema elegirá uno de los sensores dentro de la lista especificada.
3. En el caso de que se especifique uno o varios sensores, para cada uno de ellos puede especificarse opcionalmente la configuración con la que se desea que trabaje.

Cuando la especificación no indica el sensor concreto con el que muestrear —v. g. no se indica niguno, se indican varios—, el sistema se encargará de realizar la elección del sensor, según lo explicado en la Sección 15.7. En el Cuadro 13.12 se describen los atributos XML que definen la acción **medir**. La descripción de los atributos de los elementos **frecuencia** y **resolucion** de muestreo se muestra en el Cuadro 13.13 y Cuadro 13.14, respectivamente. La especificación del elemento **sensor** se muestra en el Cuadro 13.15, donde se hace referencia a su configuración, que es un elemento XML denominado **configuracion** y contenido en **sensor**.

```
1  <planDeMedicion id="1" nombre="PdM"
2                  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3                  xmlns:disparadores="http://www.disparadores.com"
4                  xmlns:equipamiento="http://www.equipamiento.com"
5                  xmlns="http://www.pdm.com"
6                  xsi:schemaLocation="http://www.pdm.com ../../esquemas/pdm.xsd">
7    <!-- ... -->
8  </planDeMedicion>
```

Algoritmo 13.13: Esqueleto del PdM

Atributo	Descripción
id	Identificador unívoco de la acción dentro de la lista a la cual pertenece
medida	Nombre de la medida cuyas muestras serán enviadas
muestras	Indica el número máximo de muestras a tomar. Con el valor **INF** se indica que se toma un número ilimitado de muestras. El atributo **muestras** es opcional y su valor por defecto es **INF**
frecuencia	Elemento XML que especifica la frecuencia de muestreo (véase el Cuadro 13.13)
resolucion	Elemento XML que especifica la resolución de muestreo (véase el Cuadro 13.14)
sensor	Elemento XML que especifica con qué sensor debe muestrearse la medida. Se puede indicar una lista o conjunto de sensores —i. e. varios elementos **sensor** (véase el Cuadro 13.15)

Cuadro 13.12: Atributos usados para especificar la acción **medir**

Atributo	Descripción
valor	Valor numérico de la frecuencia de muestreo
unidad	Unidad en que se expresa el valor de la frecuencia de muestreo; suele expresarse con múltiplos o submúltiplos del Hz

Cuadro 13.13: Atributos usados para especificar el elemento **frecuencia** de la acción **medir**

Atributo	Descripción
valor	Valor numérico de la resolución de muestreo
unidad	Unidad en que se expresa el valor de la resolución de muestreo; la unidad dependerá del tipo de medida que se especifique en la acción **medir**

Cuadro 13.14: Atributos usados para especificar el elemento **resolucion** de la acción **medir**

Atributo	Descripción
id	Identificador unívoco del sensor dentro de la lista de sensores que pueden usarse para la toma de muestras de la acción **medir**
nombre	Nombre que identifica unívocamente al sensor dentro del equipamiento sensorial (véase el Sección 8.1)
configuracion	Especificación de la configuración que debe tener el sensor durante la toma de muestras de la medida (véase el Cuadro 13.16)

Cuadro 13.15: Atributos usados para especificar el elemento **sensor** de la acción **medir**

Atributo	Descripción
nombre	Nombre del fichero que especifica la configuración que debe cargarse o usarse en el sensor; este fichero se debe encontrar almacenado en el sistema del vehículo

Cuadro 13.16: Atributos usados para especificar el elemento **configuracion**, que pertenece al elemento **sensor** de la acción **medir**

En el PdM se podrá especificar la toma de muestras de variables internas del sistema. Mediante sensores virtuales se producirán muestras de dichas variables, facilitando el tratamiento homogéneo en el resto del sistema. El PdC se validará para comprobar que las medidas indicadas están soportadas por el equipamiento sensorial (véase la Definición 12.7, Sección 12.3 y Sección 8.1). Además, en el caso de que se indique uno o varios sensores con los que muestrear la medida, también se validará que dichos sensores formen parte del equipamiento sensorial del vehículo. Los ficheros de configuración que se especifiquen también deberán estar presentes en el inventario del sistema (véase el Capítulo 9 y Sección 15.3.2).

El Algoritmo 13.14 muestra los siguientes ejemplos de tareas con acciones de medición:

1. La tarea 1 se encarga de la medición de un máximo de 50 muestras de la **temperatura exterior** a una frecuencia de muestreo de 0.1Hz y una resolución de 2m°C. No se indica ningún sensor, de modo que lo seleccionará el sistema.

2. La tarea 2 se encarga de la medición de una máximo de 500 muestras de la **temperatura exterior** a una frecuencia de muestreo de 1Hz y una resolución de 1°C. Se especifica un único sensor, cuyo nombre es **LM35-1**, con el que se debe muestrear usando la configuración que vendrá definida en el fichero **lm35-1.cfg**.

```
<tarea id="1" nombre="medirTemperatura">
  <disparadores:disparadores>
    <!-- ... -->
  </disparadores:disparadores>
  <acciones>
    <medir id="1" medida="temperatura exterior" muestras="50">
      <frecuencia valor="0.1" unidad="Hz"/>
      <resolucion valor="2" unidad="miligrado centigrado"/>
    </medir>
  </acciones>
  <periodoInhibicion valor="10" unidad="minuto" />
</tarea>

<tarea id="2" nombre="medirTemperaturaSensor">
  <disparadores:disparadores>
    <!-- ... -->
  </disparadores:disparadores>
  <acciones>
    <medir id="1" medida="temperatura exterior" muestras="500">
      <frecuencia valor="1" unidad="Hz"/>
      <resolucion valor="1" unidad="grado centigrado"/>
      <sensor id="1" nombre="LM35-1">
        <equipamiento:configuracion nombre="lm35-1.cfg"/>
      </sensor>
    </medir>
  </acciones>
  <periodoInhibicion valor="10" unidad="minuto" />
</tarea>
```

Algoritmo 13.14: Acciones del PdM

13.6. Plan de Navegación

El PdN es un plan que no se especifica en base a tareas porque no resultan semánticamente apropiadas. En su lugar se realiza una especificación *ad hoc* de los diferentes tipos de misión de navegación (véase la Sección 11.2, Sección 11.3 y Sección 11.4). El Algoritmo 13.15 muestra el esqueleto del PdN, cuyos elementos principales cubren la tipología de misiones de navegación y son:

Atributo	Descripción
repeticion	Número de veces que debe repetirse el PdN

Cuadro 13.17: Atributos del PdN, adicionales a los atributos comunes a todos los Planes de la Misión (véase el Cuadro 13.3)

ruta De acuerdo con la Definición 11.6, se especificará una ruta formada por una lista de *waypoints* y transectos. Los transectos permiten imponer condiciones relativas al recorrido entre dos *waypoints* (véase la Definición 11.4 y Definición 11.5). En la Sección 13.6.1 se detalla la especificación de las misiones de seguimiento de rutas, usando el elemento XML **ruta**.

area De acuerdo con la Definición 11.7, se especificará un área mediante una superficie poligonal[7] y una lista de especificaciones y condiciones que determinan cómo debe explorarse. Las especificaciones de exploración del área incluyen el seguimiento de medidas enunciado en la Sección 11.4; el área servirá de límite de exploración. En la Sección 13.6.2 se detalla la especificación de las misiones de exploración de áreas y seguimiento de medidas, usando el elemento XML **area**.

zonaProhibida Se especifica una superficie poligonal[7] por la que el vehículo debe evitar pasar. En la Sección 13.6.3 se detalla la especificación del elemento XML **zonaProhibida**.

```
<planDeNavegacion id="1" nombre="PdN" repeticion="1"
                  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                  xmlns:disparadores="http://www.disparadores.com"
                  xmlns="http://www.pdn.com"
                  xsi:schemaLocation="http://www.pdn.com ../../esquemas/pdn.xsd">
  <!-- ... -->
</planDeNavegacion>
```

Algoritmo 13.15: Esqueleto del PdN

El PdN estará formado por una lista de rutas y áreas por las que se navegará respetando el orden con el que se han especificado. Las rutas y áreas podrán entrelazarse como se desee y cuando el sistema termine de realizar una de ellas pasará inmediatamente a ejecutar la siguiente (véase la Sección 15.5 y Sección 15.6 para el tratamiento que hace *Sick***AUV**). También se puede especificar una lista de zonas prohibidas, donde el orden de definición es irrelevante y se recomienda especificarlas al final del PdN, por claridad. Si no se especifica ninguna ruta ni área el vehículo no navegará, salvo para evitar las zonas prohibidas —si las hubiere. Esta singularidad cubre las misiones de tipo boya (véase la Sección 11.1).

Aparte de los atributos identificativos del plan, comentados en el Cuadro 13.3, el PdN incluye un atributo adicional para indicar el número de repeticiones del mismo. El atributo XML **repeticion** se define en el Cuadro 13.17 y sus posibles valores se muestran en el Cuadro 13.18.

[7] Una superficie poligonal se definirá mediante vértices tridimensionales, i. e. con coordenadas xyz. No obstante, es posible simplificar la especificación a una espacio bidimensional en las herramientas del **planificador** e introducir las profundidades *a posteriori*. En cualquier caso, este tipo de superficies suelen definirse con una profundidad constante.

Valor	Descripción
0	No se ejecuta el PdN.
$[1, \infty)$	Se repite el PdN el número de veces indicado. Con 1 se ejecutará una sola vez.
∞	El PdN se repetirá continuamente, cada vez que se termine. Se especifica con **INF**.

Cuadro 13.18: Valores de repetición del PdN

Atributo	Descripción
id	Identificador unívoco de la ruta dentro del PdN
nombre	Nombre en lenguaje natural que es representativo de la ruta
waypoint	Lista de *waypoints* por los que debe pasar el vehículo (véase el Cuadro 13.20). Debe especificarse al menos un *waypoint*
transecto	Lista de transectos que especifican cómo navegar entre dos *waypoints* pertenecientes a la ruta (véase la Definición 11.5)

Cuadro 13.19: Atributos y elementos usados para especificar una **ruta**

13.6.1. Seguimiento de Rutas

El seguimiento de rutas explicado en la Sección 11.2 se especificará en el PdN mediante el elemento XML **ruta**, cuyos atributos y elementos XML se muestran en el Cuadro 13.19. La ruta está formada fundamentalmente por una lista de *waypoints* y otra de transectos, donde se definen unas condiciones y parámetros que determinan la navegación para alcanzar los *waypoints* (véase la Definición 11.4 y Definición 11.5).

La lista de *waypoints* especifica los puntos por los que debe pasar el vehículo. El orden en que se especifiquen los *waypoints* indica la secuencia que debe seguir el vehículo al recorrer la ruta[8]. La ruta debe tener al menos un *waypoint*. En el caso de que tenga sólo uno, el vehículo determinará como alcanzarlo desde la posición actual o de partida. Cuando se alcanza el último *waypoint* de la ruta, ésta queda terminada y el vehículo continua ejecutando el resto del PdN.

La especificación del paso del vehículo por cada *waypoint* de la ruta se muestra en el Cuadro 13.20. Para ello se especifica la **pose** y **velocidad** que debe tener el vehículo (véase el Cuadro 13.21 y Cuadro 13.22, respectivamente). También se indicará la forma en que el vehículo debe interpolar el paso del transecto de entrada en el *waypoint* al transecto de salida. El vehículo describirá un giro al pasar por el *waypoint* de acuerdo con lo especificado en el elemento **interpolacion** (véase el Cuadro 13.26).

Como el vehículo se desplaza lineal y angularmente en tres dimensiones, tanto la **pose** como la **velocidad** tendrán seis componentes. Las tres componentes lineales y las tres angulares se especifican con el elemento **posicion** y **orientacion**, respectivamente —tanto para la **pose** como para la **velocidad**[9]. La posición se especificará en relación a un sistema de coordenadas cartesianas xyz, mientras que la orientación se especificará

[8] El orden de los *waypoints* de la ruta lo determina la numeración de sus identificadores (**id**). No obstante, el **planificador** se encargará de ordenar la definición de los *waypoints* dentro de la ruta según el **id**.

[9] La velocidad angular suele ser nula —i. e. $(\phi, \theta, \psi) = (0, 0, 0)$. Si no lo es, el vehículo irá girando sobre sí mismo mientras navega. Según los ángulos de rotación sobre los que se gire, se podría comprometer el alcance del siguiente *waypoint*; esta incompatibilidad es anticipable en el proceso de validación (véase la Sección 12.3.1).

Atributo	Descripción
id	Identificador unívoco del *waypoint* dentro de la ruta
pose	Posición y orientación que el vehículo debe tener al pasar por el *waypoint* (véase el Cuadro 13.21)
velocidad	Velocidad lineal y angular que el vehículo debe tener al pasar por el *waypoint* (véase el Cuadro 13.22). Este elemento es opcional; si no se indica, la velocidad la determinará el sistema (véase la Sección 15.6)
interpolacion	Define la forma en que se debe interpolar la transición de un transecto de entrada a otro de salida de un *waypoint* (véase el Cuadro 13.26). Este elemento es opcional; si no se indica, el proceso de interpolación lo determinará el sistema (véase la Sección 15.6)

Cuadro 13.20: Atributos y elementos usados para especificar un **waypoint**

Atributo	Descripción
posicion	Posición del vehículo en el espacio tridimensional, definido por los ejes xyz (véase el Cuadro 13.23); la posición es precisamente la ubicación del *waypoint*
orientacion	Orientación del vehículo en el espacio tridimensional, con ángulos de rotación $\phi\theta\psi$ (véase el Cuadro 13.24). Este elemento es opcional; si no se indica el sistema usará la orientación que estime más apropiada para el paso por el *waypoint*
incertidumbre	Incertidumbre o margen de error máximo en la posición y orientación que debe tener el vehículo al pasar por el *waypoint* (véase el Cuadro 13.25). Este elemento es opcional; si no se indica, la pose del vehículo debe ser exactamente la definida para el *waypoint*

Cuadro 13.21: Elementos usados para especificar la **pose** en un *waypoint*

Atributo	Descripción
posicion	Velocidad lineal del vehículo en el espacio tridimensional, definido por los ejes xyz (véase el Cuadro 13.23)
orientacion	Velocidad angular del vehículo en el espacio tridimensional, con variación de los ángulos de rotación $\phi\theta\psi$ (véase el Cuadro 13.24)

Cuadro 13.22: Elementos usados para especificar la **velocidad** en un *waypoint*

Atributo	Descripción
x	Valor de la posición o velocidad lineal en el eje x
y	Valor de la posición o velocidad lineal en el eje y
z	Valor de la posición o velocidad lineal en el eje z
unidad	Unidad en la que se expresan todas las componentes —xyz— de la posición o velocidad lineal

Cuadro 13.23: Atributos usados para especificar la **posicion** de la **pose** o **velocidad** de un *waypoint*

Atributo	Descripción
roll	Valor de la orientación o velocidad angular para el ángulo de rotación ϕ
pitch	Valor de la orientación o velocidad angular para el ángulo de rotación θ
yaw	Valor de la orientación o velocidad angular para el ángulo de rotación ψ
unidad	Unidad en la que se expresan todas las componentes —$\phi\theta\psi$— de la orientación o velocidad angular

Cuadro 13.24: Atributos usados para especificar la **orientacion** de la **pose** o **velocidad** de un *waypoint*

Atributo	Descripción
valor	Radio de la esfera que determina la distancia euclídea máxima del *waypoint*. Es una medida de la incertidumbre o error máximo que puede cometerse al alcanzarlo; un valor 0 indica que el vehículo debe tener exactamente la **pose** definida en el *waypoint*
unidad	Unidad en la que se expresa **valor**

Cuadro 13.25: Atributos usados para especificar la **incertidumbre** de la **pose** o **velocidad** de un *waypoint*

mediante los ángulos de rotación $\phi\theta\psi$ de los ejes del sistema de referencia del propio vehículo. En ambos casos se tomará cada una de las coordenadas y se indicará la unidad empleada (véase el Cuadro 13.23 y Cuadro 13.24).

Respecto a la **incertidumbre** de la **pose**, ésta se especifica mediante el radio de una esfera, lo que define la distancia euclídea máxima d_{max} que puede haber entre el vehículo y el *waypoint* al pasar por éste (véase la Figura 13.1 (a)). El cómputo de esta distancia se muestra en la Definición 13.9 y su expresión matemática en la Ecuación 13.1. El vehículo deberá mantener una distancia d respecto al *waypoint*, tal que $d \leq d_{max}$.

Definición 13.9 (Incertidumbre. Distancia euclídea). *Sea un* waypoint *definido por las coordenadas* (x_w, y_w, z_w) *y una posición* (x_t, y_t, z_t) *del vehículo en un instante* t *durante el paso por el* waypoint*, la distancia euclídea máxima* d_{max} *entre ambos será:*

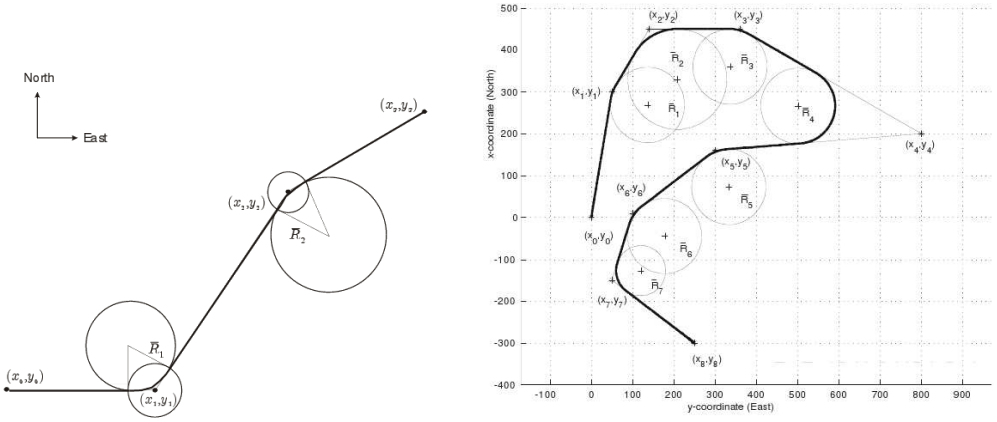
$$d_{max} = \sqrt{(x_w - x_t)^2 + (y_w - y_t)^2 + (z_w - z_t)^2} \tag{13.1}$$

que se corresponde con el radio de una esfera, que es lo que se especifica en el atributo XML ***valor*** *de la* ***incertidumbre*** *(véase el Cuadro 13.25).*

Los transectos son opcionales y definirán una serie de condiciones relativas a la forma en que alcanzar el *waypoint* final del transecto, desde su *waypoint* inicial (véase el Cuadro 13.27). Estas condiciones se entienden como parámetros de configuración de la navegación en el transecto y tienen la finalidad de indicar los valores de **tiempo** o **velocidad** deseados o aconsejables. No se trata de valores obligatorios ni impuestos, ya que podrán modificarse o ajustarse durante la ejecución de la misión para garantizar que se alcanza el *waypoint* final del transecto; los valores indicados servirán de guía para el ajuste. Además, la validación de la misión comprobará que estas condiciones no sean incompatibles con ello.

Atributo	Descripción
valor	Radio máximo de la esfera que determina el arco de giro para pasar del transecto de entrada en el *waypoint*, al transecto de salida (véase la Figura 13.1 (b)). Éste sería el giro menos pronunciado —i. e. de menor ángulo— admitido
unidad	Unidad en la que se expresa **valor**

Cuadro 13.26: Atributos usados para especificar la **interpolacion** de los transectos que une un *waypoint*



(a) Representación de los círculos de incertidumbre de los *waypoints*, que son el centro de éstos; también se muestra la representación de la interpolación de la figura (b)

(b) Representación de los círculos de interpolación en el paso de un transecto de entrada en un *waypoint*, al transecto de salida

Figura 13.1: Representación de la incertidumbre e interpolación de los *waypoints* [Fossen, 2002]

Atributo	Descripción
id	Identificador unívoco del transecto dentro de la ruta
inicio	Identificador del *waypoint* inicial del transecto
fin	Identificador del *waypoint* final del transecto
tiempo	Tiempo máximo para realizar el transecto, i. e. llegar al *waypoint* **fin** desde el *waypoint* **inicio** (véase el Cuadro 13.28)
velocidad	Velocidad de crucero que el vehículo debe mantener mientras recorre el transecto (véase el Cuadro 13.29)

Cuadro 13.27: Atributos y elementos usados para especificar un **transecto**

Para una determinada ruta no puede haber transectos que hagan referencia a una misma pareja de *waypoints* de **inicio** y **fin**, puesto que sus respectivas especificaciones podrían resultar inconsistentes. Si se define un transecto dentro de otro, i. e. sus *waypoints* de **inicio** y **fin** están dentro del intervalo que definen los *waypoints* de **inicio** y **fin** de otro transecto, se tomarán las condiciones del más interno (véase la Propiedad 13.1). El **planificador** validará que las condiciones de los transectos más internos no sean inconsistentes con las de los más externos.

Propiedad 13.1 (Transecto interno). *Sea un transecto* t_1 *definido desde el* waypoint

Atributo	Descripción
valor	Tiempo máximo que debe tardarse en realizar el transecto
unidad	Unidad en la que se expresa el **valor**

Cuadro 13.28: Atributos usados para especificar el **tiempo** máximo que debe tardarse en realizar un transecto

Atributo	Descripción
posicion	Velocidad lineal de crucero en el transecto (véase el Cuadro 13.23)
orientacion	Velocidad angular de crucero en el transecto (véase el Cuadro 13.24)

Cuadro 13.29: Elementos usados para especificar la **velocidad** de crucero a mantener en un transecto

$w_{i_{t_1}}$ al $w_{f_{t_1}}$ y otro t_2 desde el waypoint *$w_{i_{t_2}}$ al $w_{f_{t_2}}$, se dice que el transecto t_1 está contenido en el transecto t_2 y se representa por $t_1 \subset t_2$, si y sólo si el intervalo definido por los* waypoints *del transecto t_1 ($[w_{i_{t_1}}, w_{f_{t_1}}]$) está contenido en el intervalo definido por los* waypoints *del transecto t_2 ($[w_{i_{t_2}}, w_{f_{t_2}}]$), como expresa la implicación de la Ecuación 13.2 y se ilustra en el Ejemplo 13.1.*

$$[w_{i_{t_1}}, w_{f_{t_1}}] \subset [w_{i_{t_2}}, w_{f_{t_2}}] \Leftrightarrow t_1 \subset t_2 \tag{13.2}$$

Ejemplo 13.1 (Transecto interno a otro). *Sea un transecto t_1 definido desde el* waypoint *3 al 5 y otro t_2 desde el* waypoint *1 al 7, se dice que el transecto t_1 está contenido en el transecto t_2 y se representa por $t_1 \subset t_2$.*
Esto es así porque el intervalo definido por los waypoints *del transecto t_1 ($[3, 5]$) está contenido en el intervalo definido por los* waypoints *del transecto t_2 ($[1, 7]$), i. e.* $[3, 5] \subset [1, 7]$.

Propiedad 13.2 (Transecto. *Waypoint* de inicio y fin). *Para un transecto dado, sean i y f los identificadores de sus* waypoints *de **inicio** y **fin**, respectivamente, éstos deben pertenecer a* waypoints *definidos en la ruta y cumplir la siguiente condición de la Ecuación 13.3.*

$$i < f \tag{13.3}$$

Los *waypoints* de **inicio** y **fin** de un transecto no tienen que ser necesariamente consecutivos, pero para que el transecto sea válido debe cumplir la Propiedad 13.2. Cuando no se defina un transecto para una pareja de *waypoints*, el vehículo determinará la velocidad de crucero apropiada para alcanzar el siguiente *waypoint*. No obstante, el **planificador** intentará que para todos los *waypoints* quede definida la velocidad crucero o el tiempo máximo que puede tardarse en recorrer un transecto (véase el Cuadro 13.29 y Cuadro 13.28, respectivamente).[10]

Los elementos XML **tiempo** y **velocidad** son obligatorios y excluyentes mutuamente, i. e. debe definirse uno y sólo uno de los dos para cada transecto que se defina en la ruta.

[10] Para que quede definida la velocidad de crucero o el tiempo máximo que puede tardarse en alcanzar un *waypoint*, no es necesario definir un transecto con cada pareja de *waypoints*, pues basta con definir un transecto cuyos *waypoints* de inicio y fin incluyan al resto de *waypoints* de la ruta —v. g. definir un transecto cuyo *waypoint* de inicio y de fin sean el primer y último *waypoint* de la ruta.

Esto es así porque la información que proporcionan sería redundante, i. e. conocida la longitud de un transecto l_t:

1. Si se define el **tiempo** t_t, se puede hallar la **velocidad** de crucero v_t con la Ecuación 13.4.

2. Si se define la **velocidad** de crucero v_t, se puede hallar el **tiempo** t_t con la Ecuación 13.5.

$$v_t = l_t / t_t \tag{13.4}$$

$$t_t = l_t / v_t \tag{13.5}$$

Definición 13.10 (Longitud de un transecto). *La longitud de un transeto l_t se define como la distancia euclídea entre el* waypoint *de* ***inicio*** *w_i y de* ***fin*** *w_f del transecto. La Ecuación 13.6 muestra la expresión matemática, donde un* waypoint *se define en el espacio tridimensional por el vector de coordenadas (x, y, z).*

$$l_t = \sqrt{(w_{i_x} - w_{f_x})^2 + (w_{i_y} - w_{f_y})^2 + (w_{i_z} - w_{f_z})^2} \tag{13.6}$$

El Algoritmo 13.16 muestra la especificación de una ruta de ejemplo, donde se indica lo siguiente:

1. *Waypoint* 1, donde se especifica la pose y velocidad que debe tener el vehículo:

   Pose El vehículo debe alcanzar la posición $(x, y, z) = (30, 20, -10)$m con una orientación de $(\phi, \theta, \psi) = (0, 0, 1)$rad. Se especifica una incertidumbre de 2m.

   Velocidad El vehículo debe pasar por el *waypoint* con una velocidad lineal y angular de $(x, y, z) = (1, 0.2, 0)$m/s y $(\phi, \theta, \psi) = (0, 0, 0.1)$rad/s, respectivamente.

   Interpolación El vehículo interpolará el paso del transecto de entrada en el *waypoint* al de salida usando un giro definido por una esfera de radio de 10m.

2. *Waypoint* 2, donde sólo se indica la pose que debe tener el vehículo (el sistema determinará la velocidad adecuada):

   Pose El vehículo debe alcanzar la posición $(x, y, z) = (50, 30, 0)$m con cualquier orientación, pues no se especifica nada al respecto. Se especifica una incertidumbre de 2m.

   Interpolación El vehículo interpolará el paso del transecto de entrada en el *waypoint* al de salida usando un giro definido por una esfera de radio de 10m.

3. Transecto 1 cuyo *waypoint* inicial y final son el 1 y 2, respectivamente. Se indica la velocidad de crucero que debe mantenerse en el transecto, cuyas componentes lineal y angular indican:

   Velocidad lineal 1m/s y 0.2m/s en el eje x e y, respectivamente.

   Velocidad angular 0 en todas las componentes —expresada en rad/s.

```
<ruta id="1" nombre="salida">
  <waypoint id="1">
    <pose>
      <posicion x="30" y="20" z="-10" unidad="m"/>
      <orientacion roll="0" pitch="0" yaw="1" unidad="rad"/>
      <incertidumbre valor="2" unidad="m"/>
    </pose>
    <velocidad>
      <posicion x="1" y="0.2" z="0" unidad="m*s^-1"/>
      <orientacion roll="0" pitch="0" yaw="0.1" unidad="rad*s^-1"/>
    </velocidad>
    <interpolacion valor="10" unidad="m"/>
  </waypoint>
  <waypoint id="2">
    <pose>
      <posicion x="50" y="30" z="0" unidad="m"/>
      <incertidumbre valor="2" unidad="m"/>
    </pose>
    <interpolacion valor="10" unidad="m"/>
  </waypoint>

  <transecto id="1" inicio="1" fin="2">
    <velocidad>
      <posicion x="1" y="0.2" z="0" unidad="m*s^-1"/>
      <orientacion roll="0" pitch="0" yaw="0" unidad="rad*s^-1"/>
    </velocidad>
  </transecto>
</ruta>
```

Algoritmo 13.16: Ruta (PdN)

El **planificador** ofrecerá distintas facilidades para la definición de rutas, a través de herramientas gráficas como puede verse en la Figura 11.2 (véase la Apéndice A). Esto permitirá definir las rutas de forma diferente a cómo se especifican en el PdN. No obstante, el **planificador** debe generar el PdN con la sintaxis aquí comentada, antes de la validación de la misión y su posterior envío al vehículo (véase la Sección 12.3). En determinados casos, es posible convertir la especificación de exploración de un área[11] en una de seguimiento de ruta. En este caso, el vehículo se limitará a seguir la ruta generada, mientras que si se hubiera especificado directamente el área, el sistema tendría que determinar cómo explorarla de acuerdo a la especificación de la misma (véase la Sección 13.6.2).

13.6.2. Exploración de Áreas y Seguimiento de Medidas

La exploración de áreas y el seguimiento de medidas explicados en la Sección 11.3 y Sección 11.4, respectivamente, se especificarán en el PdN mediante el elemento XML **area**, cuyos atributos y elementos XML se muestran en el Cuadro 13.30. El área está formada fundamentalmente por una lista de vértices (**vertice**) que definen el área junto con la **profundidad**, el **tiempo** máximo de exploración y el tipo de **recorrido**, que determina cómo debe explorarse el área —incluyendo la posibilidad del seguimiento de medidas. Se soportan los siguientes tipos de recorrido:

Deriva El vehículo se quedará a la deriva dentro del área y sólo tendrá que evitar salirse de ésta (véase la Sección 13.6.2.1).

Exploración de Área Explorar el área describiendo transectos de un tipo determinado (véase la Sección 13.6.2.2).

[11] Las especificaciones de área de seguimiento de medidas (véase la Sección 11.4) no son susceptibles de convertirse en una especificación de seguimiento de rutas.

Atributo	Descripción
id	Identificador unívoco del área dentro del PdN
nombre	Nombre en lenguaje natural que es representativo del área
vertice	Lista de al menos 3 vértices ordenados que definen el área (véase el Cuadro 13.32)
profundidad	Profundidades mínima y máxima, que determinan desde qué profundidad se empieza a explorar el área y a cuál se termina (véase el Cuadro 13.33)
tiempo	Tiempo máximo durante el que se puede estar explorando el área (véase el Cuadro 13.34)
recorrido	Lista de al menos un recorrido, donde cada uno define el comportamiento o forma en que el vehículo debe explorar el área, bajo determinadas condiciones (véase el Cuadro 13.35)

Cuadro 13.30: Atributos y elementos usados para especificar un **área**

Mínima	Máxima	Interpretación
		Profundidad constante e igual a la que el vehículo tenía antes de comenzar la exploración del área
	✓	Se toma como profundidad mínima la superficie marina
✓		Se toma como profundidad máxima el fondo marino
✓	✓	Se toman las profundidades mínima y máxima especificadas

Cuadro 13.31: Casos posibles de especificación de la profundidad mínima y máxima de un área

Atributo	Descripción
x	Coordenada x del vértice
y	Coordenada y del vértice
z	Coordenada z del vértice
unidad	Unidad en que se expresan las coordenadas del vértice

Cuadro 13.32: Atributos usados para especificar un **vértice** del área

Seguimiento de Medidas Seguir o rastrear una determinada medida dentro del área (véase la Sección 13.6.2.3).

Los vértices usados para definir el área tienen 3 coordenadas, pero por simplicidad el **planificador** definirá una superficie plana —i. e. a profundidad constante, especificando sólo 3 vértices— y la profundidad mínima y máxima se definen por separado. De esta forma, el área sería un prisma[12] donde los vértices definen el polígono de la base y la profundidad mínima y máxima determinarán su altura y ubicación en el espacio. A la hora de especificar la profundidad se pueden dar los casos del Cuadro 13.31, según los tipos de profundidad definidos. Los vértices deben estar ordenados secuencialmente para definir el área. El orden se determinará en base al identificador de los mismos, aunque el **planificador** también se encargará de que aparezcan ordenados en el PdN.

Para indicar cuándo debe explorarse el área con un determinado recorrido, se puede especificar una lista de disparadores para cada uno de ellos. Si no se indica ningún

[12] Si se especifica un área con más de 3 vértices el resultado sería un poliedro, aunque la herramienta de construcción de misiones del **planificador** dará preferencia a la especificación mediante prismas.

Atributo	Descripción
minima	Profundidad mínima del área
maxima	Profundidad máxima del área
unidad	Unidad en que se expresan la profundidades mínima y máxima

Cuadro 13.33: Atributos usados para especificar la **profundidad** mínima y máxima del área

Atributo	Descripción
valor	Duración de la exploración del área. Con **INF** se indica que no hay restricción temporal y el vehículo podrá estar explorando el área por un tiempo indefinido
unidad	Unidad en que se expresa el **valor** y **holgura**
holgura	Porcentaje de error aplicable al **valor** del tiempo máximo de exploración del área

Cuadro 13.34: Atributos usados para especificar el **tiempo** máximo para explorar el área

Atributo	Descripción
id	Identificador unívoco del recorrido dentro del área
disparadores	Lista de disparadores que determinan cuándo debe explorarse el área con el recorrido al que pertenece (véase la Sección 13.2.1)
deriva	Mantenerse a la deriva dentro del área; este elemento no tiene ningún atributo ni elemento interno
seguimiento	Seguimiento de una medida dentro del aréa (véase el Cuadro 13.41)
transectos	Exploración del área mediante transectos (véase el Cuadro 13.36)

Cuadro 13.35: Atributos y elementos usados para especificar el **recorrido** para explorar el área

disparador, el área sólo podrá tener un único recorrido, que estará siempre activo. Si hay varios recorridos, las listas de disparadores de cada uno de ellos deben ser disjuntas y completas, pues debe hacer un y sólo un recorrido activo en cada instante de tiempo (véase la Propiedad 13.3). Durante la validación de la misión se comprobará que no se produzca esta incompatibilidad anticipable (véase la Sección 12.3.1). Cuando se cumplan los disparadores, el vehículo explorará el área de acuerdo al tipo de recorrido especificado, que debe ser uno y sólo uno de los siguientes: **deriva**, **transectos**, **seguimiento**. Si no se especifica ningún tipo de recorrido, se tomará el recorrido de **deriva** por defecto.

Propiedad 13.3 (Recorrido. Disjunción y completitud). *Sea un área a para la que se especifican n recorridos, donde para cada recorrido r_i $\forall_{1\leq i\leq n}$ se define una lista de disparadores d_{r_i}, se debe cumplir que estas listas sean:*

Disjuntas *La condición de disjunción establece que no debe haber listas de disparadores que se cumplan simultáneamente (véase la Ecuación 13.7), para que no haya más de un recorrido activo.*

Completas *La condición de completitud establece que el conjunto de todas las listas de disparadores debe abarcar todo el universo U de casos (véase la Ecuación 13.8),*

para que siempre haya un recorrido activo.

Con esto se garantiza que en un instante de tiempo concreto haya un y sólo un recorrido activo.

$$d_{r_i} \neq d_{r_j} \forall_{1 \leq i \leq n,\ 1 \leq j \leq n,\ i \neq j} \tag{13.7}$$

$$\bigcap_{i=1}^{n} d_{r_i} = U \tag{13.8}$$

13.6.2.1. Deriva. Boya

Con el tipo de recorrido **deriva** se indica al vehículo que se mantenga a la deriva dentro del área especificada. Se trata de una misión de tipo boya (véase la Sección 11.1) donde el sistema sólo tendrá que evitar salirse del área. El grado de actuación para mantenerse dentro del área variará de forma directamente proporcional al tamaño de ésta. Para emular una boya a la deriva se definiría un área muy grande, de modo que apenas se navegará —para evitar salirse del área. En el Algoritmo 13.17 se muestra un ejemplo de exploración de un área con un recorrido a la deriva, donde se especifica lo siguiente:

1. Lista de 4 vértices (**vertice**) y profundidades (**profundidad**) mínima y máxima —de 10m y 100m, respectivamente— que definen el área a explorar.
2. El **tiempo** máximo para explorar el área es de 30min ± 10 %.
3. Hay un único recorrido —identificado como 1—, sin disparadores y cuyo recorrido es a la deriva (**deriva**). Este recorrido estará activo continuamente, ya que no tiene ningún disparador.[13]

```
<area id="2" nombre="deriva">
  <vertice id="1" x="40" y="20" z="10" unidad="m"/>
  <vertice id="2" x="20" y="10" z="20" unidad="m"/>
  <vertice id="3" x="30" y="20" z="10" unidad="m"/>
  <vertice id="4" x="20" y="20" z="10" unidad="m"/>

  <profundidad minima="10" maxima="100" unidad="m"/>
  <tiempo valor="30" unidad="minuto" holgura="10"/>

  <recorrido id="1">
    <!-- Al no haber disparadores, se cumple siempre -->
    <disparadores:disparadores/>
    <deriva/>
  </recorrido>
</area>
```

Algoritmo 13.17: Área (PdN). Deriva

13.6.2.2. Exploración. Recorrido dividido en transectos

Con el tipo de recorrido **transectos** se indica al vehículo que explore el área recorriendo una serie de transectos. Se trata de una misión de exploración de área (véase la Sección 11.3), que se especifica mediante un patrón de recorrido basado en transectos (véase el Cuadro 13.36). Los patrones de recorrido se denominan modos de recorrido y el Cuadro 13.37 muestra los soportados en la especificación del PdN.

[13] Cuando hay un recorrido sin disparadores, i. e. que estará activo continuamente, sólo podrá especificarse un recorrido. Esto debe ser así para cumplir la condición de disjunción de las listas de disparadores de las especificaciones de recorridos de un área.

Atributo	Descripción
modo	Forma o patrón, subdividido en transectos, que determina cómo se explorará el área. En el Cuadro 13.37 se muestran los modos soportados
cantidad	Número de transectos en que se subdivide el recorrido del área, según el **modo**
ciclos	Número de veces que se explorará el área. Este atributo es opcional; si no se especifica, se explorará el área 1 vez
tiempo	Tiempo máximo para recorrer cada uno de los transectos en los que se subdivide el recorrido del área (véase el Cuadro 13.38). Este elemento es opcional; si no se especifica, se determinará a partir del tiempo máximo para explorar el área (véase el Cuadro 13.34)
profundidad	Incremento de profundidad al pasar de un transecto que se ha terminado de recorrer al siguiente (véase el Cuadro 13.39). Este elemento es opcional; si no se especifica, se considera un incremento de 0, i. e. se mantiene la misma profundidad para todos los transectos
angulo	Ángulo inicial a tomar para recorrer los transectos —i. e. la orientación del barrido de los transectos (véase la Definición 11.8)— (véase el Cuadro 13.40). Este elemento es opcional; si no se especifica, el vehículo determinará el ángulo más apropiado en el momento de comenzar la exploración del área —v. g. ángulo que minimice el consumo energético, aprovechando las corrientes marinas detectadas mediante el equipamiento sensorial

Cuadro 13.36: Atributos y elementos usados para especificar los **transectos** a realizar para explorar el área

Modo	Descripción
zigzag	Se recorre el área describiendo transectos transversales cambiando de sentido al pasar de un transecto al siguiente (véase la Figura 11.3 (a))
espiral	Se recorre el área en espiral, i. e. describiendo transectos circulares cuyo radio varía de un transecto al siguiente (véase la Figura 11.3 (b))
circunferencia	Se recorre el área describiendo circunsferencias de radio constante (cf. espiral)

Cuadro 13.37: Modos de recorrido soportados para la exploración de áreas

Atributo	Descripción
valor	Tiempo máximo para recorrer cada transecto de la subdivisión del recorrido de exploración del área
unidad	Unidad en que se expresa **valor**

Cuadro 13.38: Atributos usados para especificar el **tiempo** máximo para recorrer cada transecto del área

Atributo	Descripción
valor	Incremento de profundidad al pasar de un transecto al siguiente
unidad	Unidad en que se expresa **valor**

Cuadro 13.39: Atributos usados para especificar el incremento de **profundidad** entre transectos

Atributo	Descripción
valor	Ángulo inicial de orientación para realizar el barrido de los transectos
unidad	Unidad en que se expresa **valor**

Cuadro 13.40: Atributos usados para especificar el **angulo** de la orientación del barrido de los transectos

En la especificación del recorrido mediante **transectos** se ha optado por indicar simplemente el **tiempo** máximo para recorrer cada uno de los transectos en los que se subdivide el recorrido del área. Éste definirá indirectamente la velocidad de crucero que debe mantener el vehículo en el transecto (véase la Ecuación 13.4). Por este motivo no se dispone de la posibilidad de especificar la velocidad de crucero.

En el el Algoritmo 13.18 se muestra un ejemplo de exploración de un área con un recorrido basado en transectos, donde se especifica lo siguiente, respecto al recorrido:

1. El área se explorará mediante 10 transectos, usando el **modo** de recorrido *zigzag*.
2. El **tiempo** máximo para realizar cada transecto es de 5min.
3. Al pasar de un transecto al siguiente, la **profundidad** se incrementará en 10m.
4. Los transectos se realizarán con un **angulo** de 20° respecto a la geometría del área especificada

```
<area id="5" nombre="zigZagArea">
  <vertice id="1" x="40" y="20" z="10" unidad="m"/>
  <vertice id="2" x="20" y="10" z="20" unidad="m"/>
  <vertice id="3" x="30" y="20" z="10" unidad="m"/>
  <vertice id="4" x="20" y="20" z="10" unidad="m"/>
  <profundidad minima="10" maxima="100" unidad="m"/>
  <tiempo valor="30" unidad="minuto" holgura="10"/>
  <recorrido id="1">
    <!-- Al no haber disparadores, se cumple siempre -->
    <disparadores:disparadores/>
    <transectos modo="zigzag" cantidad="10">
      <tiempo valor="5" unidad="minuto"/>
      <profundidad valor="10" unidad="m"/>
      <angulo valor="20" unidad="grado"/>
    </transectos>
  </recorrido>
</area>
```

Atributo	Descripción
rango	Especifica el rango o intervalo de valores de la medida en el que ésta se seguirá (véase el Cuadro 13.42)
funcion	Especifica una función que determina cómo seguir la medida (véase el Cuadro 13.43)

Cuadro 13.41: Elementos usados para especificar el **seguimiento** de una medida dentro del área

Algoritmo 13.18: Área (PdN). Exploración en *zigzag*

13.6.2.3. Seguimiento de Medidas

El recorrido de **seguimiento** permite especificar misiones de seguimiento de medidas, que también necesitan de un área delimitadora del seguimiento (véase la Sección 11.4). La especificación de este tipo de misión se reduce a la indicación de la medida que se seguirá y la forma en que se hará (véase el Cuadro 13.41). Se soporta la especificación del seguimiento como un **rango** o una **funcion** (véase el Cuadro 13.42 y Cuadro 13.43, respectivamente). El seguimiento también se caracteriza porque el vehículo lo realizará considerando las fases indicadas en la Sección 11.4: **búsqueda**, **mantenimiento**, **finalización**. Por ello será común que se definan 3 recorridos, donde cada uno define el comportamiento de cada una de estas fases, habitualmente con el siguiente esquema:

Búsqueda Suele ser un recorrido a la **deriva** o de **transectos**, pues es el más apropiado para buscar un determinado valor de una medida, cubriendo grandes áreas de exploración.

Mantenimiento Se define un recorrido de **seguimiento** de medidas, con un **rango** o una **funcion**.

Finalización Se señaliza especificando un **recorrido** vacío, i. e. sólo tendrá los disparadores que definen cuándo se ha alcanzado esta fase.

En la Sección 11.4 se definen las principales estrategias de seguimiento de una medida, que en la especificación están soportadas mediante los elementos XML **rango** y **funcion**. La **funcion** permite definir cualquier tipo de función que produzcan un vector de navegación a partir de los valores de la medida, aunque la especificación sólo soporta el gradiente. En el caso del **rango** no se produce un vector de navegación si el valor de la medida se encuentra dentro del rango. En este caso, el comporatamiento del vehículo será el que esté predefinido en el sistema, que normalmente consistirá en avanzar hacia delante. Cuando el valor de la medida está fuera del rango, sí se dispone de un vector de navegación, que apunta en la dirección que previsiblmente hará que se vuelva a entrar en el rango.

En el Algoritmo 13.19 se muestra la especificación del seguimiento de una medida mediante un **rango**:

1. El recorrido 1 se encarga de la búsqueda de los valores de la medida necesarios para comenzar a seguirla. Este recorrido es a la **deriva** (véase la Sección 13.6.2.1).

Atributo	Descripción
medida	Medida a seguir
minimo	Valor mínimo de la medida dentro del rango en el que se seguirá
maximo	Valor máximo de la medida dentro del rango en el que se seguirá
unidad	Unidad en que se expresa el valor **minimo** y **maximo** de la **medida**

Cuadro 13.42: Atributos usados para especificar el **rango** en el que seguir una medida

Atributo	Descripción
medida	Medida a seguir
modo	Modo o tipo de función que determina cómo se seguirá la medida; en el Cuadro 13.44 se muestra la lista de modos soportados, la cual puede ser ampliada
submodo	Modificador o parámetro del **modo** de seguimiento de la medida

Cuadro 13.43: Atributos usados para especificar una **funcion** que determina cómo seguir una medida

Modo	Descripción
gradiente	El gradiente $\text{grad}\phi \equiv \nabla\phi$ es la dirección del espacio tridimensional en la que se produce una variación de los valores de una medida. Se distinguen los submodos ascendente y descendente, para indicar que se siga la dirección de máxima variación o la opuesta a ésta, respectivamente

Cuadro 13.44: Modos o funciones de seguimiento de una medida soportadas

2. Los disparadores del recorrido determinan el paso de la fase de búsqueda a la de mantenimiento, en la que se sigue la medida de acuerdo a un rango, en este caso. En el ejemplo, el recorrido 2 se encarga de la fase de mantenimiento cuando la medida **temperatura** t cumple la siguiente condición: $t \geq 25°\text{C}$. El seguimiento se realiza dentro del rango de **temperatura** $[10, 30]°\text{C}$.

En el Algoritmo 13.20 se muestra la especificación del seguimiento de una medida mediante una **funcion**:

1. El recorrido 1 se encarga de la búsqueda de los valores de la medida necesarios para comenzar a seguirla. Este recorrido está basado en **transectos** y se realiza mediante *zigzag* (véase la Sección 13.6.2.2).

2. Los disparadores del recorrido determinan el paso de la fase de búsqueda a la de mantenimiento, en la que se sigue la medida de acuerdo a una función, en este caso. En el ejemplo, el recorrido 2 se encarga de la fase de mantenimiento cuando la medida **temperatura** t cumple la siguiente condición: $27°\text{C} \leq t < 37°\text{C}$. El seguimiendo se realiza con una función de gradiente ascendente, lo cual se indica en el **modo** y **submodo**, respectivamente.

3. El recorrido 3 tiene la finalidad de indicar la terminación de la exploración del área, aparte de garantizar el cumplimiento de la condición de completitud de los disparadores de los recorridos (véase la Ecuación 13.8).

```
<area id="3" nombre="seguirRangoTemperatura">
  <vertice id="1" x="40" y="20" z="10" unidad="m"/>
  <vertice id="2" x="20" y="10" z="20" unidad="m"/>
  <vertice id="3" x="30" y="20" z="10" unidad="m"/>
  <vertice id="4" x="20" y="20" z="10" unidad="m"/>

  <profundidad minima="10" maxima="100" unidad="m"/>
  <tiempo valor="30" unidad="minuto" holgura="10"/>

  <recorrido id="1">
    <disparadores:disparadores>
      <disparadores:condicion id="1" medida="temperatura" operador="lt"
                              valor="25" unidad="grado centigrado"/>
    </disparadores:disparadores>
    <deriva/>
  </recorrido>
  <recorrido id="2">
    <disparadores:disparadores>
      <disparadores:condicion id="1" medida="temperatura" operador="ge"
                              valor="25" unidad="grado centigrado"/>
    </disparadores:disparadores>
    <seguimiento>
      <rango medida="temperatura"
             minimo="10" maximo="30" unidad="grado centigrado"/>
    </seguimiento>
  </recorrido>
</area>
```

Algoritmo 13.19: Área (PdN). Seguimiento: Rango

```
<area id="4" nombre="seguirGradienteTemperatura">
  <vertice id="1" x="40" y="20" z="10" unidad="m"/>
  <vertice id="2" x="20" y="10" z="20" unidad="m"/>
  <vertice id="3" x="30" y="20" z="10" unidad="m"/>
  <vertice id="4" x="20" y="20" z="10" unidad="m"/>

  <profundidad minima="10" maxima="100" unidad="m"/>
  <tiempo valor="30" unidad="minuto" holgura="10"/>

  <recorrido id="1">
    <disparadores:disparadores>
      <disparadores:condicion id="1" medida="temperatura" operador="lt"
                              valor="27" unidad="grado centigrado"/>
    </disparadores:disparadores>
    <transectos modo="zigzag" cantidad="10" ciclos="0">
      <tiempo valor="5" unidad="minuto"/>
      <profundidad valor="10" unidad="m"/>
      <angulo valor="20" unidad="grado"/>
    </transectos>
  </recorrido>
  <recorrido id="2">
    <disparadores:disparadores>
      <disparadores:condicion id="1" medida="temperatura" operador="ge"
                              valor="27" unidad="grado centigrado"/>
      <disparadores:condicion id="2" medida="temperatura" operador="lt"
                              valor="37" unidad="grado centigrado"/>
    </disparadores:disparadores>
    <seguimiento>
      <funcion medida="temperatura" modo="gradiente" submodo="ascendente"/>
    </seguimiento>
  </recorrido>
  <recorrido id="3">
    <disparadores:disparadores>
      <disparadores:condicion id="1" medida="temperatura" operador="ge"
                              valor="37" unidad="grado centigrado"/>
    </disparadores:disparadores>
    <!-- Se termina de recorrer el area, porque no hay acciones -->
  </recorrido>
</area>
```

Algoritmo 13.20: Área (PdN). Seguimiento: Función (gradiente)

Atributo	Descripción
vertice	Lista de al menos 3 vértices ordenados que definen la zona prohibida; en este sentido, se define de forma equivalente al área (véase el Cuadro 13.32)
profundidad	Profundiades mínima y máxima de la zona prohibida; en este sentido, se define de forma equivalente al área (véase el Cuadro 13.33)

Cuadro 13.45: Elementos usados para especificar una **zonaProhibida** del PdN

13.6.3. Zonas Prohibidas

Con el elemento **zonaProhibida** se pueden especificar múltiples zonas donde el vehículo no debe pasar durante la misión. La validación de la misión comprobará que no se hayan definido rutas ni áreas que pasen por las zonas prohibidas. Además, el sistema se encargará de evitar que el vehículo pase por ellas durante la ejecución del PdN. En el Algoritmo 13.21 se muestra un ejemplo de **zonaProhibida** en la que se especifica lo siguiente:

1. Zona prohibida definida mediante 3 vértices.

2. Las profundidades mínima y máxima de la zona prohibida son de 0m y 50m, respectivamente.

```
<zonaProhibida id="30" nombre="triangulo las bermudas">
  <vertice id="1" x="40" y="20" z="10" unidad="m"/>
  <vertice id="2" x="20" y="10" z="20" unidad="m"/>
  <vertice id="3" x="30" y="20" z="10" unidad="m"/>
  <profundidad minima="0" maxima="50" unidad="m"/>
</zonaProhibida>
```

Algoritmo 13.21: Zona Prohibida (PdN)

13.7. Plan de Supervisión

El PdS es un plan que se especifica en base a tareas. Su estructura básica es la mostrada en la Sección 13.2 y sus tareas sólo admiten acciones relacionadas con la supervisión y reconfiguración del resto de planes y el sistema; esto realmente permite la especificación de cualquier tipo de acción o comando del sistema (véase la Definición 15.1), pero con una interfaz homogénea —i. e. el nombre y la lista de parámetros. El Algoritmo 13.22 muestra el esqueleto del PdS, donde las acciones de supervisión soportadas por sus tareas son:

ejecutarPlan Especifica el nombre del plan a ejecutar y opcionalmente una lista de parámetros para configurarlo (véase el Cuadro 13.48). En el Cuadro 13.46 se describen los atributos XML que definen la acción **ejecutarPlan**.

ejecutarComando Especifica el nombre del comando a ejecutar y opcionalmente una lista de parámetros para configurarlo (véase el Cuadro 13.48). En el Cuadro 13.47 se describen los atributos XML que definen la acción **ejecutarComando**.

Atributo	Descripción
id	Identificador unívoco de la acción dentro de la lista a la cual pertenece
plan	Nombre del plan a ejecutar

Cuadro 13.46: Atributos usados para especificar la acción **ejecutarPlan**

Atributo	Descripción
id	Identificador unívoco de la acción dentro de la lista a la cual pertenece
comando	Nombre del comando a ejecutar

Cuadro 13.47: Atributos usados para especificar la acción **ejecutarComando**

Atributo	Descripción
id	Identificador unívoco del parámetro dentro de la lista de parámetros de la acción **ejecutarPlan** o **ejecutarComando** a la cual pertenece
valor	Valor con el que se instancia el parámetro

Cuadro 13.48: Atributos usados para especificar el elemento **parametro** de las acciones **ejecutarPlan** y **ejecutarComando**

```
1 <planDeSupervision id="1" nombre="PdS"
2                    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3                    xmlns:disparadores="http://www.disparadores.com"
4                    xmlns="http://www.pds.com"
5                    xsi:schemaLocation="http://www.pds.com ../../esquemas/pds.xsd">
6   <!-- ... -->
7 </planDeSupervision>
```

Algoritmo 13.22: Esqueleto del PdS

Tanto la acción **ejecutarPlan** como **ejecutarComando** pueden disponer de una lista de parámetros de **0 a N**. Estos parámetros se pasarán a la hora de ejecutar el plan o el comando, respectivamente. La reconfiguración de la misión se realizará con un comando específico para ello, denominado **CambiarVariableMision** (véase la Sección 13.9 para más detalles sobre los parámetros de la misión y la reconfiguración de la misma durante su ejecución), cuyos parámetros son:

Variable Elemento de los planes de la misión que se modificará o reconfigurará. Se puede indicar como un parámetro de la misión o como una sentencia XPath que apunte a un atributo XML concreto de los ficheros XML de la misión.

Valor Nuevo valor que se asignará a la variable.

El Algoritmo 13.24 muestra los siguientes ejemplos de tareas con acciones de supervisión:

1. La tarea 1 especifica las acciones a realizar en el hipotético caso de que se aborte la misión para regresar a la base. Estas acciones son, respectivamente:

a) Ejecutar el plan **PdNregresar**[14], al cual se le pasan dos parámetros con los valores 100 y 0.5, respectivamente.

b) Ejecutar el comando **CambiarConfiguracion**, que cambiará la configuración del sensor indicado en el primer parámetro (**LM35-1**) a la configuración del fichero cuyo nombre se proporciona en el segundo (**bajoConsumo.cfg**).

2. La tarea 2 se encarga de cambiar las siguientes variables usando el comando **CambiarVariableMision**, cuyos parámetros son la variable que se modifica y su nuevo valor (véase el Cuadro 13.7 y Cuadro 13.7).

   a) Se modifica el parámetro de la misión **VelocidadCrucero**, lo cual se indica con **@VelocidadCrucero**, que pasará a tomar el valor 4 (véase la Sección 13.9 para más detalles sobre la especificación de los parámetros de la misión).

   b) Se modifica el atributo XML referenciado por la sentencia XPath mostrada en el Algoritmo 13.23 (véase la Sección 13.9 y Sección E.6). En este caso se trata de modificar el atributo **medida** de la acción con **id** —identificador— 1 de la tarea con **id** 2 del PdM con **id** 1 (véase la Sección E.6 para más detalles sobre la interpretación de las sentencias XPath). Se le asignará como nuevo valor **temperatura exterior**.

```
/planDeMedicion[@id='1']/tarea[@id='2']/medir[@id='1']/@medida
```

Algoritmo 13.23: Modificar atributo XML con XPath

```
<tarea id="1" nombre="regresarBase">
  <disparadores:disparadores>
    <!-- ... -->
  </disparadores:disparadores>
  <acciones>
    <ejecutarPlan id="1" plan="PdNregresar">
      <parametro id="1" valor="100"/>
      <parametro id="2" valor="0.5"/>
    </ejecutarPlan>
    <ejecutarComando id="1" comando="CambiarConfiguracion">
      <parametro id="1" valor="LM35-1"/>
      <parametro id="2" valor="bajoConsumo.cfg"/>
    </ejecutarComando>
  </acciones>
  <periodoInhibicion valor="2" unidad="hora"/>
</tarea>

<tarea id="2" nombre="reconfigurar">
  <disparadores:disparadores>
    <!-- ... -->
  </disparadores:disparadores>
  <acciones>
    <ejecutarComando id="1" comando="CambiarVariableMision">
        <parametro id="1" valor="@VelocidadCrucero"/>
        <parametro id="2" valor="4"/>
    </ejecutarComando>
    <ejecutarComando id="2" comando="CambiarVariableMision">
        <parametro id="1" valor="/planDeMedicion[@id='1']/tarea[@id='2']
                                 /medir[@id='1']/@medida"/>
        <parametro id="2" valor="temperatura exterior"/>
    </ejecutarComando>
  </acciones>
  <periodoInhibicion valor="10" unidad="minuto"/>
</tarea>
```

Algoritmo 13.24: Acciones del PdS

[14] El tipo de plan se detecta directamente gracias a la información del esqueleto de su especificación.

13.8. Plan de *Log*. Registro del Sistema

El PdL se encarga del registro del sistema, el cual se especifica de acuerdo con la sintaxis de *log4j* (véase el Sección E.8). De acuerdo con [Gülkü, 2002], se trata de una librería para registrar o "*loggear*" aplicaciones desarrolladas originalmente en Java, aunque también está portada a múltiples lenguajes; en el caso de C++ —lenguaje en el que está implementado *Sick***AUV**— se trata de **log4cxx**. La especificación de la misión hará referencia a un PdL como el del Algoritmo 13.25. Aunque la sintaxis de especificación del PdL es la de *log4j*, los datos que se almacenen como parte del registro del sistema aparecerán en el inventario del sistema. Esto permitirá que el resto de planes puedan hacer referencia a estos datos para usarlos —v. g. la acción **enviarDato**, comentada en el Cuadro 13.9, puede enviar un dato que sea un registro o *log* del sistema.

Al usar la especificación de *log4j* para el registro del sistema se consigue un soporte completo para diferentes formatos de representación de los datos registrados; esto se observa en los **appender** especificados en el Algoritmo 13.25. Además, se libera al sistema de la necesidad de desarrollo de un sistema de registro, al usarse la librería de programación de **log4cxx** en el caso de *Sick***AUV**. En la especificación se ha optado por el uso de XML por las ventajas de este lenguaje (véase el Sección E.3) frente al DSL alternativo que también proporciona *log4j* (véase el Sección E.8 para más información).

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE log4j:configuration SYSTEM "log4j.dtd">

<log4j:configuration xmlns:log4j='http://jakarta.apache.org/log4j/'>
  <appender name="stdout" class="org.apache.log4j.ConsoleAppender">
    <layout class="org.apache.log4j.PatternLayout">
      <param name="ConversionPattern" value="%-4r [%t] %-5p %c - %m%n"/>
    </layout>
  </appender>

  <appender name="ficheroPlanoSubsistemaSensorial"
            class="org.apache.log4j.FileAppender">
    <param name="File" value="data/log/subsistemasensorial.txt"/>
    <param name="Inmediateflush" value="true"/>
    <param name="MaxFileSize" value="100KB"/>
    <layout class="org.apache.log4j.PatternLayout">
      <param name="ConversionPattern" value="%d %p - %m%n"/>
    </layout>
  </appender>

  <appender name="ficheroHTMLSubsistemaSensorial"
            class="org.apache.log4j.FileAppender">
    <param name="File" value="data/log/subsistemasensorial.html"/>
    <param name="Append" value="false"/>
    <layout class="org.apache.log4j.HtmlLayout">
      <param name="Title" value="Log del Subsistema de Sensorial"/>
    </layout>
  </appender>

  <logger name="SubsistemaSensorial" additivity="false">
    <level value="info"/>
    <appender-ref ref="stdout"/>
    <appender-ref ref="ficheroHTMLSubsistemaSensorial"/>
  </logger>

  <root>
    <level value ="debug"/>
    <appender-ref ref="stdout"/>
  </root>
</log4j:configuration>
```

Algoritmo 13.25: Ejemplo de PdL

Atributo	Descripción
nombre	Nombre que identifica unívocamente el parámetro de la misión, a nivel de toda la misión —i. e. no sólo a nivel de una tabla de parámetros. Con este nombre se podrá hacer referencia al parámetro de la misión desde los planes o en el control remoto (véase la Definición F.4 y Sección 12.1)
valor	Valor por defecto o inicial que se asigna al parámetro. Debe pertenecer al dominio de los componentes de la misión a los cuales afecta
elemento	Lista de elementos de **1 a N** que indican los componentes de la misión que son afectados por el parámetro, usando sentencias XPath (véase el Cuadro 13.50).

Cuadro 13.49: Atributos usados para especificar un parámetro de la misión (elemento **parametro**)

13.9. Parámetros de la Misión. Modificación dinámica de la misión

Los parámetros de la misión se definirán en tablas de parámetros referenciadas en la especificación de la misión comentada en la Sección 13.1 y facilitarán la modificación dinámica de la misión y sus componentes. Estos parámetros harán referencia a elementos o atributos XML, permitiendo cambiar el valor de los mismos durante la ejecución de la misión; el sistema se encargará de que al cambiar un parámetro se actualizan los planes de la misión que usarán dicho parámetro (véase la Sección 15.4). Con esta funcionalidad se consigue satisfacer el criterio de reconfiguración (véase la Definición F.8) de la misión, estudiado en la Sección 12.2. En el Algoritmo 13.26 se muestra el esqueleto de una tabla de parámetros, que está formado por los siguientes elementos:

parametro Especifica un parámetro de la misión, indicando su nombre y valor por defecto o inicial. Cada parámetro tendrá una lista de elementos **elemento** donde se indica con sentencias XPath los diferentes componentes de la misión a los que afecta (véase el Cuadro 13.50 para más detalles sobre la especificación de cada **elemento** de un parámetro de la misión). Una tabla de parámetros estará compuesta por varios elementos de este tipo. En el Cuadro 13.49 se describen los atributos XML que definen el elemento **parametro**.

```
1 <tablaParametros id="1" nombre="Tabla Parametros 1"
2                  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3                  xmlns:parametros="http://www.parametros.com"
4                  xmlns="http://www.tablaparametros.com"
5                  xsi:schemaLocation="http://www.tablaparametros.com
6                                      ../esquemas/tablaparametros.xsd">
7   <!-- ... -->
8 </tablaParametros>
```

Algoritmo 13.26: Esqueleto de Tabla de Parámetros

Como se comentó en la Sección 13.7 al hablar del comando **CambiarVariableMision**, para modificar la misión se dispone de dos estrategias diferentes:

1. **De forma directa**, usando una sentencia o *query* XPath que referencie un elemento[15] o atributo XML de la especificación de la misión. En la Sección E.13.2 se

[15] Por lo general las sentencias XPath sólo se usarán para referenciar atributos XML de la especificación

Atributo	Descripción
nombre	Sentencia XPath (véase la Sección E.6) que especifica el elemento o atributo XML del árbol XML de los planes de la misión al que afecta el parámetro de la misión, i. e. dicho elemento o atributo tomará el valor especificado en el atributo **valor** del parámetro (véase el Cuadro 13.49).

Cuadro 13.50: Atributos usados para especificar un **elemento** de un parámetro de la misión

explica en detalle la sintaxis de XPath usada para este fin (véase la Sección E.6 para más detalles sobre XPath).

2. Indicando un **parámetro de la misión** definido en una de las tablas de parámetros que contiene la misión. Se indicará el nombre del parámetro de la misión y el nuevo valor. Esto hará que se modifiquen todos los elementos o atributos XML a los que afecta o hace referencia el parámetro de la misión (véase el Cuadro 13.50).

Cualquiera de las dos estrategias de modificación de la misión previamente comentadas puede ser llevada a cabo por:

1. **PdS**, donde mediante el comando **CambiarVariableMision** se indica una sentencia XPath directamente o el nombre de un parámetro de la misión, al que se le asignará un nuevo valor. Esto se especificará con la acción **ejecutarComando** del PdS (véase el Cuadro 13.47, Sección 13.7, respectivamente, y Sección 15.8 para más detalles sobre la gestión del PdS que realiza *Sick***AUV**).

2. **Control remoto**, que enviará comandos remotamente al sistema del vehículo desde un **planificador** (véase la Apéndice A). Los comandos que se envíen tendrán un formato equivalente al de la acción **ejecutarComando** del PdS, el cual se serializará a la hora de transmitirlo al sistema que ejecuta la misión.

La reconfiguración dinámica de la misión obliga a que el sistema compruebe que los cambios realizados son correctos y no producen incompatibilidades anticipables a nivel de la especificación de la misión resultante (véase la Definición 12.8 y Sección 12.3.1). Al producirse la reconfiguración, el sistema se encargará de notificar a todos sus componentes para que se aplique la nueva configuración; esto suele conllevar la desactivación de acciones que estaban en ejecución y la activación de otras nuevas. Además de autorreconfigurarse, el sistema debe comprobar que no se producen problemas a causa de ello. Esta capacidad de reconfiguración también sirve de base para el aprendizaje en la misión (véase la Definición F.9), pero será a nivel interno del sistema —si éste lo soporta— donde se tenga que aplicar la técnica de aprendizaje apropiada.

El Algoritmo 13.27 muestra los siguientes ejemplos de parámetros de la misión:

1. El parámetro de la misión **SensorTemperatura**, que por defecto tiene el valor **TCM345**, referencia el nombre del sensor 1 de la acción **medir** identificada como 1 dentro de la lista de acciones de la tarea 1 del PdM 1. Con este parámetro se

de la misión, pero también será posible referenciar elementos. El problema en este último caso radica en que el valor a asignar para la modificación normalmente será un tipo compuesto o una clase, si se usa POO.

puede modificar la selección del sensor a usar para muestrear una determinada medida (véase la Sección 13.5, para más detalles sobre la especificación del PdM, y Sección 11.1.1 para más información sobre las diferentes formas de seleccionar el sensor para el muestreo).

2. El parámetro de la misión **ProfundidadLimite**, que por defecto tiene el valor 300^{16}, referencia la profundidad máxima de el área 2, definida en el PdN 1 (véase la Sección 13.6 para más detalles sobre la especificación del PdN).

3. El parámetro de la misión **VelocidadCrucero**, que por defecto tiene el valor 4^{16}, referencia el ejes x de la velocidad lineal (**velocidad**/**posicion**/) que se desea que el vehículo mantenga al navegar por el transecto 1, 2 y 3 de la ruta 1 del PdN 3 (véase la Sección 13.6 para más detalles sobre la especificación del PdN). En este caso el parámetro de la misión está compuesto por tres elementos, uno por cada uno de los transectos indicados. Al modificar el valor de este parámetro se modificarán los atributos de los tres transectos, que compartirán el mismo valor.

```
<parametro nombre="SensorTemperatura" valor="TCM345">
  <parametros:elemento nombre="/planDeMedicion[@id='1']/tarea[@id='1']/acciones
                              /medir[@id='1']/sensor[@id='1']/@nombre"/>
</parametro>

<parametro nombre="ProfundidadLimite" valor="300">
  <parametros:elemento nombre="/planDeNavegacion[@id='1']/area[@id='2']
                              /profundidad/@maxima"/>
</parametro>

<parametro nombre="VelocidadCrucero" valor="4">
  <parametros:elemento nombre="/planDeNavegacion[@id='3']/ruta[@id='1']
                              /transecto[@id='1']/velocidad/posicion/@x"/>
  <parametros:elemento nombre="/planDeNavegacion[@id='3']/ruta[@id='1']
                              /transecto[@id='2']/velocidad/posicion/@x"/>
  <parametros:elemento nombre="/planDeNavegacion[@id='3']/ruta[@id='1']
                              /transecto[@id='3']/velocidad/posicion/@x"/>
</parametro>
```

Algoritmo 13.27: Parámetro de la misión

Si observamos el Algoritmo 13.24, veremos como en la tarea 2 se dispone de dos acciones de tipo **ejecutarComando** donde se especifica el comando **CambiarVariableMision**, lo cual permite la modificación de la misión.

1. En la acción 1, el comando **CambiarVariableMision** recibe como parámetro el nombre del parámetro de la misión **VelocidadCrucero**, explicado en el Ítem 3 y especificado en el Algoritmo 13.27; de acuerdo con la sintaxis de especificación de los parámetros de la misión en el PdS (véase la Sección E.13.2), éste se indica con una arroba **@** como prefijo: **@VelocidadCrucero**. Como segundo parámetro se asigna el nuevo valor que éste tomará —4 en el Algoritmo 13.24.

2. En la acción 2, el comando **CambiarVariableMision** recibe como primer parámetro una sentencia XPath que modifica directamente un atributo XML de los planes de la misión. Como se indica en la Sección 13.7, dicha sentencia XPath (véase el Algoritmo 13.23) referencia el atributo **medida** de la acción 1 de la tarea 2 del PdM 1 (véase la Sección 13.5 para más detalles sobre la especificación del PdM). Como segundo parámetro se asigna como nuevo valor **temperatura exterior**.

[16] Los valores de los parámetros de la misión no incluyen unidades, pues éstas ya vendrán indicadas en otros atributos de la misión; si se desea puede modificarse también el atributo que indica la unidad.

Parte III

Sistema

Capítulo 14

Arquitectura

Hybrid architectures are the most used in Autonomous Underwater Vehicles and also in aerial and ground vehicles.

— Marc Carreras Pérez et al.
2003 (Computer Vision and Robotics Group)

El diseño del sistema que va embebido en el vehículo consiste en la adopción de una arquitectura concreta. En la Definición 14.1 y Definición 14.2 se explica el concepto de sistema y arquitectura desde la perspectiva del desarrollo software, respectivamente. El sistema será capaz de ejecutar una misión con la sintaxis de especificación del Capítulo 13. La división de la misión en planes se verá reflejada en la arquitectura propuesta para el sistema (*Sick***AUV**), que dispondrá de subsistemas orientados a planes de la misión concretos. No obstante, la arquitectura del sistema cubrirá aspectos adicionales y se basará en una o varias de las arquitecturas de sistemas robóticos estudiadas.

Definición 14.1 (Sistema). *Es un conjunto integrado de componentes o partes que se interrelacionan. Estos elementos suelen llamarse módulos, que a su vez pueden ser subsistemas, dependiendo si sus propiedades lo definen a su vez o no como un sistema.*

Definición 14.2 (Arquitectura (Sistema)). *La arquitectura de un sistema es una arquitectura de software que define los componentes que llevan a cabo alguna tarea de computación, sus interfaces y la comunicación entre ellos.*
Una arquitectura puede verse como un conjunto de módulos software cooperativos y reusables, que proporciona un diseño estructural a varios niveles de abstracción [Domínguez Brito, 2003].

14.1. Estudio de arquitecturas

El estudio de arquitecturas del sistema recoge una visión de las diferentes alternativas de diseño para sistemas robóticos, centrándose especialmente en los vehícu-

Tipología	Arquitectura	Descripción
Estructura	Deliberativa	Jerárquica, con múltiples niveles, basada en el paradigma de los ciclos de Percepción, Planificación y Acción (SPA)
	Reactiva	Acorta el tiempo de reacción ante estímulos externos, mediante la coordinación de comportamientos de precepción y reacción simples
	Descentralizada	Estructura paralela donde todos los módulos del sistema pueden comunicarse directamente entre ellos, sin niveles de supervisión o intermediarios
	Híbrida	Combinación, mediante varios niveles, de las arquitecturas deliberativa, reactiva y descentralizada. Aprovecha las ventajas de cada una
Módulos	RdP	RdP modulares como elementos constituyentes de la arquitectura
	Tareas	Módulos orientados al manejo de las tareas de la especificación de la misión
	Comportamientos	Módulos que definen divesos comportamientos, que el sistema debe coordinar adecuadamente

Cuadro 14.1: Tipos de arquitecturas robóticas para la implementación del sistema. Tipología de arquitecturas según su estructura o topología y en base a los módulos constituyentes

los de exploración submarina y más concretamente los AUV [Valavanis et al., 1997, Hernández Sosa, 2003, Domínguez Brito, 2003]. Por motivos de claridad se tratan las arquitecturas en base a la tipología documentada en la bibliografía, enumerando algunos casos concretos representativos de cada tipo. La tipología de arquitecturas estudiadas también se centra en el tipo de módulos en los que está basada —v. g. RdP, tareas. Por este motivo, el estudio se realiza tanto a nivel de la estructura general del sistema como en el detalle de los módulos o unidades mínimas de composición del mismo (véase el resumen de la Cuadro 14.1).

Aunque las arquitecturas objeto de estudio suelen ser arquitecturas de control, el objetivo del estudio va más allá, dado que se busca una arquitectura para el diseño de todo el sistema del vehículo. Esto no supone ningún problema, ya que el estudio se ha centrado en arquitecturas implantadas en vehículos submarinos, dando soporte a todos sus elementos —v. g. comunicaciones, almacenamiento, etc., y el propio control percepto-efector.

14.1.1. Arquitecturas Deliberativas o Jerárquicas

Las primeras técnicas de Inteligencia Artificial (IA) en sistemas robóticos adoptaron arquitecturas deliberativas o jerárquicas basadas en el paradigma de los ciclos de Percepción, Planificación y Acción (SPA) [Hernández Sosa, 2003]. Esta arquitectura usa una aproximación *top-down* para dividir el sistema en varios niveles. Los niveles más altos son responsables de los objetivos principales de la misión, y los más bajos son responsables de resolver problemas particulares para cumplir la misión. En la Figura 14.1 se muestra

un ejemplo concreto de arquitectura deliberativa, donde se observa una estructura con diferentes niveles de abstracción, para el caso concreto del AUV **Ocean Voyager II**.

La comunicación entre los componentes de esta arquitectura está serializada, i. e. la comunicación directa sólo es posible entre niveles adyacentes. Los niveles altos envían comandos a los bajos y, como resultado, reciben información perceptual de vuelta de los niveles bajos. El flujo de información decrece según nos acercamos a los niveles más altos de la jerarquía, donde tiene lugar la etapa de razonamiento manejando una reprsentación o modelo del mundo sobre el que se mapean los resultados de los sensores.

La principal ventaja de este esquema es que facilita la verificación de la controlabilidad y estabilidad, i. e. la evaluación del rendimiento de la arquitectura. La desventaja radica en la falta de flexibilidad, de modo que cualquier modificación de alguna funcionalidad requiere cambios significativos en todo el sistema. Además, como no hay una comunicación directa entre los niveles altos y los bajos —v. g. sensores, actuadores—, el tiempo de respuesta es alto y la integración de los sensores compleja. Consecuentemente, esta arquitectura no es capaz de mostrar comportamientos realmente reactivos frente a situaciones imprevistas, v. g. cuando los cambios en el entorno se producen con gran rapidez, se tiende a producir resultados desfasados en el tiempo.

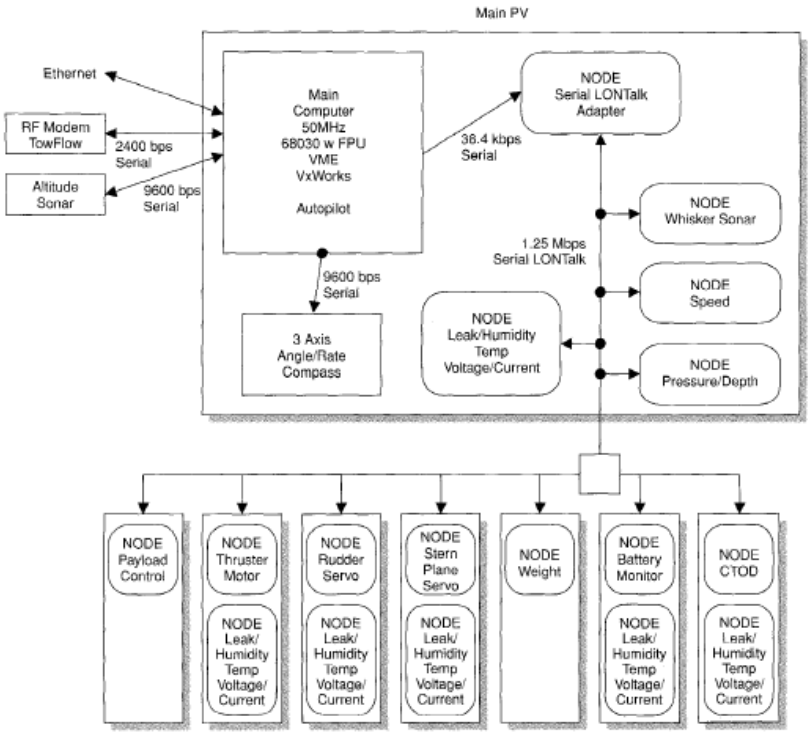


Figura 14.1: Ejemplo de Arquitectura Deliberativa o Jerárquica, para el AUV Ocean Voyager II

De acuerdo con [Valavanis et al., 1997] existen múltiples proyectos en los que se ha diseñado un sistema con una arquitectura deliberativa para su implantación en un AUV. A continuación se comentan los casos de estudio más significativos:

1. **ABE (Autonomous Benthic Explorer)**, desarrollado por el *Woods Hole Ocea-*

nographic Institution, utiliza una arquitectura deliberativa distribuida formada por dos niveles con diferentes capacidades computacionales.

2. **AUVC (Autonomous Underwater Vehicle Controller)**, desarrollado por la *Texas A&M University*, utiliza una arquitectura deliberativa que consta de hasta 18 módulos software, que permite múltiples funcionalidades.

3. **EAVE (Experimental Autonomous VEhicle) III**, desarrollado por el *Marine Systems Engineering Laboratory* de la Universidad de *New Hampshire* y el *Autonomous Undersea Systems Institute*, utiliza una arquitectura deliberativa modular y ordenada temporalmente, que consta de cuatro niveles: tiempo real, sistema, entorno y misión (que es el nivel más alto). Hace uso del controlador de la misión ORCA [Turner and Mailman, 1999].

4. **MARIUS (Marine Utility Vehicle System)**, desarrollado por un equipo multidisciplinar bajo el *Marine Science and Technology (MAST) Programme of the Commission of the European Communities*, utiliza una arquitectura deliberativa distribuida, que se divide en los niveles de organización, coordinación y funcional. El sistema consta de varios módulos dedicados a tareas específicas —v. g. comunicaciones, navegación, guiado, etc.

5. **OTTER (Ocean Technologies Testbed fot Engineering Research)**, desarrollado por el *Monterey Bay Aquarium Research Institute* y la Universidad de *Stanford*, implementa una arquitectura de control a Nivel de Tareas Basada en Objetos (OBTLC), que se modela como una arquitectura deliberativa de tres niveles: *servo*, tareas y organización (que es el nivel más alto).

6. **Ocean Voyager II**, desarrollado por la Universidad de *Florida Atlantic*, utiliza una arquitectura deliberativa distribuida, formada por subsistemas compuestos de sensores y actuadores, con un microcontrolador asociado que actúa como nodo en una red distribuida.

7. Otros, como los experimentos con los siguientes AUV: **LDUUV**, **Martin**, **Twin Burger**.

14.1.2. Arquitecturas Reactivas o basadas en Comportamientos

Como solución a los problemas de las arquitecturas deliberativas se trata de acortar el tiempo de reacción del sistema ante estímulos externos. Las arquitecturas reactivas proponen la construcción de sistemas robóticos mediante la coordinación de comportamientos de percepción o reacción simples [Hernández Sosa, 2003, Carreras Pérez, 2003]. Esta arquitectura consiste en comportamientos que trabajan en paralelo sin un supervisor. Los comportamientos son capas de control y se disparan por la percepción sensorial para realizar acciones. En la Figura 14.2 se muestra un ejemplo concreto de arquitectura reactiva, donde se observan los diferentes comportamientos de los que dispone, para el caso de estudio del AUV **Eric**.

La coordinación de los comportamientos definidos en el sistema la lleva a cabo un elemento denominado **coordinador**, el cual puede aplicar uno de los dos mecanismos primarios de coordinación siguientes [Carreras Pérez, 2003]:

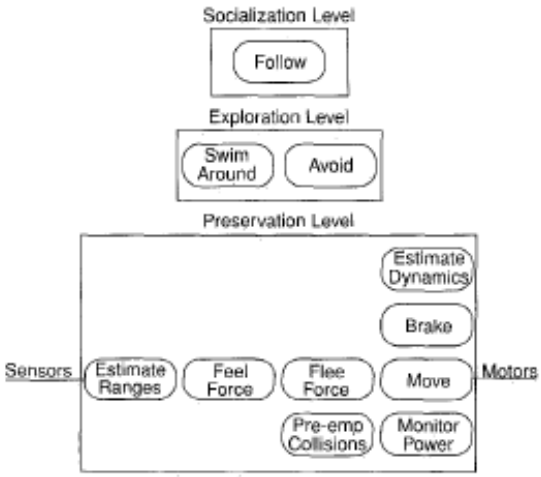


Figura 14.2: Ejemplo de Arquitectura Deliberativa o basada en comportamientos, para el AUV Eric (arquitectura de supresión con diversas mejoras)

Métodos Competitivos La salida es la selección de un solo comportamiento (véase la Figura 14.3 (a)), i. e. el coordinador elige uno solo para controlar el robot.

Métodos Cooperativos La salida es la combinación de todos los comportamientos activos (véase la Figura 14.3 (b)). El coordinador aplica un método que toma las respuestas de todos los comportamientos y genera una salida que es la que controla el robot.

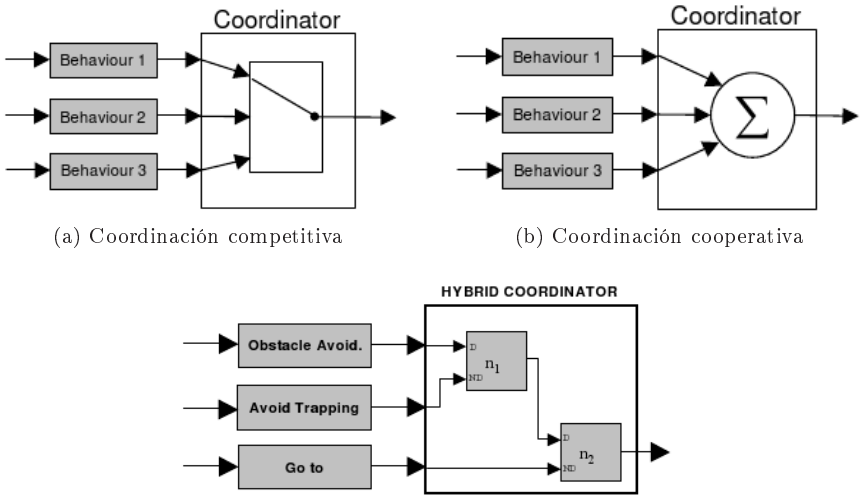
Definición 14.3 (Coordinador). *Componente de una arquitectura de control basada en comportamientos encargado de coordinar las salidas de los diferentes comportamientos activos en el sistema (véase la Figura 14.3 como ilustración de los mecanismos de coordinación principales).*

De acuerdo con el análisis de los mecanismos de coordinación de comportamientos realizado por [Carreras Pérez, 2003] se observa como las ventajas de un método son los inconvenientes del otro y *viceversa*:

Métodos Competitivos Destacan por su modularidad, robustez y reducido tiempo de ajuste de parámetros.

Métodos Cooperativos Destacan por su eficiencia, reducido tiempo de desarrollo y simplicidad.

Esto hace que en la mayoría de los casos se adopte una arquitectura híbrida, como la mostrada en la Figura 14.3 (c), que intenta beneficiarse de las ventajas de cada uno de los métodos anteriores. En este caso, el coordinador puede funcionar de forma competitiva o cooperativa según convenga. Aunque el **coordinador** produce la salida final en base a las salidas individuales de cada comportamiento, éstos siempre estarán operando paralelamente. Los datos y el control están distribuidos por todas las capas de la arquitectura, pues cada una procesa su propia información —v. g. sensorial, comandos— sin que haya un modelo o estructura de datos global en el sistema.



(a) Coordinación competitiva

(b) Coordinación cooperativa

(c) Coordinación híbrida

Figura 14.3: Métodos de coordinación de comportamientos

Las principales ventajas de este esquema son la flexibilidad, robustez y la baja carga computacional [Valavanis et al., 1997]. Los ejemplos más representativos son la arquitectura de supresión (*subsumption*) [Brooks, 1985] y los esquemas motores (*motor schema*) [Balch and Arkin, 1989]. Esta arquitectura muestra un comportamiento verdaderamente reactivo.

Las desventajas de esta aproximación son la dificultad de sincronización y temporización entre comportamientos, la complejidad del sistema cuando se tienen muchos comportamientos y la falta de un control en un nivel alto. Esto dificulta la verificación del sistema y las pruebas de estabilidad y correctitud. Por ello se suelen usar versiones modificadas de esta arquitectura con una funcionalidad de control, v. g. formalización, tabla de estado, etc.

De acuerdo con [Valavanis et al., 1997] existen varios proyectos en los que se ha diseñado un sistema con una arquitectura reactiva para su implantación en un AUV. A continuación se comentan los casos de estudio más significativos:

1. **Eric**, desarrollado por el *Key Center Robotics Laboratory* y la *University of Technology, Sydney*, utiliza una arquitectura reactiva de supresión con ciertas mejoras y tres niveles de competencia: **preservación**, **exploración** y **socialización**.

2. **Odyssey II**, desarrollado por el *Massachusetts Institute of Technology*, adopta una arquitectura reactiva de supresión con capa de control basada en estados, que comanda los actuadores para alcanzar el estado deseado.

3. **Sea Squirt**, desarrollado por el *Massachusetts Institute of Technology*, utiliza una arquitectura reactiva de supresión mejorada con una capa de control basada en estados, dividida en dos niveles: tabla de estados en el nivel superior y estructura de control por capas en el inferior.

4. **URIS (Underwater Robotic Intelligent System)**, desarrollado por la Universidad de Girona, utiliza una arquitectura reactiva con un **coordinador** híbrido

y aprendizaje por refuerzo [Carreras Pérez, 2003]. También se ha implantado en otros AUV: **GARBI**, **RAO II**.

5. Otros, como los experimentos con los siguientes AUV: **Umihico**.

14.1.3. Arquitecturas Descentralizadas o Heterárquicas

Las arquitecturas descentralizadas o heterárquicas, a diferencia de las deliberativas, se caracterizan por una estructura paralela donde todos los módulos del sistema pueden comunicarse directamente entre ellos, sin niveles de supervisión ni intermediarios [Valavanis et al., 1997]. En la Figura 14.4 se muestra un ejemplo concreto de arquitectura híbrida donde se dispone de una capa descentralizada, donde se observa el paralelismo de los módulos, para el caso de estudio del AUV **ODIN**.

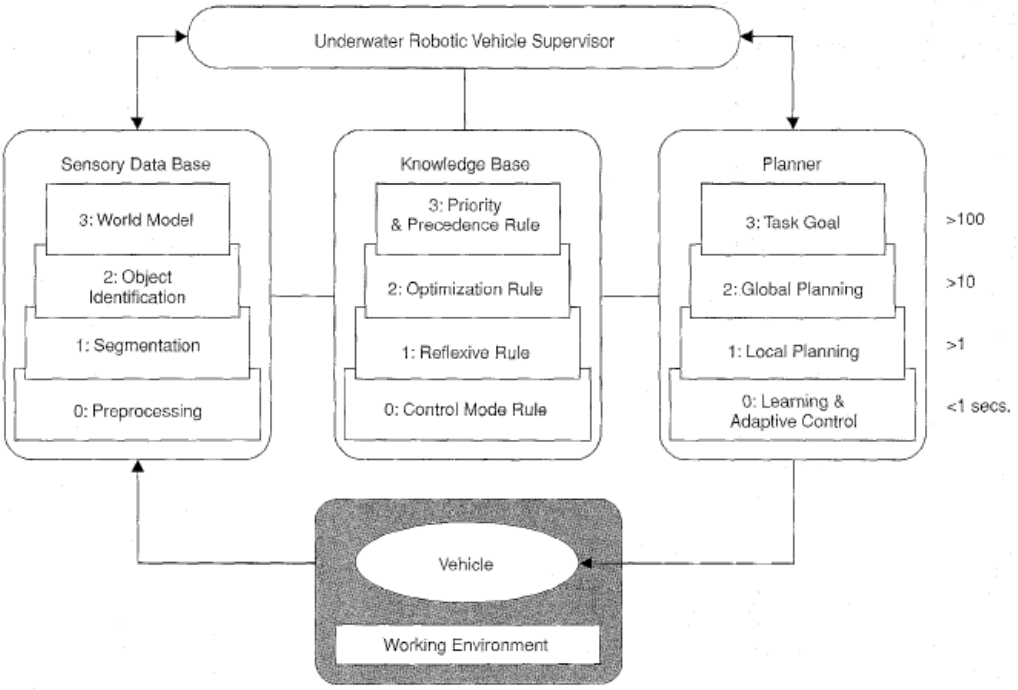


Figura 14.4: Ejemplo de Arquitectura Descentralizada o Heterárquica, para el AUV ODIN (arquitectura híbrida que en el nivel bajo está formada por bloques descentralizados)

Las principales ventajas de este esquema son la flexibilidad y la baja carga de las comunicaciones entre los módulos del sistema. Además, puede usar el procesamiento paralelo, ya que el conocimineto y la información sensorial puede ser consultada fácilmente por cualquier módulo. Por contra, la principal desventaja son la falta de supervisión, de modo que la comunicación entre módulos puede ser muy intensiva y la controlabilidad del sistema se convierte en un problema.

De acuerdo con [Valavanis et al., 1997] no existe ningún ejemplo puro de esta arquitectura, si bien hay algunas arquitecturas híbridas que la incorporan:

1. **ODIN (Omni-Directional Intelligent Navigator) II**, desarrollado por el *Autonomous Systems Laboratory* de la Universidad de Hawaii, utiliza una arquitectura híbrida basada en la deliberativa y descentralizada. Dispone de tres bloques de funciones separados (**base de datos sensorial**, **base de conocimiento** y **planificador**), que permiten un incremento de la inteligencia del sistema con el incremento del número de capas y posibilitan el procesamiento paralelo.

14.1.4. Arquitecturas Híbridas

Las arquitecturas híbridas son una combinación de varias arquitecturas: **deliberativa**, **reactiva** y **descentralizada**. La combinación de las arquitecturas intenta aprovechar las ventajas de cada una. El sistema se divide en varios niveles, donde el nivel superior usa una arquitectura deliberativa para implementar la funcionalidad de estrategia y nivel de misión, y el nivel inferior usa una arquitectura reactiva o descentralizada para controlar los subsistemas *hardware*. En la Figura 14.5 se muestra un ejemplo concreto de arquitectura híbrida (deliberativa y reactiva), para el caso de estudio del AUV **Phoenix**.

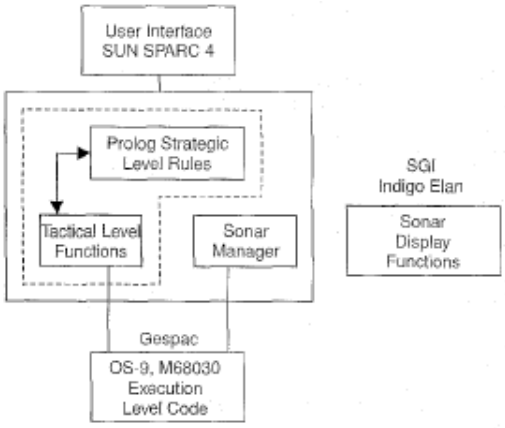


Figura 14.5: Ejemplo de Arquitectura Híbrida, para el AUV Phoenix (arquitectura híbrida de tres niveles, que es deliberativa entre los niveles y reactiva a nivel de ejecución)

La combinación de arquitecturas no es en absoluto simple, puesto que hay que poner en correspondencia dispositivos robóticos eminentemente continuos y numéricos con técnicas de IA de naturaleza discreta y simbólica. El problema radica en determinar cuándo y durante cuánto tiempo deliberar y cuándo reaccionar [Hernández Sosa, 2003]. Por lo general, las arquitecturas híbridas suelen definir tres niveles (de menor a mayor):

1. **Reactivo o de comportamientos**, con las habilidades básicas o comportamientos primitivos, cuya combinación determina las capacidades funcionales del sistema.
2. **Ejecutivo o de secuenciación**, encargado de seleccionar qué comportamientos reactivos son los más adecuados mediante un **coordinador** (véase la Definición 14.3).
3. **Deliberativo o de planificación**, encargado de establecer los pasos para la realización de la misión a largo plazo. Puede funcionar de forma síncrona o asíncrona en relación con las peticiones del nivel ejecutivo o reactivo.

De acuerdo con [Valavanis et al., 1997] existen múltiples proyectos en los que se ha diseñado un sistema con una arquitectura híbrida para su implantación en un AUV. A continuación se comentan los casos de estudio más significativos:

1. **ODIN (Omni-Directional Intelligent Navigator) II**, desarrollado por el *Autonomous Systems Laboratory* de la Universidad de Hawaii, utiliza una arquitectura híbrida basada en la deliberativa y descentralizada. El nivel de supervisión maneja los parámetros de la misión en base a la información del nivel inferior, que dispone de tres bloques de funciones separados (**base de datos sensorial**, **base de conocimiento** y **planificador**), que siguen una estructura descentralizada.

2. **Phoenix**, desarrollado en el *Naval Postgraduate School, Monterey*, utiliza una arquitectura híbrida de tres niveles (de mayor a menor): **estratégico**, **táctico** y de **ejecución**. la arquitectura deliberativa se emplea entre los tres niveles y la reactiva se aplica al nivel de **ejecución**. El nivel **estratégico** usa Prolog como lenguaje de especificación de la misión, basado en reglas, mientras que el **táctico** sirve de interfaz con el nivel de **ejecución** en tiempo real.

3. **SAUVIM (Semi-Autonomous Underwater Vehicle for Intervention Missions)**, desarrollado por la Universidad de Hawaii, usa la arquitectura híbrida *Intelligent Task-Oriented Control Architecture* (ITOCA), que se organiza en tres niveles (de menor a mayor): **ejecución**, **control** y **planificación**.

4. **REDERMOR**, desarrollado por el proyecto DCE/GESMA, utiliza una arquitectura híbrida compuesta de varios módulos, encargados de tareas de planificación, guiado, control y monitorización de la ejecución en los niveles más altos de la arquitectura, mientras que en los inferiores se trata la ejecución reactiva de procedimientos específicos.

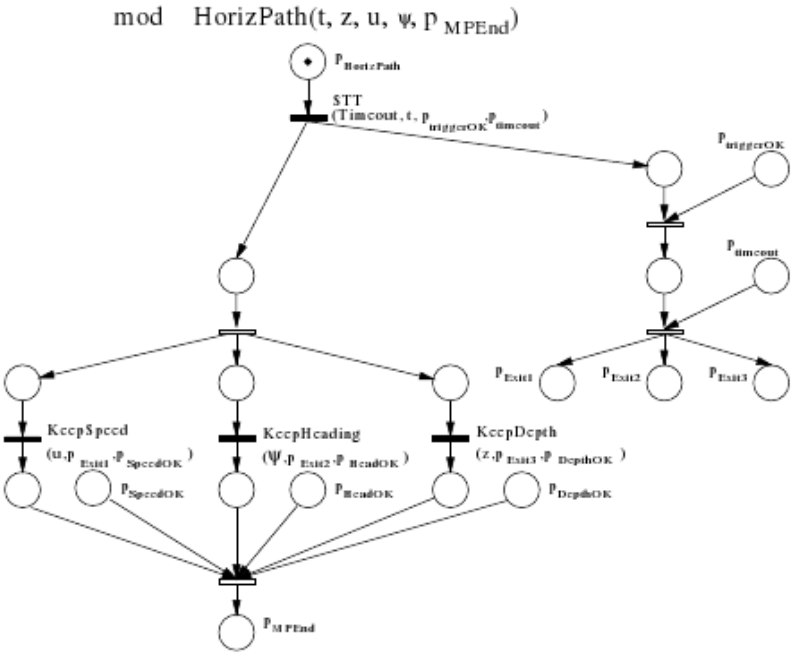
5. Otros, como los experimentos con los siguientes AUV: **ARCS**, **Aurora**, **Dolphin**, **PURL**, **PURL II**, **Theseus**, **Typhlonus**.

14.1.5. Módulos basados en Redes de Petri

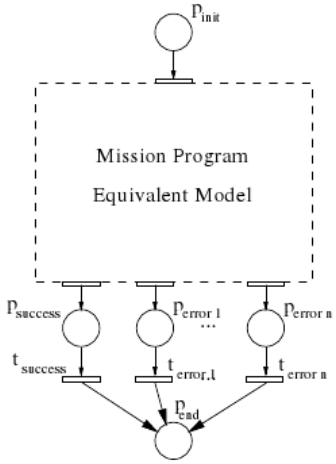
Los módulos de la arquitectura del sistema pueden modelarse mediante redes de Petri (RdP), tal y como se muestra en la Figura 14.6. Las RdP, explicadas en el Apéndice C, ofrecen un formalismo y una metodología contrastada para la especificación de las funcionalidades del sistema. Éste dispondrá de un motor de interpretación de RdP que trabajará sobre la representación textual de la misma. En este sentido se observa la estrecha relación entre la especificación de la misión y el propio sistema cuando se emplean RdP, aspecto ya comentado en el Sección 12.1.1 y que la Figura 14.6 (b) ilustra claramente, pues la especificación de la misión se integrará en el sistema como una RdP modular dentro de la RdP del sistema.

Por lo general, se hace uso de RdP modulares, pues facilitan la estructuración del sistema en diferentes niveles de abstracción (véase la Sección C.2.2). De acuerdo con la bibliografía consultada [Valavanis et al., 1997, Ramalho Oliveira et al., 1996, Barrouil and Lemaire, 1998, Ridao et al., 2005], se usan las RdP como módulos en arquitecturas deliberativas —o híbridas que incluyen este tipo de arquitectura—, sin constituir ello una imposición. No obstante, las RdP se construyen a partir de primitivas que

constituyen las funcionalidades más básicas del vehículo, por lo que las características de éstas también definirán el tipo de arquitectura del nivel más bajo del sistema. En la Figura 14.6 (a) se muestra una RdP que modela un procedimiento de la misión usando CORAL, lo cual puede constituir un ejemplo de la posibilidad de que diferentes tipos de módulos coexistan en un mismo sistema.



(a) Procedimiento de especificación de la misión mediante RdP



(b) Programa de representación para ejecución de la misión, mediante RdP modulares

Figura 14.6: Ejemplo de Arquitectura con módulos basados en RdP, para el AUV MARIUS

Las RdP son una herramienta especialmente útil en el modelado de sistemas de even-

tos discretos asíncronos y concurrentes, como es el caso de los sistemas robóticos. Esto, unido al hecho de que las RdP modulares permiten el modelado a través de diferentes planos jerárquicos, ofrece modularidad, robustez y facilidad de monitorización al sistema. El uso de RdP no impide que el sistema use otro tipo de módulos simultáneamente —v. g. tareas o comportamientos—, ya que permite que dichos módulos se modelen o implementen mediante RdP.

De acuerdo con [Ramalho Oliveira et al., 1996, Ramalho Oliveira et al., 1998, Barrouil and Lemaire, 1998, Barbier et al., 2001, Duarte Oliveira, 2003] existen varios proyectos en los que las RdP son los módulos constituyentes de la arquitectura del sistema. A continuación se comentan los casos de estudio más significativos:

1. **MARIUS (Marine Utility Vehicle System)**, desarrollado un equipo multidisciplinar bajo el *Marine Science and Technology (MAST) Programme of the Commission of the European Communities*, utiliza RdP como módulos de una arquitectura deliberativa. Hace uso de las herramientas CORAL y ATOL. CORAL es un entorno de programación con herramientas para la creación de RdP y un motor para su interpretación en el sistema. ATOL es un entorno de programación para el lenguaje ATOL, que se define como imperativo, reactivo síncrono [Ramalho Oliveira et al., 1996], y que permite el análisis y ejecución del programa de la misión.

2. **REDERMOR**, desarrollado por el proyecto DCE/GESMA, utiliza RdP como módulos de una arquitectura híbrida. Hace uso de las herramientas del software ProCoSa. Los módulos del sistema se encargan de la planificación de rutas, planificación de la misión, guiado, control y monitor de ejecución, fundamentalmente. En el nivel de ejecución se hace uso del motor de interpretación de RdP conocido como ProCoSa, que trabaja con un representación textual de la RdP en lenguaje Lisp.

14.1.6. Módulos basados en Sub-objetivos y Tareas

El modelado del sistema mediante módulos de tareas es bastante común, ya que éstas constituyen una unidad mínima de funcionalidad y ejecución, tanto de la misión como del sistema. La arquitectura del sistema suele ser híbrida en estos casos, basada en la deliberativa y la reactiva. Los niveles superiores son deliberativos y se encargan del control, la supervisión y planificación de las tareas a realizar, mientras que en los niveles inferiores son reactivos y su competencia es la realización de las tareas *per se*, interactuando con el entorno —i. e. percepción o actuación.

En la Figura 14.7 se muestra el módulo de tarea, o *Task Module*, de la arquitectura ITOCA [Ridao et al., 2005, Roberts et al., 2003]. A partir de éste se construye toda la arquitectura, donde también se dispone de módulos de supervisión de tareas, o *Task Supervisor*, entre otros elementos de planificación para terminar de constituir el nivel de planificación (véase la Figura 12.6). En general, las arquitecturas basadas en tareas siguen éste esquema, donde puede variar la estructura de la arquitectura que gestiona la ejecución de las tareas.

Las propias características de una tarea facilitan la ejecución concurrente o paralela de las mismas en el sistema. En la Figura 12.7 se ilustran algunos ejemplos típicos de tareas exploración (*survey*), concretamente en la arquitectura ITOCA. Mientras el nivel inferior

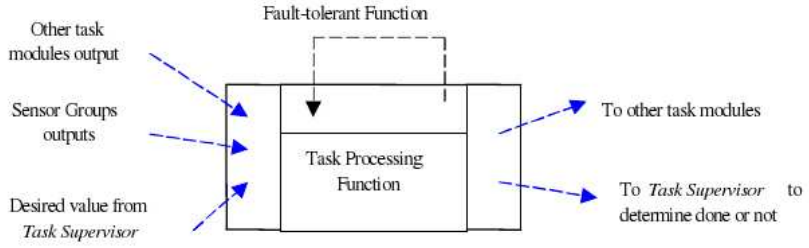


Figura 14.7: Ejemplo de módulo definido como una tarea. *Task Module* de la arquitectura ITOCA

y reactivo se encarga de la ejecución de las tareas, los niveles superiores y deliberativos realizan la conexión entre el software de aplicación y los módulos de tareas usando un manejador de tareas, que permitirá crear y destruir tareas, controlar la sincronización y la comunicación entre ellas, etc.

De acuerdo con [Ridao et al., 2005, Roberts et al., 2003] existen varios proyectos en los que la arquitectura del sistema dispone de módulos de tareas. A continuación se comentan los casos de estudio más significativos:

1. **SAUVIM (Semi-Autonomous Underwater Vehicle for Intervention Missions)**, desarrollado por la Universidad de Hawaii, usa la arquitectura híbrida *Intelligent Task-Oriented Control Architecture* (ITOCA), utiliza módulos de tareas como parte de una arquitectura híbrida de tres capas (de menor a mayor): **ejecución**, **control** y **planificación**. En esta arquitectura la estabilidad y controlabilidad son fácilmente verificables, si bien el tiempo de respuesta es grande. Este problema se solventa con un Bus de Datos Sensoriales (SDB) que mitiga el problema gracias a que permite la comunicación directa entre el nivel bajo, medio y alto de la arquitectura. Así, las señales urgentes pueden ser manejadas en tiempo real, mientras que el resto se filtrarán y procesarán para obtener datos limpios que se tratarán en niveles superiores y deliberativos [Roberts et al., 2003].

14.1.7. Módulos basados en Comportamientos

Las arquitecturas reactivas son las que muestran un sistema compuesto por módulos basados en comportamientos —de ahí que también se las conozca como arquitecturas basadas en comportamientos (véase la Sección 14.1.2). Los comportamientos modelan las acciones que deben realizarse en base a un conjunto de datos o información perceptual del entorno (véase la Definición 12.4). Lo explicado en la Sección 14.1.2 es perfectamente válido para este tipo de arquitectura —atendiendo a la clasificación según los módulos que la componen—, ya que está basada en comportamientos. En la Figura 14.8 se muestra un diagrama donde se observa el proceso de coordinación de varios comportamientos, para el caso concreto de la propuesta de [Carreras Pérez, 2003].

De acuerdo con [Carreras Pérez, 2003, Carreras Pérez et al., 2003] existen varios proyectos en los que la arquitectura del sistema dispone de módulos de comportamientos. A continuación se comentan los casos de estudio más significativos:

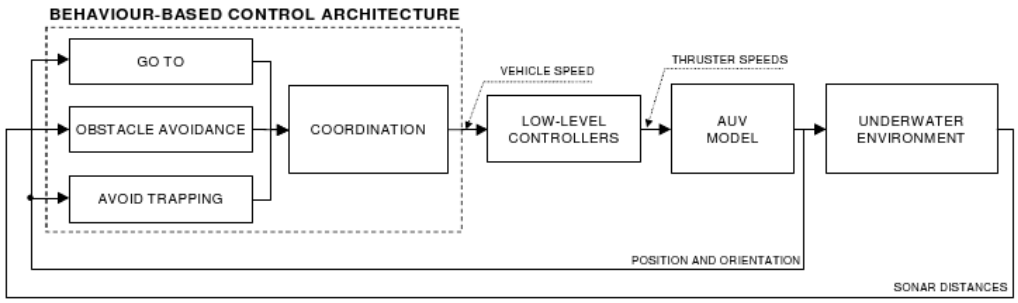


Figura 14.8: Ejemplo de arquitectura con módulos basados en comportamientos. Proceso de coordinación de comportamientos [Carreras Pérez, 2003]

1. **URIS (Underwater Robotic Intelligent System)**, desarrollado por la Universidad de Girona, utiliza una arquitectura reactiva con un **coordinador** híbrido y aprendizaje por refuerzo [Carreras Pérez, 2003]. El **coordinador** puede trabajar de forma cooperativa o competitiva según determine el nivel superior y deliberativo, en base al estado del modelo del entorno que mantiene.

2. **GARBI**, desarrollado por la Universidad de Girona, utiliza esquemas motores o *motor schemas*, que es una arquitectura reactiva o basada en comportamientos concreta, caracterizada por la definición de campos potenciales tridimensionales, como se muestra en la Figura 14.9. Esta arquitectura se caracteriza por la simplicidad, robustez y buenos resultados en las tareas de navegación [Carreras Pérez et al., 2003].

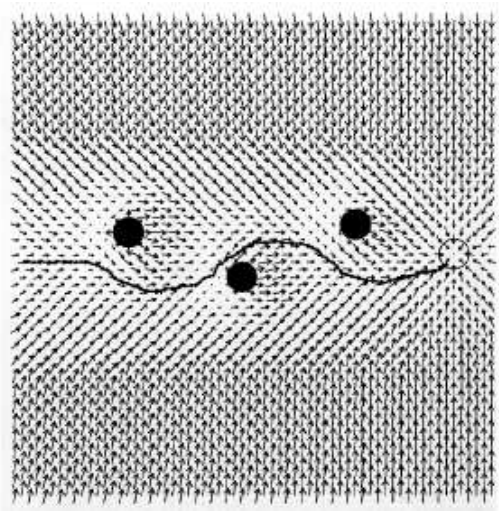


Figura 14.9: Esquemas motores. Campo potencial [Carreras Pérez et al., 2003]

14.1.8. Conclusiones

De acuerdo con la bibliografía consultada queda patente que desde el punto de vista de la estructura de la arquitectura, la mejor alternativa es la híbrida. Sólo esta arquitectura

combina las ventajas de las aproximaciones deliberativa y reactiva —y, en menor medida, la descentralizada. Por ello no parece necesario realizar un análisis para determinar la mejor opción, ya que éste ya ha sido realizado y demostrado en numerosos estudios y experimentos [Valavanis et al., 1997, Hernández Sosa, 2003, Domínguez Brito, 2003, Carreras Pérez, 2003].

Dentro de las arquitecturas híbridas lo habitual y más apropiado suele ser disponer de los tres niveles comentados en la Sección 14.1.4: **reactivo**, **ejecutivo** y **deliberativo**. Conforme a las competencias de cada uno de estos niveles de la arquitectura, el sistema interpretará la misión y planificará a largo plazo en el nivel **deliberativo**, los comandos o acciones que deben realizarse comunicarán mediante el nivel **ejecutivo** al nivel **reactivo**, que es el encargado de ejecutarlos exhibiendo las habilidades o comportamientos primitivos del vehículo.

Sin embargo, desde el punto de vista de los módulos de la arquitectura, la elección no está tan clara y no será tan determinante en la calidad del sistema final. Así, todas las alternativas serán igualmente válidas, aunque cada una tiene sus ventajas e inconvenientes. La opción adoptada dependerá de factores que no estarán tan relacionados con el rendimiento del sistema, como ocurre con la estructura de la arquitectura. Algunos de estas factores o criterios de evaluación son los siguientes:

Modularidad

Coste de Implementación

Flexibilidad

Robustez

Monitorización

Reconfiguración

Aprendizaje

Aparte de esta lista de criterios de evaluación, resulta importante el soporte que tiene que dar el sistema para poder realizar la misión mostrada en el Capítulo 13. Dado que se trata de una misión que adopta una arquitectura de especificación de misiones basada en tareas, en principio se considera que lo más adecuado es que el sistema también emplee este tipo de módulos como elementos básicos de sus arquitectura.

14.2. Arquitectura Propuesta. *Sick*AUV

Para el sistema del AUV se propone un arquitectura híbrida con módulos basados en tareas, que se recibe el nombre de *Sick***AUV** y se explica detalladamente en el Capítulo 15. El sistema se estructura en varios subsistemas encargados de un tipo de tareas específicas de la misión (véase la Capítulo 11), donde cada uno se subdivide en componentes CoolBOT (véase el Sección E.1).

Capítulo 15

*Sick*AUV. Sistema Integrado de Control para un AUV

I said that the oceans were sick
but they're not going to die.
There is no death possible in the oceans
—there will always be life— but they're getting sicker every year.

— Jacques-Yves Cousteau
1996 (French naval officer, inventor, explorer and researcher)

AUVs are unmanned, underwater robots akin to the Exploration Rover NASA uses on Mars.
AUVs operate independent of humans, using their sensors to create maps of the ocean floor, record environmental information, and sense what humans have left behind.

— The National Oceanic and Atmospheric Administration (NOAA)
2008 (United States Department of Commerce)

En lo sucesivo se usarán las abreviaturas del Cuadro 15.1 para referirnos a los diferentes subsistemas que constituyen el sistema *Sick***AUV**.

Se realizará un diseño detallado de los componentes desde arriba hacia abajo, i. e. desde los subsistemas hasta los elementos más simples de cada uno de ellos. Por tanto, incluso los subsistemas se tratarán como componentes de CoolBOT [Domínguez Brito, 2003], en

Abreviatura	Subsistema
act	Actuador
alm	de Almacenamiento
com	de Comunicación
gui	de Guiado
nav	de Navegación
sen	Sensorial
sup	de Supervisión
cen	Central

Cuadro 15.1: Abreviaturas de los Subsistemas del sistema *Sick*AUV

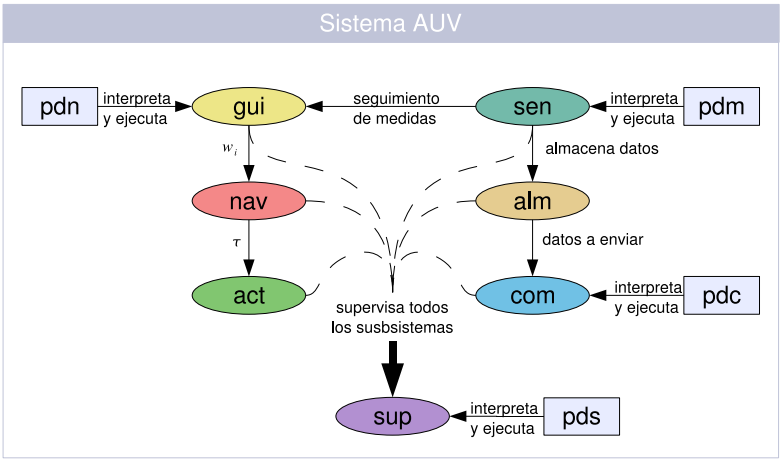


Figura 15.1: Sistema AUV con las relaciones entre sus subsistemas

concreto serán componentes compuestos[1]. En el cuadro 15.1 se listan las abreviaturas adoptadas para cada uno de los subsistemas, que a su vez se modelarán como componentes compuestos CoolBOT, tal y como se ha dicho. En la figura 15.1 se muestra el sistema del AUV —que también podrá ser en sí mismo un componente compuesto de CoolBOT— con las relaciones entre sus subsistemas.

A partir del diagrama de relaciones entre subsistemas de la figura 15.1 cabe citar las siguientes aclaraciones:

1. Debe incluir el plan de almacenamiento (PdA) para el subsistema de almacenamiento, ya que en el diseño final se considera que lo especificado en el plan de medición (PdM) no debe almacenarse necesariamente. Esto dota al sistema de mayor versatilidad, pues es posible indicar que simplemente se mida una medida, que simplemente se almacene o que se mida y almacene. Evidentemente, si sólo se indica que se almacene, pero ésta no se mide, como resultado no se habrá almacenado nada.

2. En contra de lo antes dicho, al realizar el diseño detallado de los subsistemas, como se observa en la figura 15.1, no se considera necesario usar el Subsistema de

[1] Los componentes compuestos o *compound components* de CoolBOT son componentes que contienen internamente a otros componentes.

Almacenamiento para el envío de datos del Subsistema Sensorial al de Guiado.

3. Por otro lado, el Subsistema Central no aparece en la figura 15.1 debido a que no se detecta ninguna comunicación interna de éste con el resto de subsistemas. Esto puede ser indicativo de que quede asimilado por parte del componente CoolBOT del sistema del AUV.

4. En principio se intentará evitar que el *proyecto* de desarrollo no sea monolítico, ya que si alguien deseara cambiar algún componente, lo ideal es que no tenga que trabajar sobre todo el sistema. Por ello, sin dejar de usar CoolBOT, se plantea la opción de hacer —al menos— los subsistemas como si fueran componentes *remotos* de CoolBOT.

A continuación se tratará cada subsistema como un componente compuesto de CoolBOT donde, si se requiere comunicación externa o remota, se incluirá el componente servidor de CoolBOT —y el proxy. Esto se aplicará, por tanto, a todos los subsistemas, ya que se pretende que se pueda acceder remotamente a cualquiera de los subsistemas. Por claridad, no se mencionan en el estudio de los subsistemas. Además, en principio sólo se enumeran y definen los componentes —atómicos[2]— para posteriormente analizar internamente cada uno —indicando fundamentalmente los observables, controlables y el autómata de usuario.

En el caso del Subsistema Actuador y Sensorial se hacen las siguientes consideraciones, ya que se hará uso de *Player* [Gerkey et al., 2006].

1. El uso de actuadores y sensores se sitúa por encima de la HAL que proporciona *Player*, por lo que se usarán las interfaces que éste define como la base para la creación de componentes atómicos de CoolBOT para ellos —u otro tipo de elementos software.

2. Tanto para el Subsistema Actuador como Sensorial se debe conocer a priori la lista de actuadores y sensores para poder definir claramente cuantos componentes se desarrollarán en CoolBOT —a parte de su adaptación a *Player*. Adicionalmente, se debe analizar el coste de añadir un nuevo actuador o sensor en CoolBOT, para un desarrollador; se trataría de incluir una nueva interfaz de *Player* o una aún no tratada en el sistema del AUV, ya que la creación de nuevos drivers se abstrae con la HAL.

15.1. Servicios

El núcleo del sistema *Sick***AUV** se basa en un modelo de solicitud de servicios entre los diferentes componentes CoolBOT que lo componen. Este sistema utiliza un mecanismos de mensajes de solicitud de servicios entre un componente y otro. El solicitante actúa como cliente del servicio y el que lo ofrece como suministrador.

Cuando el suministrador recibe la solicitud de realizar un servicio, creará un pedido. La finalidad del pedido es poder gestionar adecuadamente el servicio prestado y permitir la monitorización de su estado. En este sentido, el suministrador enviará mensajes de informe en los que se indicará el nivel de realización del pedido con el que se está

[2]Los componentes atómicos o *atomic components* de CoolBOT son aquellos no divisibles internamente en otros componentes; en principio proporcionan una funcionalidad bien diferenciada.

tratamitando el servicio solicitado. Esto también permite la notificación de errores y la indicación de la finalización.

Definición 15.1 (Comando). *Funcionalidad atómica bien definida con un nombre unívoco y una serie de parámetros opcionales. El ámbito de un comando puede ser relativo a todo el sistema o interno a un subsistema de la arquitectura del SickAUV.*

Definición 15.2 (Servicio). *Funcionalidad que suministra el sistema. Permite comandar el AUV y la ejecución de tareas internamente —v. g. ejecución de misiones. Un servicio controla la ejecución de un comando (véase Definición 15.1) desde su solicitud hasta su terminación.*

Los servicios sirven para la gestión de los comandos que ejecuta el sistema. De es este modo, los servicios ofrece un sistema de gestión apropiado para la ejecución de los comandos. Los comandos, por tanto, son las funcionalidades que en última instancia se ofrecen. Se trata de la codificación de las tareas que realiza el vehículo o el sistema, v. g. medir muestras de una medida, almacenar o comunicar dichas muestras, etc.

Desde el punto de vista de los planes de la misión, las tareas especificados en los mismos disponen de acciones, las cuales se corresponden en última instancia con los servicios que ofrecen determinados componentes CoolBOT del sistema; éstos, a su vez, están codificados mediantes comandos, como se explicó anteriormente.

Los subsistemas que se explican en la siguientes secciones se fundamentan en el uso de este sistema de gestión de servicios. Cada subsistema ofrecerá el conjunto de todos los servicios que ofrecen sus componentes CoolBOT internos. La gestión de servicios entre subsistemas también está soportado y se fundamenta en el enrutamiento de las solicitudes en base a una base de datos que relaciona los nombres de los servicios con los nombres de los componentes que ofrecen los mismos.

15.2. Subsistema Actuador

En este subsistema todo partirá de las entradas de control de los impulsores, definidas en el vector τ —se trata de una orden generada internamente por el Subsistema de Navegación (véase la Sección 15.6). Estas entradas dependerán de los impulsores empledos y de su pose o disposición —posición x, y, z y orientación ϕ, θ, ψ, respecto al sistema de referencia BODY, con origen en el centro de gravedad CG, de flotabilidad CB o bien en otro punto del vehículo O, elegido por conveniencia— en el vehículo.

En la Figura 15.2 se muestra el componente CoolBOT compuesto que modela el subsistema actuador. Con el nivel de diseño alcanzado, internamente sólo se identifica un componente CoolBOT, denominado **actuador**, tal y como se muestra en el diagrama de la estructura interna del subsistema (véase la Figura 15.3). Aparte de las bandejas de entrada y salida para soportar los mensajes de gestión de servicios, dispone de un puerto de entrada para recibir los comandos de control, cuyo nombre es **comando**.

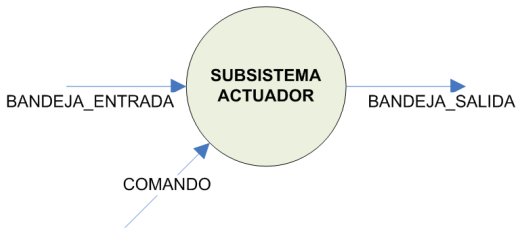


Figura 15.2: Subsistema Actuador. Componente Compuesto

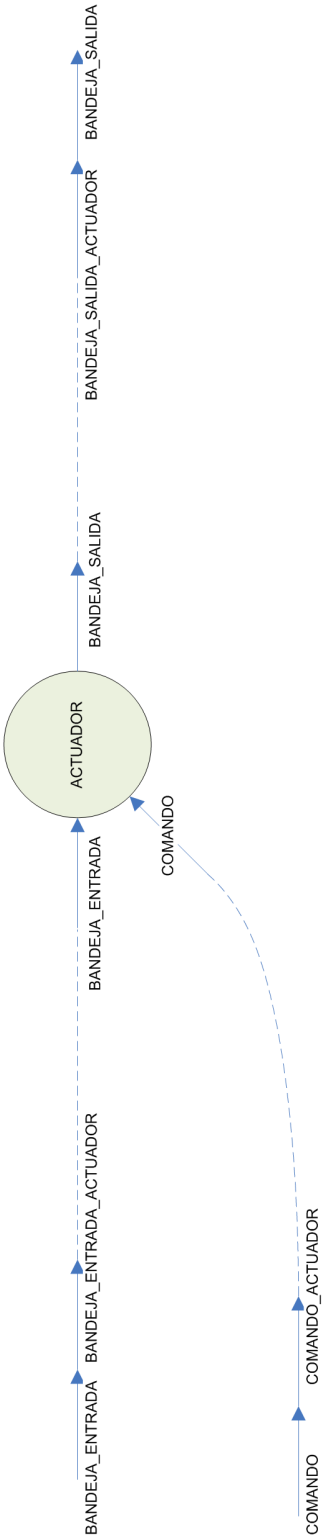


Figura 15.3: Subsistema Actuador. Componentes Internos

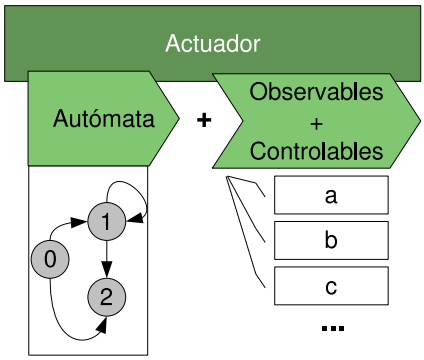


Figura 15.4: Actuador: Autómata de Control Homogéneo y Observables

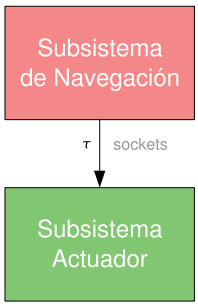


Figura 15.5: Acciones del actuador

En la siguiente lista se enumeran las necesidades de este subsistema.

1. Se debe aplicar una gestión de los actuadores, que será similar a la aplicada a los sensores en el Subsistema Sensorial (ver sección **??**). Por tanto, como se muestra en la figura 15.4, un actuador se modelará con un autómata de control homogéneo —que podría ser el mismo que el usado para los sensores— y una serie de observables y controlables. El actuador será un proceso al que el resto de componentes y subsistemas se podrán comunicar mediante algún mecanismo de comunicación, tratándose de un cliente —dentro de la arquitectura cliente/servidor—, ya que al ser un actuador sólo servirá a las órdenes de un proceso, que podrá tomar de un almacén, como se muestra en la figura 15.5 y en el siguiente punto. No obstante, si no se usa ningún almacén y se controla directamente, el actuador debe ser un servidor, aunque sólo aceptará un cliente en un momento dado.

2. Se asociará un almacén al actuador o se controlará directamente el actuador, ya que normalmente no es necesario que se almacenen datos. En la figura 15.5 se muestra como el actuador se controla directamente con las órdenes τ del Subsistema de Navegación, sin que se emplee un almacén intermedio. Por ello, el actuador será un servidor que aceptará un único cliente. Finalmente, el actuador accionará el impulsor, a través de la API del *driver* del mismo. En este sentido, se podrá emplear la arquitectura proporcionada por *Player*.

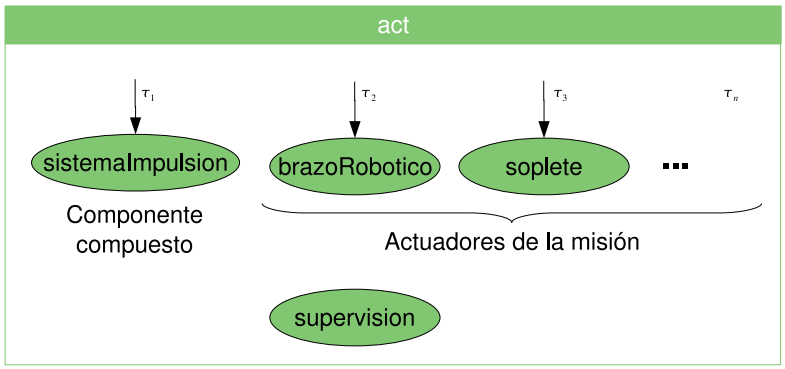


Figura 15.6: Componentes del Subsistema Actuador

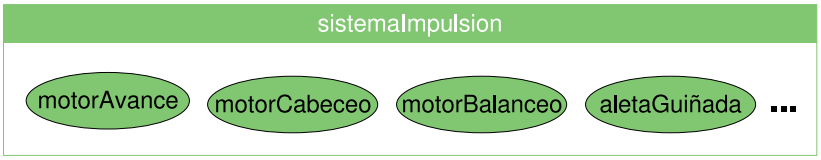


Figura 15.7: Componentes del sistema de impulsión

15.2.1. Componentes

El Subsistema Actuador incluirá tanto los motores y superficies de control del sistema de impulsión como otro tipo de actuadores propios de la misión y cuya presencia no es obligatoria para que funcione el AUV. En la figura 15.6 se muestran los componentes principales del Subsistema Actuador, donde el sistema de impulsión es otro componente compuesto, que contendrá los motores y superficies de control que lo forma y que se indican con algunos ejemplos en la figura 15.7.

Como se muestra en la figura 15.6 los componentes del subsistema reciben las entradas de control de los actuadores τ, que se dividirán en las entradas de control $\tau_1, \tau_2, \ldots, \tau_n$ de cada uno de los n actuadores —incluyendo los propios de la misión, a parte de los del sistema de impulsión, que estarían contenidos en $\tau_1 = (\tau_{1_1}, \tau_{1_2}, \ldots, \tau_{1_m})$, para m componentes del sistema de impulsión.

Desde el punto de vista del Subsistema Actuador, el sistema de impulsión sólo hace referencia a los actuadores —como hardware—, de modo que no existe ningún sistema de impulsión desde el punto de vista software —es en la sección **??**, al analizar los componentes del Subsistema de Navegación, donde se tendrá el modelo del sistema de impulsión, con su codificación software—, sino hardware. Así, el sistema de impulsión indicado está formado por componentes atómicos CoolBOT específicos de dicho sistema de impulsión; se podrían poner sin encapsular en el componente del sistema de impulsión de la figura 15.6.

En conclusión, se dispone de dos componentes (ver algoritmo 15.1) si no consideramos los actuadores específicos de la misión y modelamos los motores y superficies de control de forma genérica. Además, también se tendría un supervisor (ver figura 15.6). La finalidad del supervisor es chequear el correcto funcionamiento del resto de componentes —esto mismo se realizará para el resto de subsistemas, gracias a las facilidades que proporciona

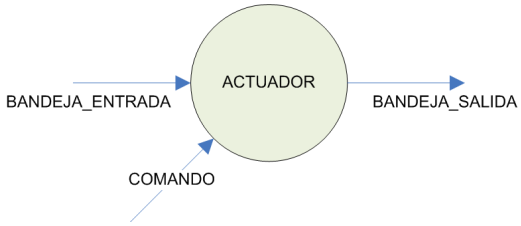


Figura 15.8: Componente Actuador

CoolBOT para ello.

```
1 act::motor
2 act::superficieControl
3
4 act::supervisor
```

Algoritmo 15.1: Componentes del Subsistema Actuador

Como la modificación de los elementos del sistema de impulsión no es trivial —requiere actualizar su contrapartida software— y los componentes de cada elemento de éste —v. g. los motores y superficies de control mostradas en la figura 15.7— son simples, lo más lógico y apropiado será disponer de un único componente CoolBOT para todo el sistema de impulsión, que se correspondería con el de la figura 15.6.

Con el nivel de diseño alcanzado se ha optado por modelar los componentes antes descritos mediante un único componente CoolBOT, denominado **actuador** y que se ilustra en la Figura 15.8. Éste simplemente dispone de la bandeja de entrada y salida, así como del puerto de entrada **comando**, por el cual recibirá los comandos de control.

15.3. Subsistema de Almacenamiento

Este subsistema surge de la necesidad de disponer de una API que haga que el proceso de almacenamiento sea transparente, tanto en los accesos a dispositivos de almacenamiento, como en la gestión de los datos almacenados. Por ello, se concibe con el objetivo de facilitar las siguientes tareas:

1. Almacenar datos.

2. Identificar los datos almacenados unívocamente, si bien esta responsabilidad podrá compartirse o realizarse por otros subsistemas —v. g. se trataría de introducir información de control a los datos, como el sello temporal.

En la Figura 15.9 se muestra el componente CoolBOT compuesto que modela el subsistema de almacenamiento. Internamente dispone de los componentes CoolBOT de interpretación del plan de almacenamiento, que son el **interprete pda** y el **gestor disparadores pda**, y un **inventario**, tal y como se muestra en el diagrama de la estructura interna del subsistema (véase la Figura 15.10).

Para el proceso de interpretación del PdA se reciben las muestras de las medidas del sistema a través del puerto de entrada **muestra**. Este esquema es común a todos los intérpretes de planes. Aunque no aparece en el diagrama, el inventario también dispondrá de un puerto de entrada **muestra**, al que también se conecta el puerto de entrada

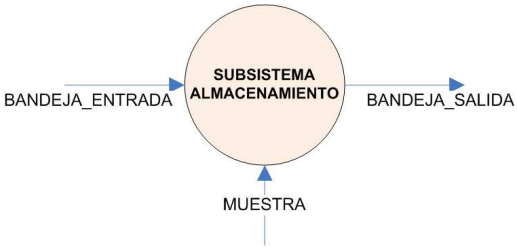


Figura 15.9: Subsistema de Almacenamiento. Componente Compuesto

muestra del subsistema. Esto es necesario para que el inventario reciba la muestras y las almacene si hay una acción de almacenamiento activa para la medida a la que pertenece dicha muestra.

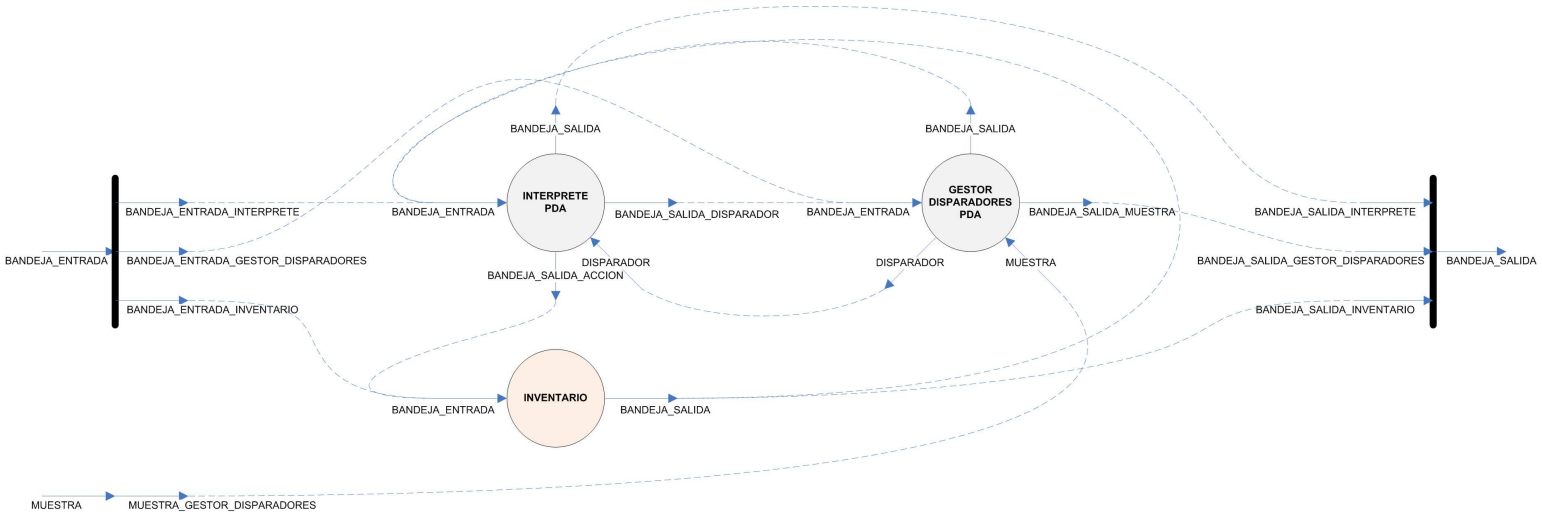


Figura 15.10: Subsistema de Almacenamiento. Componentes Internos

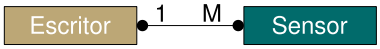


Figura 15.11: Relación entre escritor y sensor. Funcionamiento de escritor como cliente y de sensor como servidor

En este caso todo partirá de una orden como la del algoritmo 15.2, que sería generada internamente por el sistema para indicar los datos a almacenar —adicionalmente podrían venir acompañados de la información adicional para mantener la coherencia de los datos, i. e. poder identificarlos.

```
almacenar "dato"
```

Algoritmo 15.2: Orden para el Sistema de Almacenamiento

En la siguiente lista se enumeran las necesidades del Subsistema de Almacenamiento.

1. Se debe disponer de procesos encargados de toda la gestión del almacenamiento —mencionados ya en el Subsistema Sensorial, en la sección 15.7. A estos procesos se les denominará escritores, que tomarán los datos y la información de control adicional para almacenarla en un almacén. La información de control adicional podría ser la siguiente, aunque dependerá de los requerimientos de todo el sistema del AUV y por ende de sus subsistemas.

   a) Sello temporal, generado por el sistema para la identificación unívoca de los datos dentro de una misma misión —en principio.

   b) Identificador de la misión o del plan de la misión al que pertenece el dato, para complementar la identificación unívoca, extendiéndola a varias misiones.

   c) Información del componente o subsistema que han generado —o todos los que lo hayan generado o distribuido— los datos, para permitir posteriormente la traza del flujo de datos en el sistema.

   d) Otra información que pueda resultar útil para los subsistemas o componentes del AUV, o para garantizar la integridad y coherencia de los datos.

   Con la información de control se pretende que la recuperación de datos de los almacenes sea posible manteniendo los *metadatos* —información de control— para poder determinar cuándo y quién la generó, fundamentalmente.

   Los escritores funcionarán como clientes, de forma que —como se muestra en la figura 15.11— un escritor podrá relacionarse con un sólo sensor, mientras que un sensor, que funciona como un servidor, podrá tener asociados múltiples escritores (ver sección **??**, del Subsistema Sensorial).

2. El Subsistema de Almacenamiento recibirá por algún mecanismo de comunicación las peticiones para almacenar datos o volúmenes de datos, que pudieran venir como un flujo de datos de una transmisión. De esta forma, el subsistema dará el servicio de almacenamiento creando el escritor correspondiente para servir a dichos datos, almacenándolos en el almacén de datos oportuno. Como se mencionó previamente los datos se almacenarán con la información de control para garantizar su integridad y coherencia, fundamentalmente. La información de control, en principio, no la

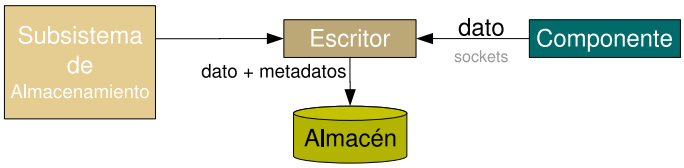


Figura 15.12: Almacenamiento de datos, junto con los metadatos —proceso escritor

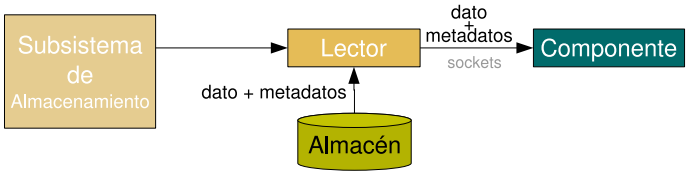


Figura 15.13: Recuperación de datos, junto con los metadatos —proceso lector

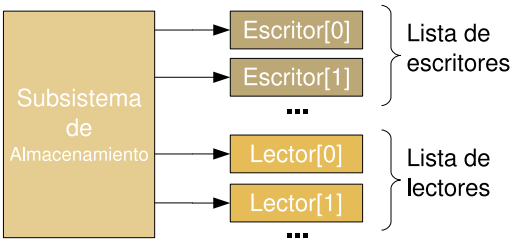


Figura 15.14: Lanzamiento de escritores y lectores

creará el Subsistema de Almacenamiento, sino que la obtendrá de otro subsistema y creará un paquete con datos e información de control. Esto es lo que realmente se almacenará, si bien todo esto será transparente para el subsistema que envía los datos a almacenar, que sólo tendrá que encargarse del envío de dichos datos y el Subsistema de Almacenamiento se encargará de empaquetarlos y almacenarlos con la información de control. En la figura 15.12 se muestra la arquitectura propuesta, donde se tendrá que modelar un escritor genérico para cualquier tipo de dato, que se debe serializar para ser recibido por los mecanismos de comunicación y posteriormente depositado en el almacén.

3. Del mismo modo que se tienen escritores para almacenar los datos en los almacenes, se dispondrá de lectores, que constituyen la contrapartida de los anteriores. Los lectores permitirán la lectura de los datos almacenados en un almacén, como muestra la figura 15.13. Un lector, al igual que un escritor, será un proceso independiente que funcionará como un cliente que tomará datos de un almacén y se los proporcionará al componente o subsistema que le haya solicitado este servicio —mediante un protocolo sobre la infraestructura de comunicación.

4. Para gestionar o administrar los escritores y lectores lanzados se tendrán que manejar las listas de escritores y lectores lanzados o activos. De esto se encargará el propio Subsistema de Almacenamiento, como se observa en la figura 15.14.

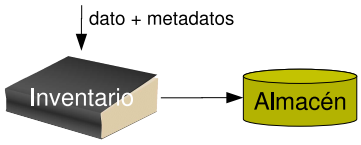


Figura 15.15: Inventario: Fachada del Almacén

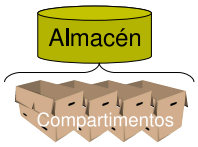


Figura 15.16: Compartimentos del Almacén

5. Para poder registrar los elementos almacenados en los almacenes se tendrá un inventario, como se muestra en la figura 15.15. El inventario nos permitirá garantizar la integridad de los datos, ya que permitirá la recuperación de los datos almacenados, gracias a los metadatos asociados a los mismos y la información del inventario. El inventario será también la fachada para acceder al almacén, i. e. a través del inventario se indicará los datos a escribir —incluyendo las modificaciones— o a leer. El inventario ofrecerá el servicio y registrará los cambios. Internamente el almacén se organizará como un conjunto de compartimentos (ver figura 15.16) en los que se almacenarán los diferentes datos. En el inventario quedarán registrados los datos que se han depositado en los diferentes compartimentos, para gestionarlos correctamente.

15.3.1. Componentes

Como muestra la figura 15.17, en el Subsistema de Almacenamiento se dispone de un total de 4 componentes, a parte del supervisor y el intérprete del plan de almacenamiento —*pda*— (ver algoritmo 15.3). Este subsistema se encargará de facilitar la labor de registro —*log*—, que se llevará a cabo según lo indicado en el plan de almacenamiento; el Subsistema de Supervisión sólo interpretará el plan de supervisión —*pds*— (ver sección **??**). Así, el Subsistema de Almacenamiento se encargará de las tareas de registro —indicadas en el plan de almacenamiento—, mientras que el Subsistema de Supervisión se encargará de las tareas propias de la supervisión, gestión de excepciones y aplicación de los planes de contingencia —todo ello indicado en el plan de supervisión, en principio.

```
1  alm::almacen
2  alm::escritor
3  alm::lector
4  alm::administrador
5
6  alm::interprete
7  alm::supervisor
```

Algoritmo 15.3: Componentes del Subsistema de Almacenamiento

En este caso hay componentes que funcionan como consumidores, mientras que otros son productores, como es el caso del escritor y lector, respectivamente. Por otro lado, el administrador se encarga de labores de gestión de escrituras y lecturas en los almacenes,

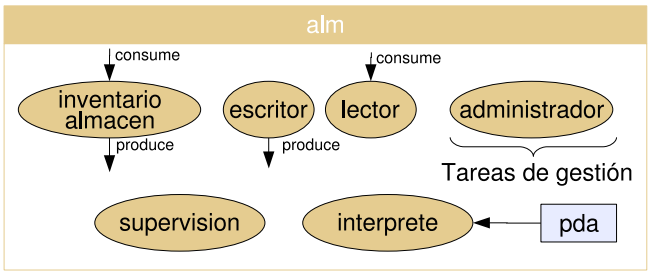


Figura 15.17: Componentes del Subsistema de Almacenamiento

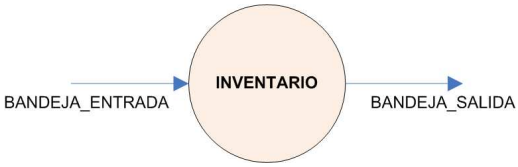


Figura 15.18: Componente Inventario

así como del propio inventario del almacén. Todo esto se indica en los componentes internos del Subsistema de Almacenamiento de la figura 15.17.

Por simplicidad se ha supuesto que el almacén se modela por encima de una HAL, por lo que en principio no importará si se trata de un HD (Hard Disk —Disco Duro), un SSD (Solid State Disk —Disco de Estado Sólido), etc. El inventario proporciona integridad a los datos almacenados y permite la recuperación/lectura de los mismos. Esto es posible porque en el inventario se registrará la información de los datos almacenados —ubicación, dimensiones o volumen, identificador o nombre, origen/fuente de los datos, etc. El inventario funcionará como la fachada del almacén. Canalizará los servicios de escritura o lectura que se soliciten al almacén. Cuando se escriban datos, no sólo se depositarán en el almacén, sino que también se actualizará el inventario.

15.3.2. Inventario

El diseño que finalmente se adopta en *Sick***AUV** consta de un componente CoolBOT denominado **inventario**, aparte de los componentes que se encargan de interpretar el PdA. En la Figura 15.18 se muestra la estructura del **inventario**, donde faltaría por indicarse el puerto de entrada **muestra**, para recibir las muestras generadas en el sistema. Esto es necesario para poder almacenarlas, si procede de acuerdo al PdA.

15.4. Subsistema de Comunicación

En la Figura 15.19 se muestra el componente CoolBOT compuesto que modela el subsistema de comunicación. Internamente dispone de los componentes CoolBOT de interpretación del plan de comunicación, que son el **interprete pdc** y el **gestor disparadores pdc**, y un **comunicador** y **acceso remoto**, tal y como se muestra en el diagrama de la estructura interna del subsistema (véase la Figura 15.20).

El componente **comunicador** se emplea para las comunicaciones de envío y recepción, tanto de datos como de muestras de medidas (véase la Sección 15.4.1). Mientras tanto, el

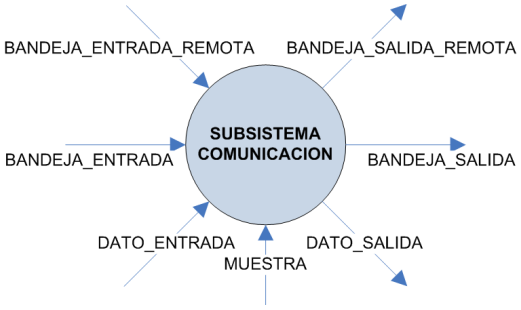


Figura 15.19: Subsistema de Comunicación. Componente Compuesto

acceso remoto facilita la posibilidad de comandar remotamente el sistema, i. e. recibir solicitudes de servicios que se direccionan internamente al subsistema que los gestiona. También permitirá que el sistema saque información al exterior (véase la Sección 15.4.2).

Dado las características de los componentes internos de este subsistema, se dispone de varios puertos de entrada y salida, como ilustra la Figura 15.19. Para el acceso remota se dispone de una bandeja de entrada y de salida remotas (**bandeja entrada acceso remoto** y **bandeja salida acceso remoto**), mientras que para la comunicador se dispone de puertos de entrada y salida para los datos y las muestras. Los datos entran por **dato entrada** para ser enviados al exterior, mientras que si son recibidas se sacan por el puerto **dato salida**. Respecto a las muestras, están entran por **muestra** para enviarse al exterior y cuando se reciben saldrían por otro puerto **muestra salida**, no representado en la figura. Las muestras que se reciben se inyectan en el sistema, lo que permite la teleoperación como si se tratase de un ROV.

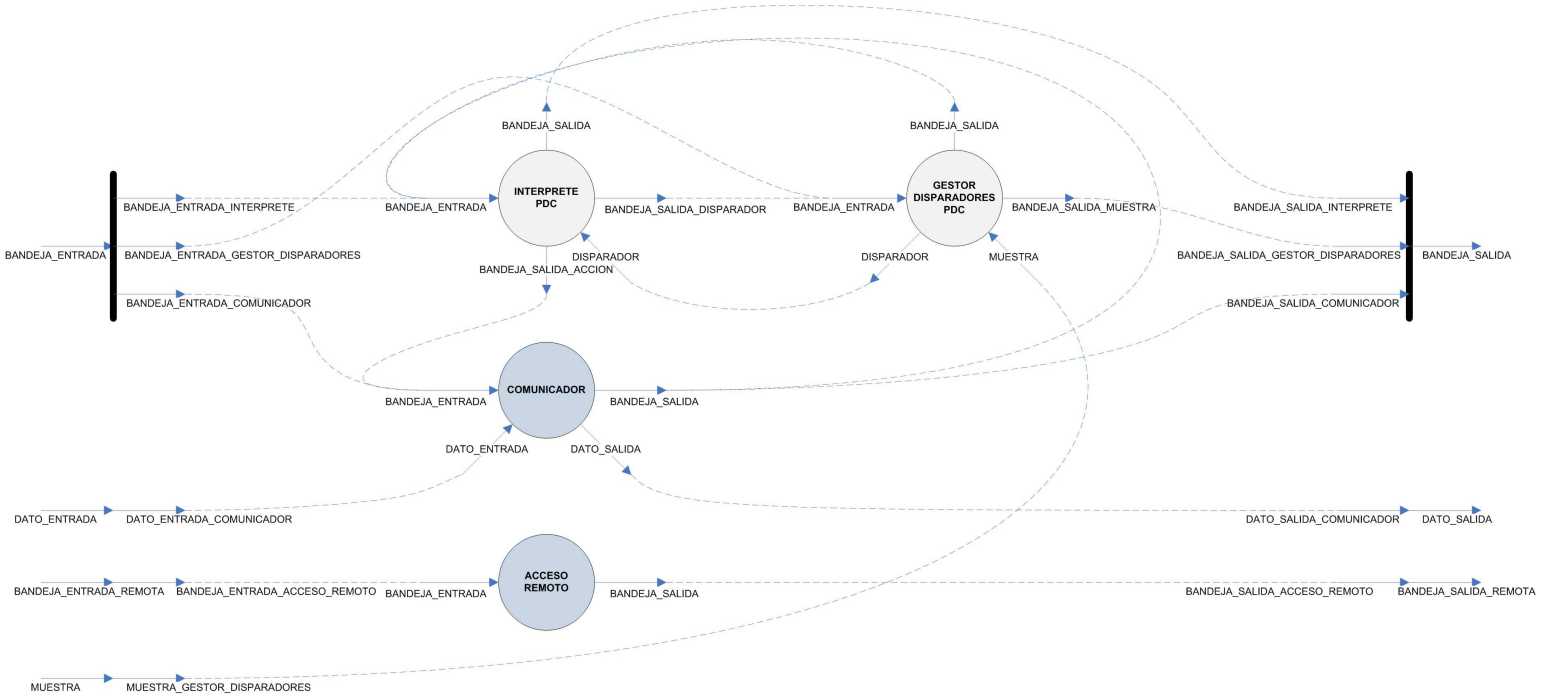


Figura 15.20: Subsistema de Comunicación. Componentes Internos

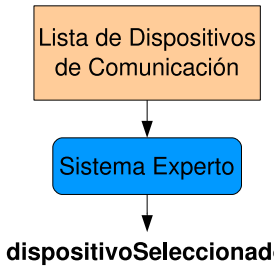


Figura 15.21: Dispositivos de comunicación

De acuerdo con el análisis del Subsistema de Comunicación existen distintos casos de uso aplicables a la hora de comunicarse. Las necesidades de comunicación, sin embargo, a la hora de analizar los requerimientos del Subsistema de Comunicación, pueden tratarse conjuntamente en ciertos casos, gracias al solapamiento de las mismas. Este es el caso del envío y la recepción de datos, ya que el tratamiento será similar y simétrico, a la vez que los componentes que intervendrán serán prácticamente los mismos. Por otro lado, también se debe considerar en la comunicación la posibilidad de control remoto y las notificaciones internas del sistema, que se redireccionarán y almacenarán en un buzón[3] —similar a un gestor de correo electrónico. Estas notificaciones internas sólo son aplicables a los mensajes que los subsistemas desean enviar al exterior, motivo por el cual se los notifican al Subsistema de Comunicación, que los almacenará en la bandeja de salida para enviarlos en cuanto pueda; la bandeja de entrada tendrá los mensajes recibidos del exterior, que deberán redireccionarse o notificarse a los subsistemas a los que vayan destinados —o a los que deban estar informados. En las tres secciones siguientes se trata cada una de estos casos de uso de comunicación —del Subsistema de Comunicación.

15.4.1. Envío y recepción de datos

Considerando el caso de uso consistente en el envío o recepción de datos, todo partirá de una orden como la del algoritmo 15.4 —se trata de un envío de datos, si bien también valdría una orden de recepción.

```
enviar "aviso" cada "2 horas"
```

Algoritmo 15.4: Envío o Recepción de datos

En la siguiente lista se enumeran las necesidades para el envío y recepción de datos.

1. Mecanismos para la comprobación de la conexión, i. e. si el AUV está en línea. Para ello se requiere una lista de dispositivos de comunicación (ver figura 15.21), si bien la comunicación debe tratarse de forma transparente —abstrayendo el hardware subyacente.

   Se tendrá que probar la conectividad a través de los diferentes dispositivos disponibles —que podrían tratarse de forma similar a los sensores, para lo cual el uso de CoolBOT [Domínguez Brito, 2003] proporcionará las herramientas para el diseño y codificación de estos componentes. Cuando se haya encontrado uno con conexión o,

[3] El buzón se subdividirá en bandeja de entrada y de salida para los mensajes recibidos y los salientes —debidos a notificaciones internas del sistema—, respectivamente (ver sección **??**).

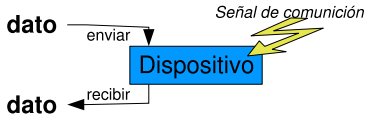


Figura 15.22: Dispositivo de Comunicación

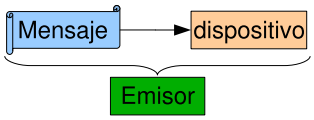


Figura 15.23: Proceso de emisión

en definitiva, cuando haya conexión —ya que el proceso debe ser transparente— se pasará a una siguiente fase. En esta fase se deben verificar los requerimientos mínimos para que la comunicación sea posible; en el caso de que haya varios dispositivos con conexión se deberá realizar una elección en base a ciertos criterios:

Ahorro energético

Alcance El rango que es capaz de alcanzar la señal del dispositivo de comunicación.

Seguridad Seguridad de que los datos llegarán correctamente a su destino. Se trata de una medida de la calidad de la señal y la consistencia ante los errores que pudieran producirse en el proceso comunicativo.

Velocidad (Ancho de Banda)

Como ya se ha dicho, todo esto será transparente, por lo que la elección del dispositivo a usar la realizará un Sistema Experto, dentro de la arquitectura del Subsistema de Comunicación.

2. Preparar, configurar o activar el dispositivo de comunicación seleccionado. El programa encargado en última instancia del envío o recepción de datos deberá disponer de un protocolo con el que se comunicarán con él otros procesos y le indicarán o proporcionarán el mensaje a enviar, o bien le pedirán el mensaje que desean recibir (ver figura 15.22). Al hacer esto con algún mecanismo de comunicación se consigue portabilidad en la arquitectura.

3. Crear el mensaje a enviar o recibir. Si es necesario, este mensaje se podrá situar en un almacén de forma intermedia en el proceso, bajo la gestión del Subsistema de Almacenamiento (ver sección **??**). Una vez se dispone del mensaje, en el caso de tratarse de un envío, un proceso denominado emisor se encargará de llevar a cabo la emisión controlando todo el proceso (ver figura 15.23).

   De forma análoga ocurre en la recepción de mensajes, si bien ahora el mensaje se construye al recibir los datos desde el dispositivo, de modo que el flujo de datos es en sentido contrario. Este proceso se controlará por un receptor, como se muestra en la figura 15.24.

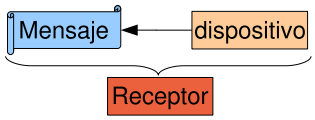


Figura 15.24: Proceso de recepción

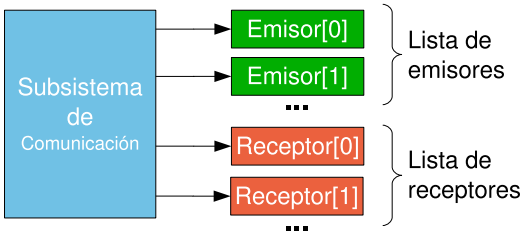


Figura 15.25: Lista de emisores y receptores

4. El propio Subsistema de Comunicación debe disponer de la lista de emisores o receptores, como la de la figura 15.25. Cuando cada uno de éstos tenga que realizar la labor de comunicación correspondiente tendrá que usar un dispositivo, que se seleccionará de forma transparente, de modo que en principio no es de ningún tipo concreto. Además, también se deberá acceder al almacén del que se leerá el mensaje a enviar o donde se almacenará el mensaje que se reciba, según se trate de un envío o recepción, respectivamente.

5. En el propio Subsistema de Comunicación se tendrá un bucle para comprobar en todo momento los disparadores —las condiciones de disparo— de las tareas de comunicación definidas en el plan de comunicación. Su arquitectura —desde el punto de vista de su implementación— será similar a la de los gestores de eventos para sistemas de ventanas, donde al producirse algún evento en el sistema del AUV, se chequearán las condiciones de los disparadores para lanzar, en caso de que se cumplan, las acciones o tareas asociadas. En este proceso se tendrá que evitar la espera activa —polling.

15.4.2. Control Remoto

Si ahora consideramos el caso de uso del control remoto, todo partiría con la recepción de un mensaje de petición de control remoto como el que se muestra en el algoritmo 15.5 —se trata de un envío de datos, si bien también valdría una orden de recepción.

```
peticion "controlRemoto"
```

Algoritmo 15.5: Control Remoto

Aunque en principio parece simple, el problema radica en que el Subsistema de Comunicación debe disponer de al menos una interfaz —dispositivo— de comunicación activa. Si no fuera así, sería imposible la comunicación con el AUV, pues no estaría en línea. La elección del dispositivo de comunicación a mantener activo tiene cierta complejidad y en dicha elección intervienen múltiples factores, como se comentará posteriormente e ilustra en la figura 15.26.

Una vez se haya recibido la petición de control remoto, el Subsistema de Comunicación se encargará de recibir y ejecutar comandos, funcionando como un intérprete de comandos. Algunos ejemplos de posibles comandos se indican a continuación, si bien no son de especial importancia en el proceso de comunicación, sino que formarán parte de dicho intérprete de comandos, que se muestra en la figura 15.27.

1. Cambios en los planes de la misión.
2. Deshabilitación/Habilitación de un sensor o componente del sistema del AUV.
3. Cambios en la configuración de la misión, como pueden ser los siguientes aspectos:
   a) Consumo energético medio deseado, máximo u otros.
   b) Velocidad deseada de crucero o en algún *waypoint*, tanto lineales como angulares —si es de interés.

En la siguiente lista se enumeran las necesidades para la gestión del control remoto —incluyendo la necesidad de solventar el problema de elegir un dispositivo de comunicación, o varios, para que se esté en línea y con la posibilidad de recibir la petición de control remoto.

1. Sistema Experto para la selección del dispositivo de comunicación más apropiado para estar activo y poder recibir la petición de control remoto. Ya se ha comentado previamente la problemática que afectaba a las peticiones de control remoto y como se hace necesaria la correcta selección de un dispositivo de comunicación para que dicha petición pueda recibirse. En esta selección hay varios factores que influirán en la misma, los cuales se mencionan a continuación y se ilustran en la figura 15.26.

   Ahorro energético Normalmente sería recomendable no activar aquellos dispositivos de comunicación con un elevado consumo energético, si bien también es importante el peso de otros factores, como el alcance, posiblemente muy relacionados con este factor.

   Alcance Lo ideal sería activar el dispositivo con mayor alcance, para que la posibilidad de control remoto sea posible desde zonas alejadas. Sin embargo, esto podría suponer un consumo energético muy alto. Se pone de manifiesto la contraposición de los factores ahorro energético y alcance.

   Tiempo de respuesta Este factor podría estar relacionado con la velocidad o ancho de banda, si bien hace referencia al tiempo que se tardará en servir a la petición de control remoto. Lo ideal será que este tiempo sea el menor posible, aunque tampoco es especialmente crítico. Otra cosa muy diferente, es que una vez recibida la petición de control remoto, el Subsistema de Comunicación —y el resto de subsistemas afectados— sirva a la petición y notifique al cliente de que se está a la espera de la recepción de comandos. Durante el control remoto habrá que determinar que ocurre con la ejecución de los planes de la misión, que en principio se continuarán ejecutando de forma normal, hasta que algún comando enviado a través del control remoto los modifique o aborte, fundamentalmente.

Figura 15.26: Sistema Experto para seleccionar el dispositivo de comunicación más apropiado para escuchar las peticiones de Control Remoto

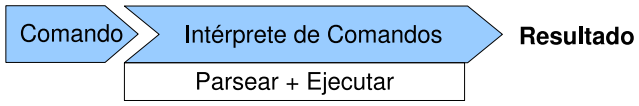


Figura 15.27: Intérprete de Comandos para Control Remoto

2. Intérprete de Comandos para que —una vez se entre en el modo de control remoto— se reciban y ejecuten los comandos del cliente de control remoto (ver figura 15.27). Los comandos se deberán chequear y analizar sintácticamente para que una vez verificada su correctitud y posibilidad de ejecución con éxito se ejecuten en el sistema del AUV, afectando a los subsistemas implicados.

3. La ejecución de los comandos se realizará mediante la implementación de los mismos de forma que se lancen de la forma más transparente y portable posible, del mismo modo que las tareas asociadas a los mismos se llevarán a cabo mediante mecanismos de comunicación entre el comando y los subsistemas y componentes implicados o afectados por el comando. Si suponemos como ejemplo la deshabilitación de un sensor, indicada por un comando como el del algoritmo 15.6, la comunicación dentro del sistema se realizará con el Subsistema Sensorial (ver sección 15.7), que ofrecerá como uno de sus servicios la posibilidad de deshabilitar sensores —adicionalmente se comprobará la consistencia del plan de medición, en este caso, al deshabilitar un sensor, i. e. si no imposibilita la ejecución de dicho plan.

```
deshabilitarSensor "IDENTIFICADOR"
```

Algoritmo 15.6: Comando de deshabilitación de un sensor

Por otro lado, para que los comandos puedan realizar sus tareas comunicando las acciones necesarias a los diferentes subsistemas del AUV, la arquitectura planteada establece que dichos subsistemas deberán funcionar como servidores, ya que en un momento dado podrán comunicarse múltiples clientes con ellos para solicitar determinados servicios. Del mismo modo que los comandos sería clientes en esta arquitectura, también los subsistemas deben funcionar como clientes —a la vez que como servidores— para poder realizar peticiones derivadas; esta funcionalidad como cliente podría realizarla un módulo interno a los subsistemas.

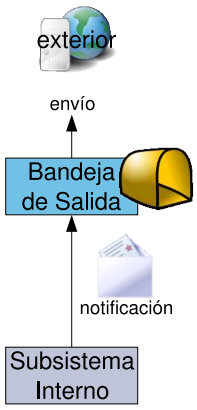


Figura 15.28: Bandeja de Salida

15.4.3. Notificaciones Internas

Si consideramos el caso de uso de las notificaciones internas o sistémicas al Subsistema de Comunicación nos encontraremos con un orden muy similar a la del algoritmo 15.4 —i. e. la que se planteaba para el caso de uso de los envíos y recepciones de datos. De hecho, ahora se considerará que la orden que se envía al Subsistema de Comunicación es idéntica. La diferencia es que antes era el plan de comunicación el que originaba dichas órdenes y ahora son subsistemas del sistema del AUV. En cualquier caso serán idénticas y los requerimientos de la primera son aplicables también a ésta —i. e. como se trata de envíos y recepciones de datos en última instancia, se requiere de la misma infraestructura. Sin embargo, se requerirán otros elementos adicionales que se citan seguidamente. Se trata principalmente de los buzones o bandejas de entrada y salida.

1. Bandeja de salida para almacenar las notificaciones originadas por los subsistemas —i. e. las notificaciones internas o sistémicas. Esto se ilustra en la figura 15.28, donde se observa que dichas notificaciones se irán almacenando en la bandeja de salida —equivalente a la de cualquier aplicación de correo electrónico— y el Subsistema de Comunicación las irá enviando al exterior cuando sea posible, según algún esquema de prioridades —o urgencia, aplicada a las notificaciones.

   Estas notificaciones podrán solicitar tanto un envío de datos —que estarían en el sistema del AUV— como una recepción —i. e. solicitando que se envíen datos desde el exterior para que se reciban por el AUV. Es en las recepciones, motivadas por este tipo de notificaciones, donde se fundamenta la necesidad de otro tipo de bandeja: la bandeja de entrada, que se comenta en el siguiente punto.

2. Bandeja de entrada para almacenar los datos entrantes (ver figura 15.29) que van destinados a los subsistemas —internos. En realidad, la bandeja de entrada no sólo se requiere para el caso de uso de las notificaciones internas, sino para la recepción de cualquier tipo de datos. Sin embargo, es especialmente importante para este tipo de notificaciones porque en el caso de haber varias notificaciones/peticiones habrá que gestionar su almacenamiento temporal en este tipo de buzón —la bandeja de entrada— para que según un esquema de prioridades se vayan enviando los datos a los subsistemas destinatarios.

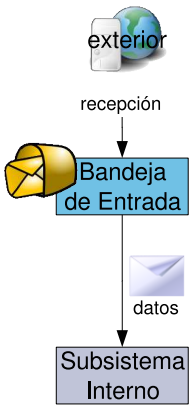


Figura 15.29: Bandeja de Entrada

3. La coherencia de las notificaciones se garantizará definiendo un formato uniforme para las mismas. En defnitiva, se trata de disponer de un elemento con cierta entidad en el sistema, que denominaremos notificación. La notificación deberá disponer de la siguiente información:

   a) Acción a realizar sobre el Subsistema de Comunicación, i. e. el tipo de servicio que se solicita a éste. Se trata de una orden como la del algoritmo 15.4, que queda integrada con la infraestructura ya planteada para el caso de uso del envío y recepción de datos, comentado en la sección 15.4.1.

   b) Información del remitente, i. e. el subsistema o componente interno —del sistema del AUV— que origina o lanza la notificación al Subsistema de Comunicación; este elemento será el remite. Esto le permitirá al Subsistema de Comunicación saber a quién informar de problemas en la realización de la notificación/petición —v. g. problemas en el envío o recepción de los datos. Igualmente, permitirá saber a qué subsistema se deben enviar internamente los datos recibidos tras ejecutar una recepción de datos, indicada en la orden —de recepción de datos— de la notificación.

   c) Metadatos adicionales sobre la misión o el AUV, como pueden ser:

      1) Misión o plan de la misión relacionado con la notificación.
      2) Identificador del AUV y del subsistema o componente que origina la notificación. Esto constituirá realmente el remite.
      3) Sello temporal.
      4) Cualquier otra información relevante.

4. El propio Subsistema de Comunicación debe proporcionar los mecanismos para gestionar el envío al exterior de los datos o mensajes depositados en la bandeja de salida. Del mismo modo, tendrá otro mecanismo equivalente para el envío a los subsistemas o componentes internos de los datos o mensajes depositados —tras su recepción del exterior— en la bandeja de entrada. Mediante una notificación asíncrona, mediante eventos, se podrán realizar estas tareas de forma eficiente, como se observa en la figura 15.30.

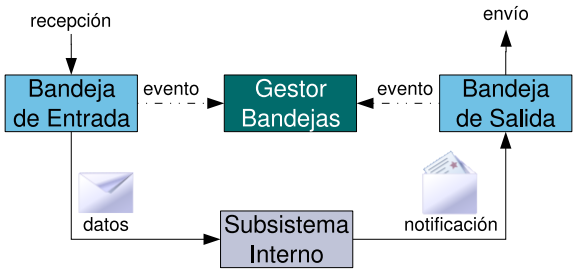


Figura 15.30: Gestión de las bandejas de entrada y salida

15.4.4. Componentes

El Subsistema de Comunicación dispone de un total de 6 componentes (ver algoritmo 15.7), además del intérprete para el plan de comunicación y el supervisor. En la figura 15.31 se observan estos componentes y como se identifican dos productores y consumidores bien diferenciados:

Productores Se trata de:

1. com::emisor
2. com::bandejaSalida

Consumidores Se trata de:

1. com::receptor
2. com::bandejaEntrada

Un administrador —o gestor de comunicaciones— se encargará de la gestión de emisiones y recepciones, así como de informar a otros subsistemas de los mensajes recibidos —que se almacenan temporalmente en la bandeja de entrada— y de enviar los mensajes asociados a las notificaciones internas o sistémicas —que habrán sido depositadas en la bandeja de salida. Es posible que este componente de administración —el administrador— no sea finalmente necesario, quedando embebido en el componente del propio subsistema o en los propios buzones —bandeja de entrada y de salida— y el intérprete del plan de comunicación.

```
1 com::dispositivo
2 com::emisor
3 com::receptor
4 com::bandejaEntrada
5 com::bandejaSalida
6 com::administrador
7
8 com::interprete
9 com::supervisor
```

Algoritmo 15.7: Componentes del Subsistema de Comunicación

Por simplicidad se ha supuesto que los dispositivos de comunicaciones se modelan por encima de una HAL, por lo que en principio no importará si se trata de Ethernet, WiFi, FM —radiofrecuencia—, Satélite, etc. No obstante, es posible que finalmente se tenga que desarrollar un componente CoolBOT para cada uno de ellos, ya que las interfaces de *Player* —de algunos de ellos— muestran importantes diferencias.

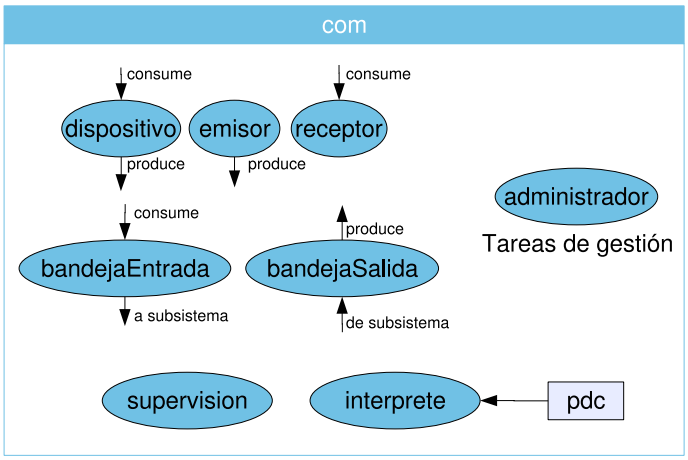


Figura 15.31: Componentes del Subsistema de Comunicación

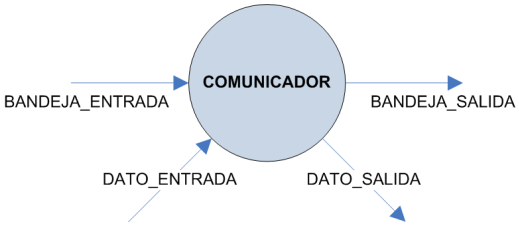


Figura 15.32: Componente Comunicador

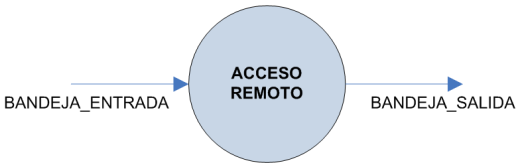


Figura 15.33: Componente Acceso Remoto

El diseño que finalmente se adopta en *Sick***AUV** consta de un componente CoolBOT denominado **comunicador** para el envío y recepción de muestras de medidas y de datos (véase la Sección 15.4.1). En la Figura 15.32 se muestra la estructura del mismo, donde se observa como recibe datos de entrada y salida; el mismo esquema se usaría para las muestras de medidas.

Del mismo modo, para el acceso remoto se dispone del componente CoolBOT **acceso remoto**, mostrado en la Figura 15.33, y que dispone de bandejas de entrada y salida remotas para que el sistema puede controlarse desde el exterior, del mismo modo que él también puede controlar otros sistemas.

15.5. Subsistema de Guiado

En la Figura 15.34 se muestra el componente CoolBOT compuesto que modela el subsistema de guiado. Internamente dispone de los componentes CoolBOT de interpretación

Figura 15.34: Subsistema de Guiado. Componente Compuesto

del plan de navegación, que son el **interprete pdn** y el **gestor disparadores pdn**, y un **guia**, tal y como se muestra en el diagrama de la estructura interna del subsistema (véase la Figura 15.35).

El componente **guia** encapsula toda la gestión que se hará de la navegación, de cara a generar los *waypoints* a los que debe llevar el vehículo. Estos *waypoints* no son los indicados en el PdN, sino unos internos que alimentan al subsistema de navegación y permiten que éste haga que el vehículo navegue apropiadamente.

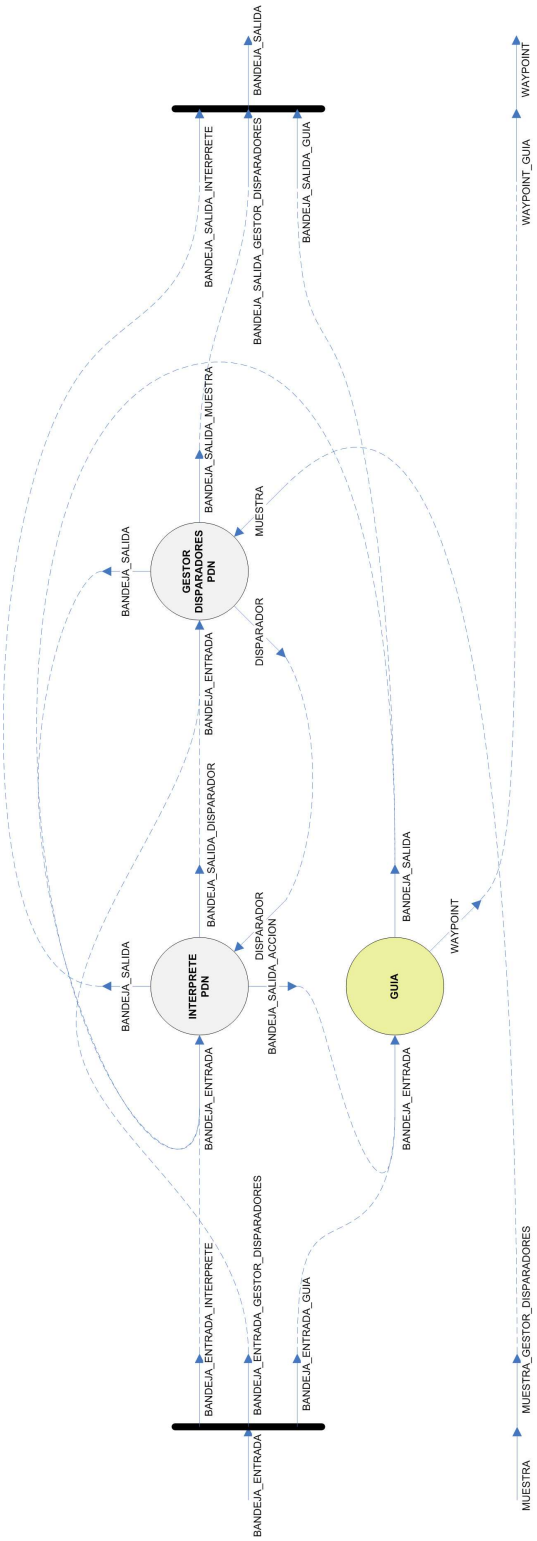


Figura 15.35: Subsistema de Guiado. Componentes Internos

De acuerdo con la especificación del plan de navegación, se tienen tres posibles indicaciones de la ruta a seguir, que serían tres casos de uso diferentes. No obstante, es labor del Subsistema de Guiado convertirlas a una representación única. Esta representación consiste en la indicación de los *waypoints* que se desean cumplir —que es precisamente una de las tres posibles definiciones del plan de navegación. En la figura 15.36 se ilustran los tres tipos de especificación del plan de navegación, que se comentan a continuación.

1. *Lista de* waypoints → Se tendrá una lista de *waypoints*, de modo que todo parte —de forma atómica— de una orden como la del algoritmo 15.8, idéntica a la orden que recibe el Subsistema de Navegación, como se ve en el algoritmo 15.12. En la figura 15.36(a) se muestra como sería la lista de *waypoints* y la trayectoria resultante.

```
1 irA "waypoint[i]"
```

Algoritmo 15.8: Alcanzar un *waypoint*

2. *Área a cubrir* → Cuando se define el plan de navegación con un área se indica el área deseada y el modo en que se desea cubrir ésta, con una orden como la del algoritmo 15.9. El área se define de forma poligonal y los modos estarán predefinidos. Como se muestra en la figura 15.36(b), dicha área se convertirá a una lista de *waypoints*, según el modo en que se recorra —v. g. en la figura se ha recorrido en zigzag.

```
1 cubrir "area[i]", "modo[j]"
```

Algoritmo 15.9: Cubrir un área

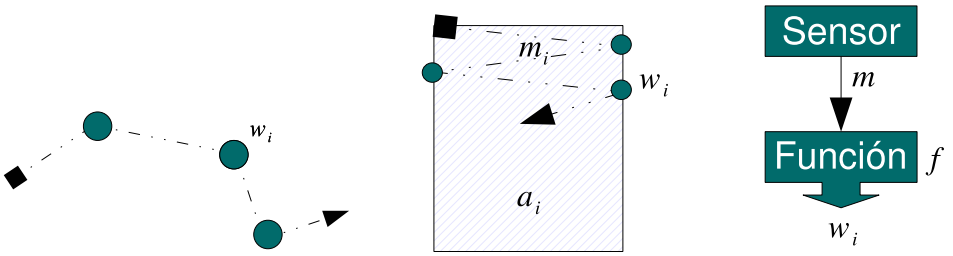
3. *Seguimiento de una medida* → Para indicar que se siga una medida se usará una orden como la del algoritmo 15.10, en la que se indica la forma en que se sigue la medida —mediante una función, como puede ser un gradiente— y la medida —para que se determine de forma transparente por el sistema del AUV de qué sensor obtenerla, lo cual está contemplado en el Subsistema Sensorial (ver sección **??**). De esta forma, la función determina el cálculo del siguiente *waypoint* —en la traducción a *waypoints*— y la medida determina el sensor a usar. Como resultado se van obteniendo *waypoints* dinámicamente, como se muestra en la figura 15.36(c).

```
1 seguir "gradiente", "temperatura"
```

Algoritmo 15.10: Seguimiento de una medida

En la siguiente lista se enumeran las necesidades de este subsistema.

1. Un módulo o sistema experto con la capacidad deliberativa para que a partir de las especificaciones de la misión por áreas a cubrir y seguimiento de medidas, produzca o las traduzca a una lista *waypoints* w_i, tal y como se definen las misiones de listas de *waypoints*. Esto es necesario para que cada uno de los *waypoints* se comuniquen al Subsistema de Navegación, que sólo acepta órdenes en forma de *waypoints* (ver sección 15.6). En el caso de que se especifique el plan de navegación directamente como una lista de *waypoints*, en principio no será necesario hacer nada, en cuyo caso se realizaría un *bypass* —i. e. se envían directamente los *waypoints* al Subsistema de Navegación—, pero en realidad los *waypoints* pueden verse modificados por otros



(a) Lista de *waypoints* w_i

(b) Área a_i a cubrir con el modo m_i

(c) Seguimiento de una medida m según una función f

Figura 15.36: Especificaciones del plan de navegación

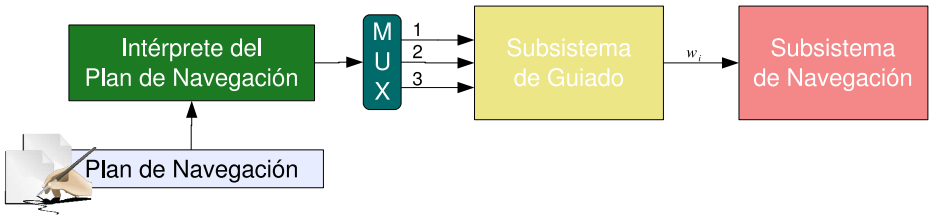


Figura 15.37: Conversión de la especificación del plan de navegación a *waypoints*

aspectos y por ello pasan por el Subsistema de Guiado —en lugar de ir directamente al Subsistema de Navegación—, como se observa en la figura 15.37[4].

Al realizar la conversión de áreas a cubrir y de seguimiento de medidas, a la lista de *waypoints*, el tratamiento será diferente y por ello se aplicará de forma separada desde su diseño.

2. Se requiere el acceso a otros subsistemas para poder tomar los datos necesarios o a considerar en la conversión de la especificación del plan de navegación a la especificación mediante la lista de *waypoints* —v. g. la toma de medidas para el seguimiento de medidas, que requiere el acceso al sensor que proporcionará la medida, interactuando con la fachada proporcionada por el Subsistema Sensorial (ver sección 15.7). Como se muestra en la figura 15.38, el Subsistema de Guiado solicitará al Subsistema Sensorial el servicio que necesita, i. e. tomar muestras de un sensor —apropiado para la medida deseada. Estas muestras se depositirán en un almacén —por intervención del Subsistema de Almacenamiento (ver sección 15.3)— y cuando el Subsistema de Guiado las solicite al Subsistema de Almacenamiento, éste le indicará en dónde se han almacenado. De esta forma, el Subsistema de Guiado accederá al almacén para recuperarlas, con la posible intervención del Subsistema

[4]Los tres tipos de especificación del plan de navegación se indican en la figura 15.37 de forma reducida con la siguiente numeración:

a) Lista de *waypoints*

b) Área a cubrir

c) Seguimiento de una medida

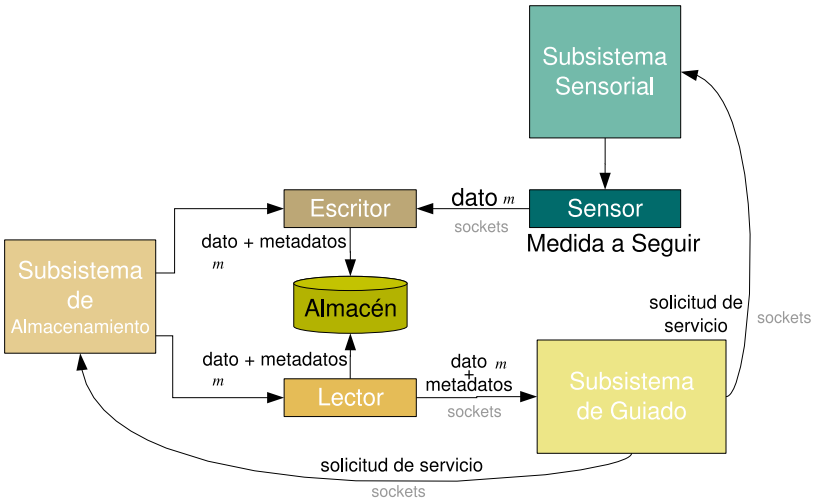


Figura 15.38: Acceso al Subsistema Sensorial para tomar la medida a seguir

de Almacenamiento —esto mismo ocurre cuando se accede a las muestras de los instrumentos de navegación, por parte del Subsistema de Navegación.

3. También se deben considerar otros aspectos en la deliberación para determinar los *waypoints* w_i, que requerirán, por tanto, la solicitud de servicios a otros subsistemas. Los datos que éstos proporcionen se despositarán en almacenes de datos para su posterior recuperación por parte del Subsistema de Guiado. Algunos de estos aspectos son:

   a) *Evitar obstáculos* → Se tendrán que tomar datos de un sensor que detecte obstáculos, como será posiblemente de un sónar de barrido frontal —situado en la posición de avance del vehículo. Según la información recibida, se corregirá el valor del w_i determinado previamente.

   b) *Girar según las restricciones del modelo del vehículo* → El Subsistema de Guiado debe proporcionar valores de *waypoints* consecutivos w_i, w_j al Subsistema de Navegación que sean posibles de alcanzar, de acuerdo con las restricciones del modelo del vehículo.

   c) *Ahorro energético* → Hay que tomar datos de la carga de las baterías. Según la carga se pueden tomar *waypoints* que definen trayectorias más o menos largas y velocidades mayores o menores al recorrelas.

   d) *Eficiencia en el recorrido* → La eficiencia en el recorrido podrá venir determinada por múltiples factores, como la velocidad, la trayectoria seguida, etc. Estos factores a su vez pueden depender de ciertas restricciones, como las del modelo dinámico del vehículo.

   e) Otros aspectos que se definirán según las necesidades concretas detectadas.

Todos estos aspectos se incluirán en el Subsistema de Guiado, proporcionándole un carácter deliberativo para que dada una serie de reglas con restricciones o deseos/preferencias, pueda inferir el w_i más adecuado, ya que el Subsistema de Navegación sólo tratará de alcanzar dicho punto.

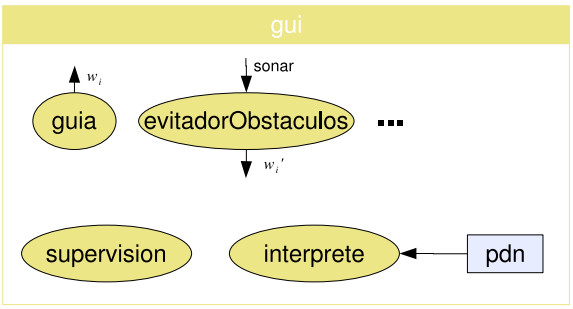


Figura 15.39: Componentes del Subsistema de Guiado

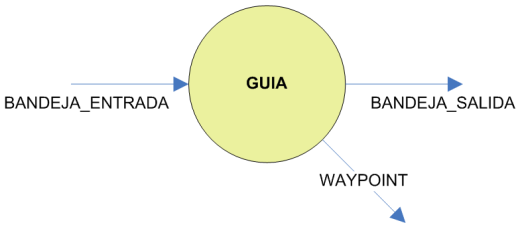


Figura 15.40: Componente Guia

15.5.1. Componentes

El Subsistema de Guiado será el que disponga de un intérprete, como se muestra en la figura 15.39, para realizar las tareas del plan de navegación, comandando al Subsistema de Navegación con los *waypoints* w_i apropiados. Para ello, a parte del intérprete y el supervisor del subsistema, dispondrá de un guía, que integrará la información de múltiples componentes orientados a corregir el valor final del *waypoint* w_i resultante. A modo de ejemplo, en la figura 15.39 se muestra un componente encargado de evitar obstáculos, denominado `gui:evitadorObstaculos` (ver algoritmo 15.11).

```
1  gui::guia
2  gui::evitadorObstaculos
3  ...
4
5  gui::interprete
6  gui::supervisor
```

Algoritmo 15.11: Componentes del Subsistema de Guiado

El diseño que finalmente se adopta en *Sick***AUV** consta de un componente CoolBOT denominado **guia** (véase la Figura 15.40). Éste dispone de un puerto de salida denominado **waypoint** el cual proporciona *waypoints* internos al sistema, que indicando al subsistema de navegación a dónde debe ir. Este diseño es muy abstracto, lo cual se justifica en el hecho de que esta parte del sistema, junto con el subsistema de navegación y actuador, se ha diseñado en menor detalle, ya que engloban todo un campo de estudio bastante complejo.

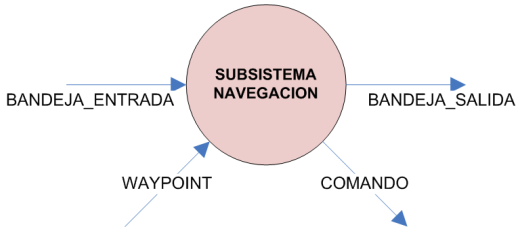


Figura 15.41: Subsistema de Navegación. Componente Compuesto

15.6. Subsistema de Navegación

En la Figura 15.41 se muestra el componente CoolBOT compuesto que modela el subsistema de guiado. Internamente dispone de los componentes CoolBOT de interpretación del plan de navegación, que son el **interprete pdn** y el **gestor disparadores pdn**, y un **piloto automatico**, tal y como se muestra en el diagrama de la estructura interna del subsistema (véase la Figura 15.42).

El componente **piloto automatico** dispone de un puerto de entrada llamado **waypoint**, por donde recibe el *waypoint* interno que debe alcanzar el vehículo, el cual ha sido producido por el subsistema de guiado. Internamente el piloto automático aplicará algoritmos de control y resolverá ecuaciones hidrodinámicas sobre el modelo físico del vehículo, para terminar generando los comandos de control que a través del puerto de salida **comando** serán enviados al subsistema actuador.

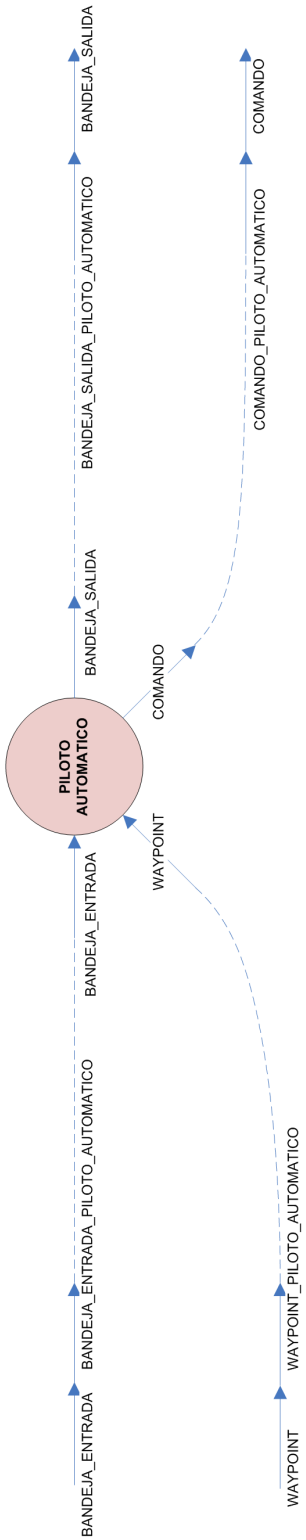


Figura 15.42: Subsistema de Navegación. Componentes Internos

En el Subsistema de Navegación todo partirá de una orden como la del algoritmo 15.12, suponiendo, por tanto, que la navegación se realiza de acuerdo a la indicación de *waypoints* exclusivamente. En este sentido, no se podrán tomar directamente las indicaciones del plan de navegación, ya que en éste se distinguen hasta tres tipos diferentes de especificación de la misión; esto se muestra en la sección 15.5, al tratar el Subsistema de Guiado. Para la navegación sólo se podrían tomar los planes de navegación —o la parte de éstos— definidos por *waypoints*.

```
1 irA "waypoint[i]"
```

Algoritmo 15.12: Caso de Uso del Subsistema de Navegación

Hay que recordar que un *waypoint* no sólo proporciona información del punto por donde debe pasar el vehículo, sino también de cómo debe pasar por éste, en términos del estado de la dinámica del mismo, como son su pose —i. e. posición y orientación—, su velocidad —traslacional o lineal y rotacional o angular—, etc. En principio se supondrá la siguiente información en un *waypoint*, aunque podría extenderse durante el diseño del sistema del AUV —o simplemente del Subsistema de Navegación.

1. *Pose*, es decir, posición —definida por x, y, z— y orientación —definida por ϕ, θ, ψ. Como se observa se trata de 6 parámetros, ya que en un vehículo submarino trabajaremos con 6 DOFs.

2. *Velocidades*, tanto lineales —que son u, v, w— como angulares —que son p, q, r—, que son las derivadas de la pose y, por tanto, también se dispone de 6 parámetros, para los 6 DOFs.

3. Otra información complementaria, como pueden ser:

   a) La incertidumbre de cualquiera de los valores anteriores —pose y velocidades. Se trataría de rangos de incertidumbre, que en el caso de las posiciones —dentro de la pose— indicadas en un *waypoint* se trataría de una elipse de incertidumbre en el caso más general —podría ser simplemente una circunferencia.

   b) El tipo de interpolación de las condiciones de un *waypoint* a otro. Esto define, por ejemplo, cómo se describen las curvas al pasar por varios *waypoints* consecutivos, ya que se interpolan las condiciones —pose y velocidades— realizando un movimiento más fluido/suave. Además, las restricciones de la dinámica del vehículo no permitirían determinados giros o acelaraciones, por lo que en realidad siempre se tendrá que aplicar este tipo de interpolaciones —i. e. el vehículo no podrá alcanzar realmente una posición puntual, o condiciones sin incertidumbre, ya que la nueva trayectoria para ir al siguiente *waypoint* definirá una variación brusca de las condiciones, difícilmente realizable por el vehículo, conforme a sus restricciones dinámicas.

Dicho esto, en la siguiente lista se enumeran las necesidades de este subsistema, para la navegación gobernada por *waypoints*.

1. Para obtener la pose —posición x, y, z y orientación ϕ, θ, ψ— a partir de los datos obtenidos por los instrumentos de navegación, que se tratan como sensores gestionados por el Subsistema Sensorial (ver sección 15.7), se tendrá que solicitar a

este subsistema el servicio de notificación de dichos valores o bien que se almacenen las muestras tomadas, mediante el Subsistema de Almacenamiento (ver sección 15.3). Éste es el esquema genérico de almacenamiento de muestras de cualquier sensor, visto en la figura 15.52. Los datos proporcionados por los instrumentos de navegación no indicarán realmente la pose del vehículo, sino que se deben computar mediante modelos de hidrodinámica y aplicando algoritmos de control o corrección de errores —v. g. filtro de Kalman. La obtención de la posición y la orientación, así como sus derivadas —las velocidades lineales y angulares, respectivamente—, se hará de forma separada, usando diferentes instrumentos de navegación, aunque aplicando un modelo de hidrodinámica común. En la siguiente lista se comenta cómo se obtiene cada uno de estos valores, que definen la pose y sus derivadas.

a) La posición x, y, z se podrá obtener con un GPS, pero no siempre se estará en línea con los satélites, ya que con el AUV sumergido no funcionará este sistema de posicionamiento. Por ello, para determinar la posición del AUV se tendrá que integrar la información histórica de la posición obtenida con el GPS y la aplicación del modelo de hidrodinámica, fundamentalmente. Para que su cómputo sea correcto se incluirá información odométrica, según el control de los impulsores del vehículo. La velocidad lineal se computará con dicho modelo a partir del histórico de posiciones determinadas.

b) La orientación ϕ, θ, ψ se integrará a partir de los datos obtenidos de un giróscopo —que podrá tener hasta 3 DOFs, que es lo necesario para un vehículo submarino. Junto con el modelo de hidrodinámica, la información del giróscopo se integrará para conocer la orientación en todo momento y por ende las velocidades angulares a partir del histórico de orientaciones.

Como se muestra en la figura 15.43, el Subsistema de Navegación dispondrá de un Piloto Automático, que es realmente el que realiza la navegación —si se activara el control remoto, se desactivaría el Piloto Automático— usando la información proporcionada por los instrumentos de navegación y los modelos de hidrodinámica y del AUV. De esta forma se integran y conocen la posición y orientación, que definen la pose, al mismo tiempo que se conocen las velocidades lineales y angulares. Con estos datos y la indicación del *waypoint* a alcanzar, el Piloto Automático será capaz de aplicar las entradas de control oportunas para que el vehículo alcance dicho *waypoint* —i. e. alcanzar una determinada pose con unas velocidades determinadas, a parte de otras posibles condiciones, previamente mencionadas—, como se verá en el siguiente punto.

El Piloto Automático dispondrá de varios procesos o hilos internamente, ya que se debe encargar de varias tareas. De hecho, determinar la pose y velocidades no sería más que una de dichas tareas.

Por otro lado, al tomar los datos de los instrumentos de navegación es posible que no sea necesario el uso de un almacén. No obstante, como esta tarea es gestionada por los subsistemas Sensorial y de Almacenamiento (ver secciones 15.7 y 15.3), en principio se realizará por un escritor, que depositará los datos de los instrumentos de navegación en un almacén, del cual los tomará el proceso del Piloto Automático encargado de determinar la pose y velocidades.

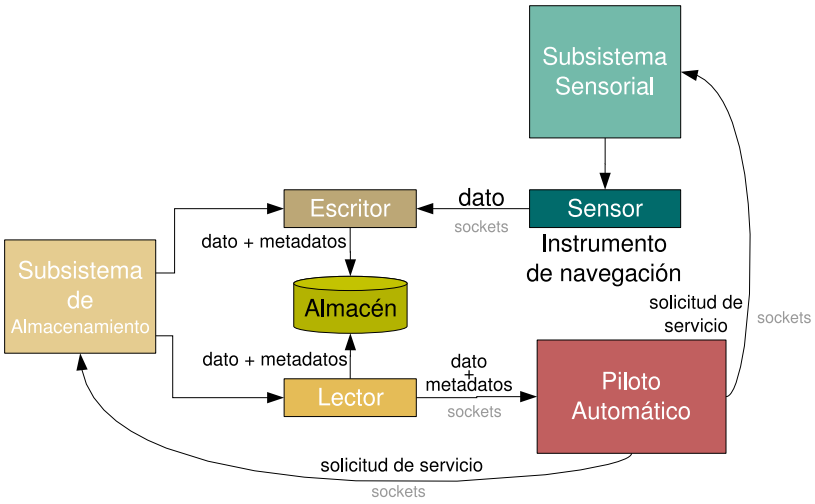


Figura 15.43: Piloto Automático e Instrumentos de Navegación

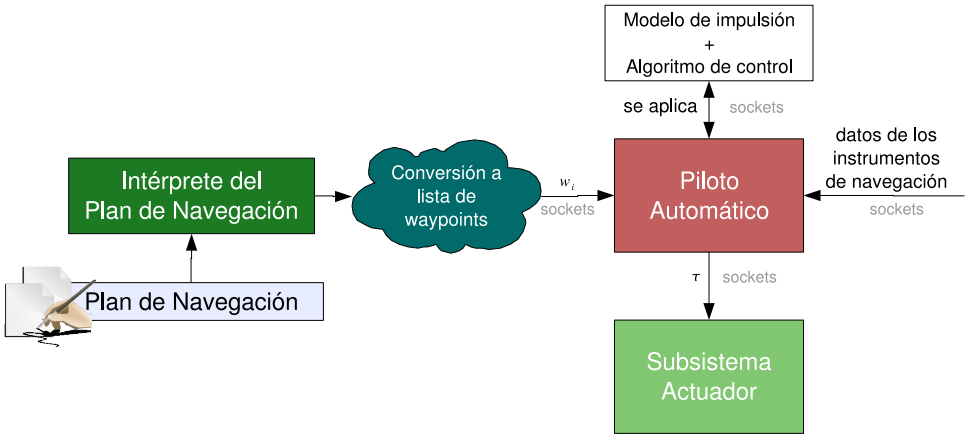


Figura 15.44: Piloto Automático y control de impulsores

2. Conocida la pose y velocidades, hay que determinar las entradas de los impulsores τ aplicando un modelo del sistema de impulsión y un algoritmo de control, con la finalidad de alcanzar el *waypoint* w_i indicado —que es la señal de control del Subsistema de Navegación. En la figura 15.44 se observa como otro de los procesos internos del Piloto Automático se encargará de aplicar un modelo del sistema de impulsión y un algoritmo de control —este modelo de impulsión y algoritmo de control podrían ser procesos que proporcionan los datos necesarios, para conseguir gran flexibilidad en el sistema— para proporcionar las entradas τ de los impulsores al Subsistema Actuador (ver sección 15.2). Éste es el encargado en última instancia de controlar o accionar los actuadores —entre los que se encuentran los impulsores y superficies de control, que forman parte del sistema de impulsión del AUV— según lo indicado en el vector de entradas τ.

   Por otro lado, hay que decir que aunque se dispone de un plan de navegación y un

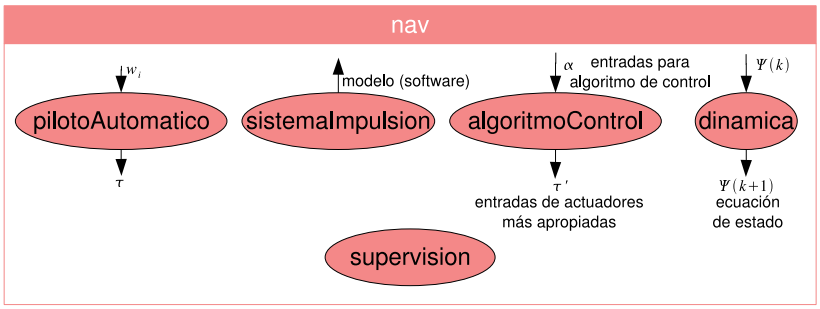


Figura 15.45: Componentes del Subsistema de Navegación

intérprete del mismo, en realidad será el Subsistema de Guiado (ver sección 15.5) el que se encargue de esto para poder convertir cualquier especificación de la misión a órdenes en forma de *waypoints*; esto es lo que se quiere expresar en la figura 15.44 y que luego se comentará en la sección 15.5.

3. Podría ser necesario el acceso a otros sensores a parte de los instrumentos de navegación. Esto sucederá en el caso de que el plan de navegación se defina como un seguimiento de medidas (ver sección 15.5). Sin embargo, el *grano* del Subsistema de Navegación se ha definido de forma que sólo recibe órdenes en forma de *waypoints*, por lo que esta labor la cubrirá el Subsistema de Guiado, que tendrá un carácter más deliberativo. Como se explica en la sección 15.5, en el caso de indicaciones del plan de navegación que no estén en la forma de *waypoints*, se convertirán a éstos, mientras que si son directametne *waypoints* se cortocircuitará —aunque en realidad también se permitirá que los *waypoints* se corrijan o modifiquen— el Subsistema de Guiado, para enviar dichos *waypoints* directamente al Piloto Automático del Subsistema de Navegación.

15.6.1. Componentes

En el Subsistema de Navegación diferenciaremos 4 componentes, a parte del supervisor (ver algoritmo 15.13 y figura 15.45). Se recibirá el *waypoint* w_i desde el Subsistema de Guiado —que actúa como filtro sobre el plan de navegación— y será el piloto automático el que lo emplee para generar las entradas de los actuadores del sistema de impulsión τ (ver sección 15.2). Para esto hará uso del modelo del sistema de impulsión —entendido como un elemento software—, el algoritmo de control y la dinámica[5].

```
1  nav::pilotoAutomatico
2  nav::sistemaImpulsion
3  nav::algoritmoControl
4  nav::dinamica
5
6  nav::supervisor
```

Algoritmo 15.13: Componentes del Subsistema de Navegación

El diseño que finalmente se adopta en *Sick***AUV** consta de un componente CoolBOT denominado **piloto automatico** (véase la Figura 15.46). Este componente genera los

[5]El modelo de dinámica aplicado realmente será de hidrodinámica en una implementación avanzada —pero simplificable a otros modelos. Este modelo realmente determina el comportamiento dinámico del medio —el líquido— y el vehículo —el AUV—, con sus restricciones.

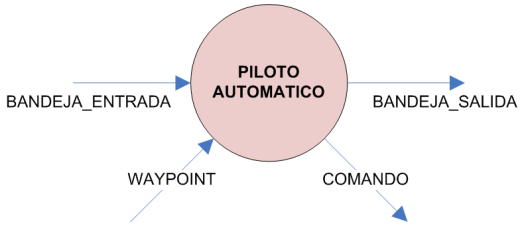


Figura 15.46: Componente Piloto Automático

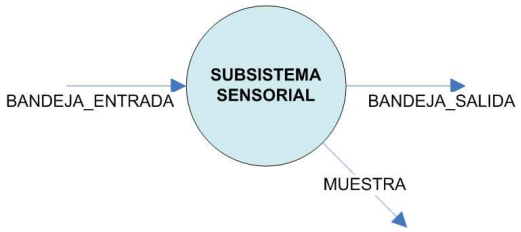


Figura 15.47: Subsistema Sensorial. Componente Compuesto

comandos de control para los actuadores a partir del *waypoint* interno al que debe ir el vehículo. Para ello dispone del puerto de salida **comando** y el de entrada **waypoint**, respectivamente.

15.7. Subsistema Sensorial

En la Figura 15.47 se muestra el componente CoolBOT compuesto que modela el subsistema sensorial. Internamente dispone de los componentes CoolBOT de interpretación del plan de medición, que son el **interprete pdm** y el **gestor disparadores pdm**, y un **sensor**, tal y como se muestra en el diagrama de la estructura interna del subsistema (véase la Figura 15.48).

El componente **sensor** encapsula toda la gestión de los sensores del vehículo y a través del puerto de salida **muestra** ofrece las muestras de todos ellas. La muestra estará por tanto autodefinida, i. e. indicará la medida a la cual pertenece, así como otra información adicional —v. g. unidades, medidas compuestas.

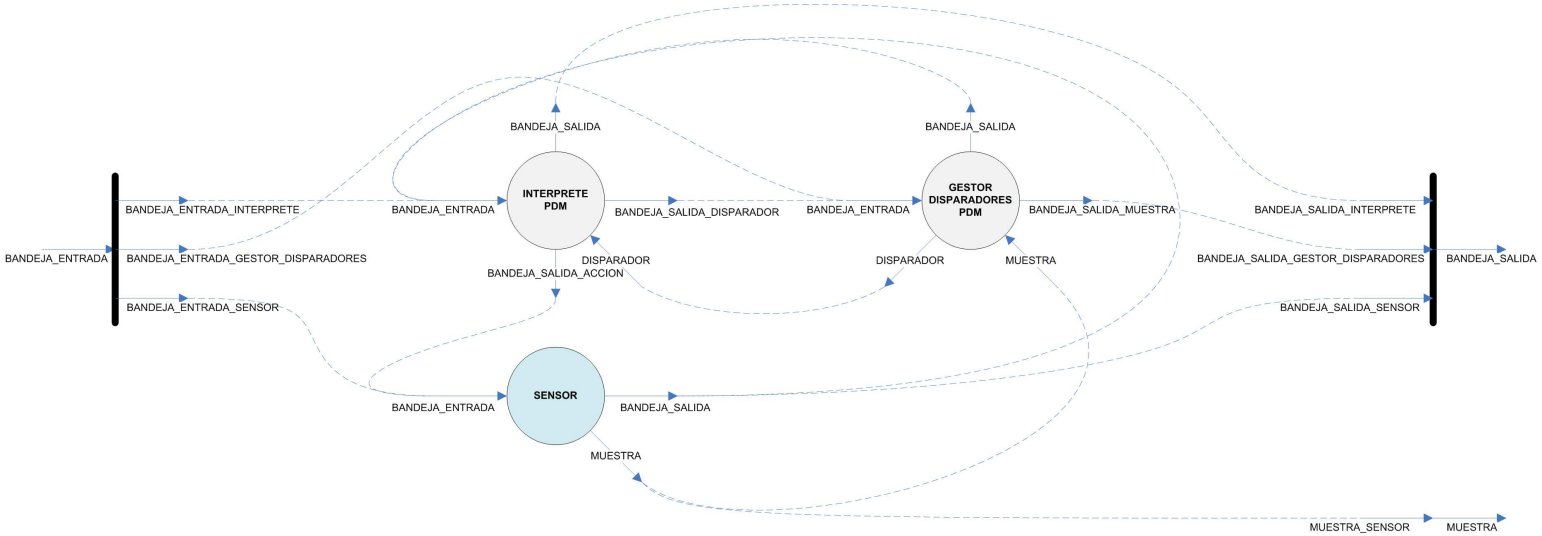


Figura 15.48: Subsistema Sensorial. Componentes Internos

Definición 15.3 (Muestra). *Información de una medida tomada en un instante concreto representado por un sello temporal. La información de la medida se representa por su valor y la unidad en que se mide* grosso modo *(véase la Sección 15.7 para más información sobre la representación adoptada para las muestras en* Sick***AUV****).*

15.7.1. Elección del sensor

A la hora de la toma de muestras, la posibilidad de disponer de sensores redundantes —i. e. que pueden medir una misma medida— resulta interesante por diversos motivos:

1. Proporcionar redundancia en las mediciones, haciendo que midan todos simultáneamente e integrando y combinando las medidas con algún tipo de filtro —v. g. filtro de Kalman. Este esquema resulta interesante para reducir el error en el muestreo.

2. Disponer de sensores de reserva, que permanecerán ociosos, para que en caso de que el sensor activo falle, sustituyan a éste. Este esquema permite seguir con la misión, sin tener que detenerla debido a un error de estas características.

La gestión interna de selección de sensores para el muestreo de una determinada medida se realizará de forma transparente al usuario. El sistema del AUV tomará las decisiones de acuerdo a su lógica interna (véase la Sección 15.7). La interfaz de planificación no requerirá que se indique ningún dato al respecto.

El proceso de validación de la misión que realiza el planificador de misiones (véase la **??**) se encargará de estimar el consumo de la misión. Dicha estimación debe considerar la gestión interna de los sensores, especialmente el uso de redundancia sensorial, para que la estimación sea al alza. Como resultado de la estimación se obtendría el porcentaje de batería restante al finalizar la misión, con un margen de error. Esta información se proporcionará junto con el equipamiento para que el sistema actúe en consecuencia (véase la Parte I).

En la especificación de las mediciones existen dos posibilidades:

1. Indicar qué sensor usar para tomar una determinada medida.

2. Indicar qué medida quiere tomarse y restricciones sobre el muestreo de la misma —v. g. frecuencia de muestreo y resolución, fundamentalmente. El sistema del AUV, en base al equipamiento, se encargará de seleccionar el sensor más apropiado dentro de los que ofrezcan dicha medida.

Las misiones darán soporte para ambos casos. El usuario simplemente tendrá que indicar qué medidas quiere en la interfaz del planificador de misiones y opcionalmente podrá indicar qué sensor o grupo de sensores deben usarse —quedando excluidos el resto, aunque también proporcionen la medida. Esto da mayor flexibilidad y portabilidad a la misión, pues será independiente del equipamiento; simplemente se validará que el equipamiento al menos disponga de un sensor que proporcione la medida requerida en la misión.

Todo partirá de una orden como la del algoritmo 15.14, que forma parte de los casos de uso del Subsistema Sensorial.

```
1 medir "temperatura"
```

Algoritmo 15.14: Caso de Uso del Subsistema Sensorial

La orden del algoritmo 15.14 es lo suficientemente representativa para analizar las necesidades de todo el Subsistema Sensorial de forma coherente. En la siguiente lista se enumeran dichas necesidades.

1. En primer lugar será necesario seleccionar un sensor que mida la magnitud indicada —v. g. la temperatura, para el ejemplo del algoritmo 15.14. En esta decisión se incluirán también otros aspectos, como pueden ser la frecuencia de muestreo, la resolución de los datos, etc. Se supone un Subsistema Sensorial con un Sistema Experto —más o menos complejo según las necesidades y alcance final— capaz de seleccionar un sensor que mida la magnitud indicada cumpliendo los requerimientos impuestos por las otras posibles condiciones. Este Sistema Experto se apoyará, por tanto, en una base de datos donde se almacenan los sensores disponibles en el AUV y las medidas que son capaces de medir, así como las configuraciones que admiten, en términos de las condiciones que se plantearán para las mediciones. Como se observa en la figura 15.49, tras recibir como entrada la medida y las condiciones, el Sistema Experto consultará la base de datos para proporcionar como salida el sensor seleccionado. En realidad, **sensorSeleccionado** será una lista de n elementos, donde nos podremos encontrar con los siguientes casos:

   $n = 0$ No existe ningún sensor para medir la magnitud indicada, conforme a las condiciones impuestas. Aunque podrían relajarse las condiciones, en principio, supondría un problema en la ejecución de la orden —entendida como la que se muestra en el algoritmo 15.14— y se produciría una excepción en el Subsistema Sensorial. Esta excepción deberá tratarse por el Subsistema de Supervisión (ver sección **??**), dentro de la gestión de excepciones. La rutina controladora de la excepción deberá determinar si la misión —o, al menos, al plan de medición, en este caso— puede continuarse o debe abortarse.

   $n = 1$ Se dispone de un único sensor para medir la magnitud cumpliendo las condiciones. Se considera que la elección es perfecta y no hay problemas o consideraciones derivadas.

   $n > 1$ Existen varios sensores candidatos para emplearse en la medición de la magnitud, todos ellos cumpliendo las condiciones. Los sensores seleccionados vendrán ordenados de acuerdo a algún criterio —i. e. ahorro energético, margen de confianza respecto a las condiciones, etc.— o configuración del Subsistema Sensorial —i. e. uso de la redundancia de medidas (usar varios sensores para tomar la misma medida), etc. Es en estos casos en los que el Sistema Experto tendrá que acceder a otros elementos del sistema del AUV o la misión —a parte de la base de datos con las relaciones entre sensores y medidas, además de sus características— y disponer de la lógica para emplearlos con el fin de proporcionar el orden más apropiado para la selección de sensores candidatos. Por tanto, dicho esto, se tomará el primer sensor seleccionado de la lista **sensorSeleccionado**.

2. Producir la transición —o transiciones— de estado en el sensor seleccionado para que comience a funcionar, es decir, a tomar muestras usando la configuración indicada en las condiciones. En primer lugar, estas transiciones de estado permitirán que el sensor se inicialice y configure, y que luego pase al estado de funcionamiento normal o ejecución. Los estados concretos vendrán determinados para el

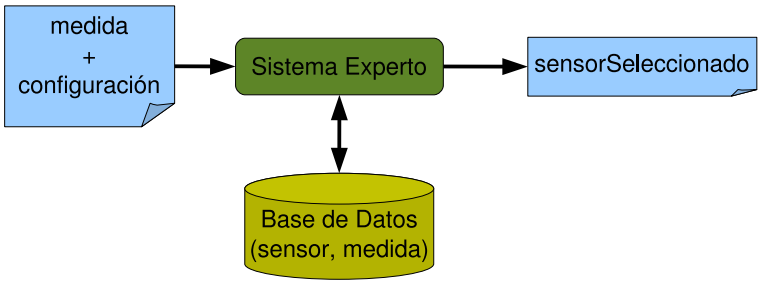


Figura 15.49: Sistema Experto para la selección de sensores a partir de medidas y condiciones

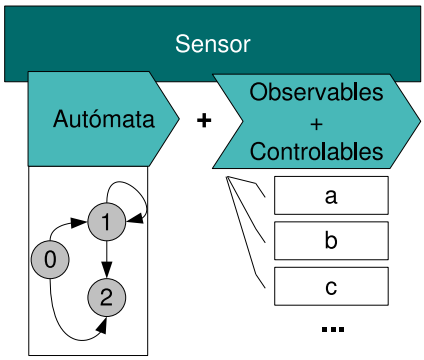


Figura 15.50: Autómata Control Homogéneo y Observables. Producir transición de estado

autómata de control adoptado. Se usará un autómata de control homogéneo, tomando la idea de *frameworks* como CoolBOT [Domínguez Brito, 2003] o *Player* [Gerkey et al., 2006], con la finalidad de disponer de un control homogéneo de los sensores —que podrá extenderse a otros componentes del sistema del AUV— y facilitar la recuperación frente a excepciones. En la figura 15.50 se puede observar como los sensores se modelarán internamente con un autómata de control homogéneo y con observables[6] y controlables[7]. Este esquema permite, respectivamente, transitar de estado y ver —mediante los observables— y editar —mediante los controlables— la configuración —según las condiciones impuestas.

En el caso de que se produzca algún tipo de fallo al transitar de estado o al intentar aplicar una configuración, el Subsistema Sensorial intentará solucionar el problema. En este sentido, podría incluirse la posibilidad de usar otro sensor, siempre y cuando la lista de sensores seleccionados dispusiera de más de uno —v. g. si se produjera un fallo con el primer sensor, se probaría con el segundo y así sucesivamente. Si el fallo

[6] Por observables entenderemos aquellos atributos, de los componentes, que son visibles —i. e. observables— por otros externos —v. g. la temperatura medida por un termómetro sería un observable, ya que no puede editarse/modificarse.

[7] Por controlables entenderemos aquellos atributos, de los componentes, que son editables —i. e. controlables— por otros externos —v. g. la frecuencia de muestreo de un termómetro es un controlable si se puede editar/modificar su valor. Un determinado atributo podrá ser a la vez observable y controlable —si así se precisa.

Figura 15.51: Almacén de datos para las muestras de un sensor

es imposible de recuperar, la excepción asociada al mismo se pasará al Subsistema de Supervisión (ver sección ??).

3. Asociar un almacén (ver figura 15.51) de datos para el sensor, donde éste podrá salvar correctametne las muestras que tome. En este punto se tendrán que considerar aspectos relativos a la estrategias de comunicación, donde se aprovecharán las expuestas en CoolBOT [Domínguez Brito, 2003].

   Desde el punto de vista de la implementación, el sensor dispondría de un puntero al almacén en una implementación conjunta de ambos componentes. Sin embargo, como se plantearán mecanismos de comunicación entre los distintos componentes del sistema del AUV, en realidad, se tendrán componentes que se ejecutarán por separado y se comunicarán con el componente de interés. Para permitir que un sensor se pueda usar para tomar varias medidas, en una arquitectura cliente/servidor, los sensores asumirán el rol de servidores; se realizarán componentes CoolBOT simples para los sensores, usando la API de los *proxies* que proporciona *Player* para las diferentes interfaces. Estos servidores de datos podrán disponer de múltiples clientes, que serán los escritores[8]. Se tratará de programas encargados de recopilar los datos de un sensor y almacenarlos en un almacén de datos —sea disco o memoria.

   Con la arquitectura propuesta se alcanzan los siguientes objetivos:

   a) Se libera al programa que maneja el sensor de las tareas de almacenamiento de los datos que toma.

   b) El programa que maneja el sensor se limitará a la toma de medidas, disponiendo de un autómata de control homogéneo, observables y controlables.

   c) Mediante la implementación de un servidor —definiendo un protocolo estandarizado para comunicarse con él— en el sensor o a través de los observables y controlables —de acuerdo a una arquitectura propia de CoolBOT—, otros componentes podrán tomar datos del sensor o configurarlo. De esta forma, si se dispone de la implementación del controlador o *driver* del sensor, será posible adaptarlo a esta arquitectura. No será necesario realizar la implementación del mencionado servidor para todos los sensores que se quieren adaptar a la arquitectura, ya que bastará con proporcionar una pequeña librería de desarrollo, en la que los sensores heredarían de una clase base `Sensor` que proporcionaría esta funcionalidad subyacente. Lo mismo ocurre si se usan observables y controlables, ya que se hará uso de los componentes de CoolBOT.

   d) En caso de ser necesario, el propio sensor tendría que disponer de buffers intermedios para salvar las diferentes frecuencias de producción y consumo de datos del sensor y escritor, respectivamente —en CoolBOT esto quedaría

[8] Se consideran escritores a los procesos que tomarán los datos y los almacenarán en un almacén, que se tratará de memoria o disco. También se les podría denominar *almacenadores*, pero parece más correcta la denominación de escritores —cuya contrapartida son los lectores.

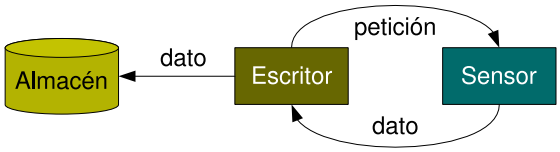


Figura 15.52: Arquitectura de almacenamiento: sensor, escritor y almacén

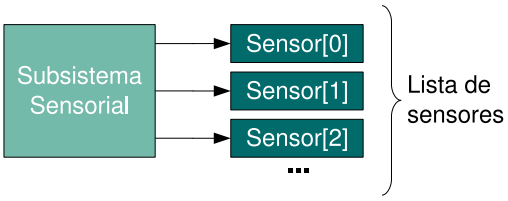


Figura 15.53: Lanzamiento de sensores

abstraído con la especificación del mecanismo y topología de comunicación que se emplee.

En la figura 15.52 se observa el proceso básico en el que un escritor realiza peticiones de datos al sensor y los datos que recibe los almacena en un almacén, incluyendo información adicional para que los datos sean coherentes —i. e. sello temporal, misión a la que pertenecen los datos, etc. Este esquema es lo más simple y portable posible, evitando que en el sensor se realicen tareas adicionales a las propias del sistema perceptual de un agente. Además, proporciona una arquitectura asíncrona, necesaria en sistemas robóticos.

4. El Subsistema Sensorial, propiamente dicho, se encargará del lanzamiento de los escritores y sensores necesarios, conforme al plan de medición, definido en la misión (ver figura 15.53). Se gestionará una lista de escritores en el Subsistema de Almacenamiento (ver sección **??**) y una lista de sensores en el Subsistema Sensorial. En principio, la relación de multiplicidad entre escritores y sensores que soporta la arquitectura es uno a muchos, es decir, el sensor es un servidor de datos —o productor, que permite múltiples accesos— y el escritor es un cliente —o consumidor—, de modo que a un sensor podrán acceder múltiples escritores (ver figura 15.11, de la sección **??**); por su parte, un escritor sólo accederá a un sensor. Esto es fácilmente modelable con los mecanismos de comunicación de los componentes de CoolBOT.

5. La gestión de los almacenes donde se encuentran las medidas tomadas se llevará por parte del Subsistema de Almacenamiento. Esto es necesario para que estos datos sean accesibles a otros componentes del resto de subsistemas. Se trata de una aplicación del patrón *facade* o fachada [Gamma et al., 2003]. El Subsistema Sensorial no se encargará de esto, pues para ello el Subsistema de Almacenamiento dispone de un inventario (ver figura 15.15), que es la fachada del almacén —internamente el almacén se divide en compartimentos (ver figura 15.16) para diferentes datos, cuyos metadatos —productor, sello temporal, etc.— quedan registrados en el inventario.

6. Se dispondrá de un bucle para lanzar las tareas de medidas, lo cual tendrá lugar al inicio, cuando se analice el plan de medición. No obstante, este bucle, que residirá —en principio— en el propio Subsistema Sensorial, también se encargará de vigilar el estado de los sensores por si se produce algún problema y tendrá que realizar las notificaciones pertinentes a otros subsistemas en caso de excepciones —fundamentalmente al Subsistema de Supervisión y excepcionalmente al de Comunicación (ver las secciones ?? y ??). Se tratará de un componente de supervisión —CoolBOT facilitará la creación de componentes de supervisión.

15.7.2. Componentes

Todos los sensores funcionarán como productores de datos —*senders* desde el punto de vista de los mecanismos de comunicación definidos en CoolBOT [Domínguez Brito, 2003]. Se distinguen tres tipos de sensores según la finalidad de los mismos (ver sección ??):

1. Instrumentos de navegación, si bien algunos sensores de misión podrían usarse para navegar en determinados casos —v. g. un sónar frontal para la evitación de obstáculos, un sensor de conductividad o salinidad para el seguimiento de dicha medida, etc.

2. Sensores de misión, que en principio no son obligatorios para la correcta operación del sistema del AUV.

3. Sensores internos, que permiten medir aspectos internos o sistémicos del AUV. Con ellos se vela por la integridad del mismo —v. g. la temperatura, la estanqueidad, etc.

De acuerdo con la heterogeneidad de las interfaces de *Player* para los sensores, parece lógico que cada sensor tendrá su propio componente, como se muestra en la figura 15.54. Además, el Subsistema Sensorial dispondrá del intérprete del plan de medición y un supervisor. En el algoritmo 15.15 se muestra el mismo subconjunto de muestra de sensores que en la figura 15.54, que servirá de primera aproximación en el análisis de los mismos. En total se dispone de 4 componentes para instrumentos de navegación, 5 para sensores de misión y 3 para sensores internos, a parte del intérprete del plan de medición y el componente de supervisión. No obstante, el número de elementos puede variar de una misión a otra y según los instrumentos empleados para la navegación y para vigilar el estado interno del AUV.

```
1  sen::imu
2  sen::brujula
3  sen::gps
4  sen::profundimetro
5  ...
6
7  sen::sonar
8  sen::laser
9  sen::termometroExterior
10 sen::ctd
11 sen::iops
12 ...
13
14 sen::baterias
15 sen::termometroAUV
16 sen::sensorEstanqueidad
17 ...
18
19 sen::interprete
20 sen::supervisor
```

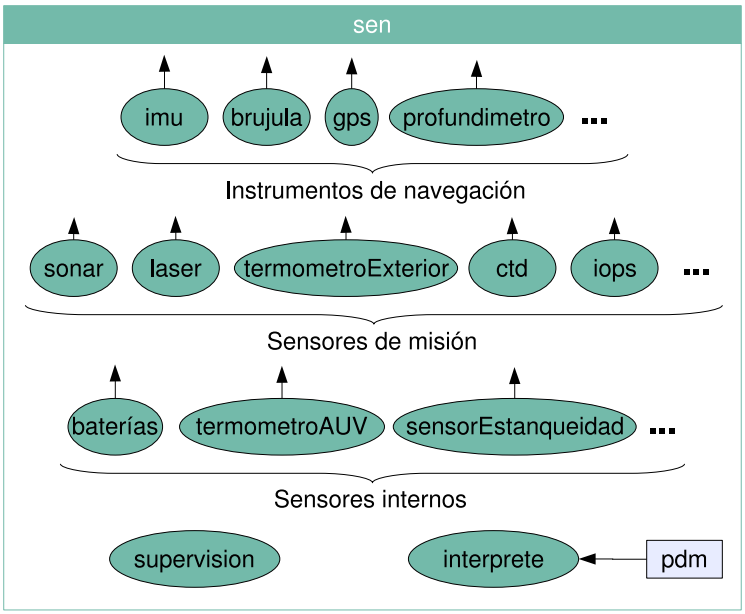


Figura 15.54: Componentes del Subsistema Sensorial

Tipología de sensor	Plan de la misión
Instrumentos de Navegación	Plan de Navegación
Sensores de Misión	Plan de Medición
Sensores Internos	Plan de Almacenamiento
	Plan de Supervisión

Cuadro 15.2: Relación entre las tipologías de sensores y los planes de la misión

Algoritmo 15.15: Componentes del Subsistema Sensorial

Existe la posibilidad de que se cree un sensor genérico del que heredaría el resto añadiendo otras características y funcionalidades, aunque sólo proporcionaría herencia de atributos.

Debido a los tres tipos de sensores diferentes —según su finalidad, como se mencionó previamente—, sería posible disponer de subsistemas separados para cada uno y de componentes CoolBOT que controlaran todo el equipamiento sensorial de los mismos. Además, sólo los sensores de misión se ven afectados por el plan de medición. De hecho, en la tabla 15.2 se muestra qué plan de la misión afecta a cada una de las tipologías de sensores; en el caso de los sensores internos el plan de almacenamiento indicará qué hay que registrar y el de supervisión qué se tiene que vigilar/supervisar. Esto justificaría la separación en subsistemas. Sin embargo, se mantiene sólo un subsistema para gestionar los tres tipos de sensores —el Subsistema Sensorial. No obstante, en lugar de un componente CoolBOT para cada sensor individual, se podrá disponer de un componente que gestione todos los sensores de un tipo.

El diseño que finalmente se adopta en *Sick***AUV** consta de un componente CoolBOT denominado **sensor** (véase la Figura 15.55). Éste dispone de un puerto de salida

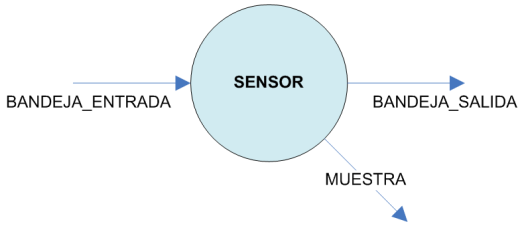


Figura 15.55: Componente Sensor

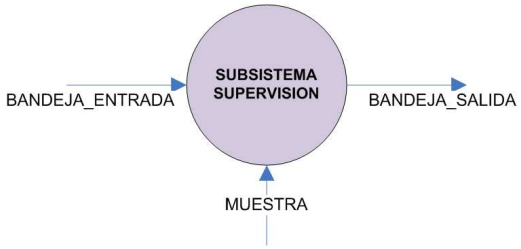


Figura 15.56: Subsistema de Supervisión. Componente Compuesto

denominado **muestra** el cual proporciona las muestras de todos los sensores del sistema, sean del tipo que sean; esto incluye aquellos sensores virtuales creados con la finalidad de ofrecer una muestra de una variable interna del sistema.

15.8. Subsistema de Supervisión

En la Figura 15.56 se muestra el componente CoolBOT compuesto que modela el subsistema de supervisión. Internamente dispone de los componentes CoolBOT de interpretación del plan de medición, que son el **interprete pds** y el **gestor disparadores pds**, y un **supervisor**, tal y como se muestra en el diagrama de la estructura interna del subsistema (véase la Figura 15.57).

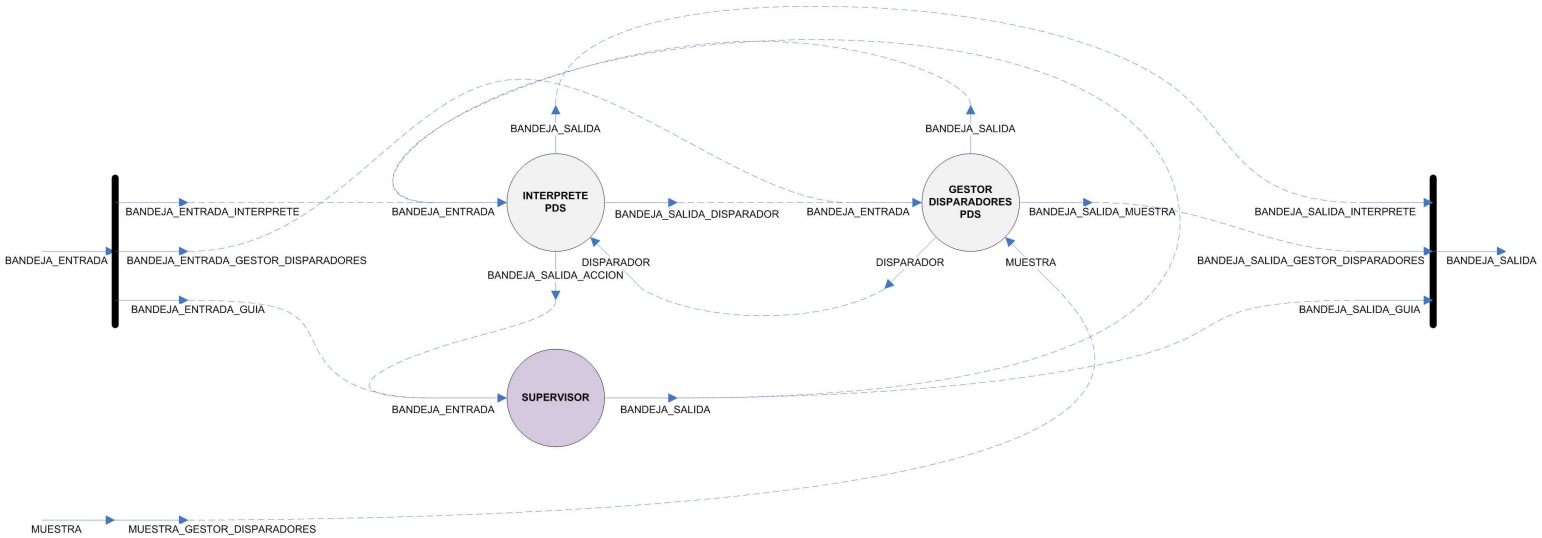


Figura 15.57: Subsistema de Supervisión. Componentes Internos

Antes de nada hay que identificar los casos de uso aplicables a las labores de supervisión en el sistema del AUV. Se identifican los siguientes:

1. Registrar tareas que se realizan en el sistema —i. e. en los diferentes subsistemas del sistema del AUV. Estas tareas son realmente datos —desde un punto de vista genérico—, por lo que se podrán tratar como las muestras de los sensores, en el sentido de que se usará un proceso escritor para que se encargue de salvar los datos registrados, añadiéndoles los metadatos oportunos, v. g. el sello temporal (ver la sección 15.8, del Subsistema de Almacenamiento).

2. Analizar la tareas o procesos en ejecución —evitando el *polling*[9]— para detectar posibles errores —v. g. valores medidos por los sensores, controlados por el Subsistema Sensorial, extraños o incoherentes de acuerdo con algún modelo, lo que los convierte en un indicio de error en el sensor. En principio, los subsistemas, e incluso los componentes, proporcionarán cierto soporte para la detección de errores, fallos o excepciones en los elementos que controlan —serán componentes de supervisión. No obstante, para evitar un alto solapamiento de las funcionalidades —incrementando la cohesión de componentes y subsistemas— y facilitar el diseño e implementación de los subsistemas, será el Subsistema de Supervisión el que dispondrá de la lógica para detectar y gestionar este tipo de problemas, en el resto de subsistemas.

3. Actuar frente a los errores, fallos o excepciones detectados —v. g. resetear un proceso, deshabilitar un componente como puede ser un sensor, etc. Es posible que se deban parar los componentes afectados y que se tenga que notificar lo sucedido a otros, lo que en última instancia requerirá cambios en los planes de la misión o puede contravenir las restricciones de la configuración actual del sistema del AUV. En este sentido, el Subsistema de Supervisión debe tener la suficiente *inteligencia* como para actuar de la forma más apropiada, para lo cual puede disponer de un módulo o sistema experto.

Tras este análisis previo, y disponiendo de un plan de supervisión, el formato de indicación de las tareas del Subsistema de Supervisión es bastante homogéneo, aunque con un alto nivel de abstracción —que provocará cierta complejidad en el proceso de codificación de la arquitectura diseñada y comentada en estas líneas. En principio, distinguiremos dos tipos de tareas —o casos de uso— diferentes:

1. Elementos que se desean registrar o supervisar, i. e. aquellos componentes o subsistemas que se desean subscribir al Subsistema de Supervisión para que se registren las actividades de los mismos. El nivel de detalle de este proceso de supervisión también debe indicarse. De hecho, este punto podría indicarse de forma más simple con el nivel de detalle o grado de supervisión deseado, bien para todo el sistema del AUV, o para cada componente o subsistema por separado. En el algoritmo 15.16 se muestra un ejemplo de orden de este tipo. Hay que hacer notar que las órdenes de registro deben hacer referencia a los observables y controlables de los componentes software del sistema del AUV, i. e. no se deben considerar como elementos a registrar los valores medibles por sensores. Sin embargo, existen ciertos casos en los que la frontera entre valores medidos por sensores y valores propios del sistema, no está bien definida; a continuación se comentan algunos ejemplos representativos:

[9] En lugar de *polling* se usarán eventos, por lo que se tendrá un esquema asíncrono.

a) La posición del AUV, que no se conoce con un sensor, sino que se obtiene gracias a un modelo de hidrodinámica y del propio AUV, y un algoritmo de control.

b) La carga de las baterías, que si bien se puede conocer con un sensor —es posible que realmente se interrogue a la API del módulo de administración de energía del Sistema Operativo anfitrión, como puede ser ACPI o APM, que en realidad constituye el driver de un sensor—, no parece lógico que se tenga que indicar en el plan de medición la toma de muestras del mismo. No obstante, para poder conocer los valores de su carga se tendrá que usar el Subsistema Sensorial y otros secundarios —como el Subsistema de Almacenamiento.

c) La temperatura interna del AUV, que del mismo modo que la carga de baterías —aunque de forma más clara— constituye la toma de muestras de un sensor. Por tanto, la toma de valores se gestionará con el Subsistema Sensorial —y otros—, pero no se indicará en el plan de medición, sino que será parte del plan de supervisión. De este modo, el Subsistema de Supervisión solicitará el servicio de toma de muestras al Subsistema Sensorial, enviándole la orden apropiada, como la del algoritmo 15.14, visto en la sección 15.7.

En principio, en estos casos, se optará por incluir las indicaciones de registro en el plan de supervisión, por no resultar apropiadas en el plan de medición, desde el punto de vista de los casos de uso.

```
1 registrar "datos almacenados"
```

Algoritmo 15.16: Registro de un elemento

2. Las acciones a tomar frente a los diferentes errores, fallos o excepciones identificados en los componentes y subsistemas. Esto forma parte de la gestión de las excepciones, pero no se trata de indicar las acciones exactas o de forma algorítmica, sino más bien una descripción o configuración de cómo se desea que se actúe frente a dichas excepciones. Con estas indicaciones, el Subsistema de Supervisión será capaz de tomar las accciones adecuadas conforme a las mismas. Se tendrá una orden como la del algoritmo 15.17, donde `"excepcion"` es el nombre de una excepción dentro de una lista de excepciones —v. g. `"falloSensor"`, `"falloImpulsor"`, etc.— y `"accion"` es la acción a tomar, especificable de dos formas diferentes:

   a) Acciones por defecto o predefinidas, como pueden ser:

      1) *Abortar* una tarea, un componente o subsistema —deshabilitarlo—, un plan, una misión, etc.
      2) *Continuar* —i. e. en cierto modo equivaldría a ignorar la excepción, o bien no es necesaria ninguna acción.
      3) *Abortar y Propagar*, en cuyo caso, tras abortar se informa a los posibles elementos afectados lanzando nuevas excepciones en cadena. Éstas se tratarán internamente por el Subsistema de Supervisión y los propios subsistemas o componentes afectados.
      4) Otras acciones predefinidas.

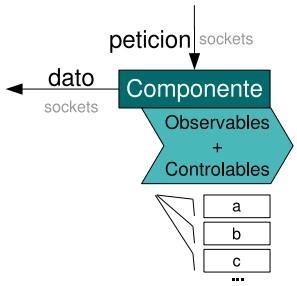


Figura 15.58: Observables de un componente

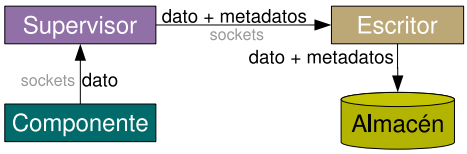


Figura 15.59: Supervisor y gestión del proceso de supervisión

b) Acciones más específicas o *ad hoc*, que podría tomar la forma de rutinas —especificación algorítmica de las acciones. Sería más complejo indicar acciones de esta forma, pero permitiría una flexibilidad completa. No obstante, la arquitectura del sistema del AUV debe ser capaz de permitir este tipo de especificación de las acciones.

```
1  si "excepcion" entonces "accion"
```

Algoritmo 15.17: Actuación frente a una excepción

En la siguiente lista se enumeran las necesidades del Subsistema de Supervisión para los dos casos de uso antes definidos.

1. *Escritor* de datos a registrar —junto con los metadatos. Usará el Subsistema de Almacenamiento (ver sección 15.3) para solicitar el servicio de un escritor, para que tome los datos del elemento a registrar y los deposite en un almacén.

2. Acceso o consulta a los demás subsistemas o componentes de todo el sistema del AUV. Se realizará a través de mecanismos de comunicación con los que, en principio, se accederá a observables y controlables —en la figura 15.58 se muestra un componente y sus observables y controlables.

3. Se necesita un Supervisor —que hasta cierto punto sería un sistema experto, en determinados casos. Será un proceso encargado de supervisar ciertas tareas o elementos. Para ello gestionará los datos que se almacenarán —v. g. para detectar datos corruptos, incoherentes, etc.—, como se muestra en la figura 15.59, entre otras tareas.

4. Gestor de excepciones, encargado de recibir las notificaciones de las excepciones que se producen en el sistema, de modo que funcionará como un servidor asíncrono

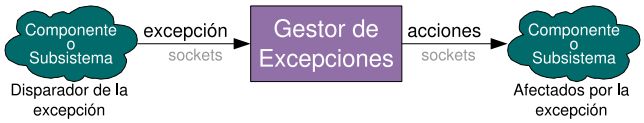


Figura 15.60: Gestor de excepciones y gestión de las mimas

—en el marco de una arquitectura cliente/servidor. Recibirá las excepciones y se encargará de ejecutar las acciones asociadas a las mismas, así como determinar que otras acciones derivadas deben llevarse a cabo, según los elementos que se vean afectados (ver figura 15.60). En este sentido, se tratará de un sistema experto, capaz de determinar los elementos afectados y cuál es la acción más adecuada, conforme a las indicaciones del plan de supervisión y el estado actual del sistema del AUV.

15.8.1. Componentes

El Subsistema de Supervisión dispone de un componente (ver algoritmo 15.18 y figura 15.61) fundamental:

1. El supervisor del sistema del AUV —i. e. el supervisor del resto de subsistemas.

No se requiere un componente encargado de registrar y realizar el *logging* de determinados aspectos del sistema del AUV —que se denominaría registrador— en el Subsistema de Supervisión, pues esta labor se llevará a cabo por el Subsistema de Almacenamiento (ver sección 15.3). Los registros se realizarán según lo indicado en el plan de almacenamiento, que contiene este tipo de información en exclusiva. El plan de supervisión será ortogonal a éste, indicando sólo tareas de supervisión y gestión de excepciones fundamentalmente.

Se dispondrá de un intérprete para el plan de supervisión, donde vendrán indicadas las tareas que debe realizar el componente antes mencionado —`sup::supervisorAUV`. También se tendrá un supervisor, que se diferencia del propio del subsistema porque éste último se denomina `sup::supervisor`.

```
1 sup::supervisorAUV
2
3 sup::interprete
4 sup::supervisor
```

Algoritmo 15.18: Componentes del Subsistema de Supervisión

La presencia de los componentes de supervisión en cada subsistema y un Subsistema de Supervisión permitirá una supervisión jerarquizada. Cuando los componentes de un subsistema detecten algún problema lo notificarán al componente de supervisión de su subsistema mediante una excepción. Si no puede solucionarse el problema de forma local al subsistema, se notificará con otra excepción al Subsistema de Supervisión y será el componente de supervisión del AUV —`sup::supervisorAUV`— el que intente solucionarlo —aplicando la gestión de excepciones o el plan de contingencia apropiado, que podrá venir indicado en el plan de supervisión.

El diseño que finalmente se adopta en *Sick***AUV** consta de un componente CoolBOT denominado **supervisor** (véase la Figura 15.62). Éste no dispone de ningún puerto, aparte de las bandejas de entrada y salida. Se encargará de lanzar y controlar las acciones

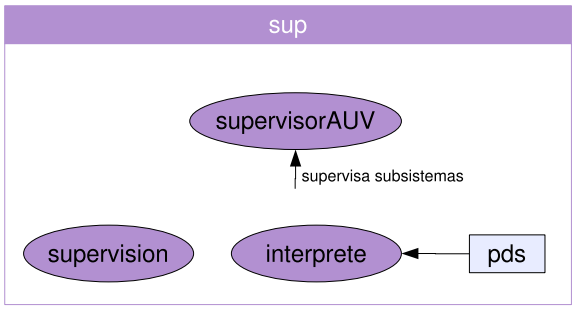


Figura 15.61: Componentes del Subsistema de Supervisión

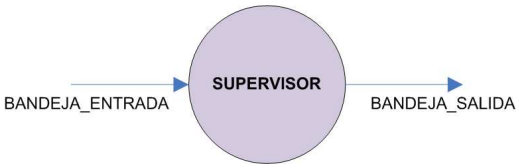


Figura 15.62: Componente Supervisor

del PdS, como son la ejecución de planes y de cualquier comando del sistema. Para esto último realizará una traducción para realizar la solicitud del servicio de dicho comando al componente del sistema que la ofrezca. También es importante indicar que cada subsistema también realizará labores de supervisión de sus componentes internos, así como el propio sistema las hará de los subsistemas. No obstante, estos procesos de supervisión serán intrínsecos al propio sistema, mientras que en el caso del **supervisor** se trata de la realización de la supervisión aplicando lo indicado en el PdS.

Este subsistema se encargará de dos tareas fundamentales:

1. *Fase de Inicialización* → Cuando el sistema del AUV se arranque, el Subsistema Central será el que controle dicho proceso, diferenciándose varias fases intermedias, después de iniciarse el Sistema Operativo anfitrión.

   a) Tras inicializarse *Linux* —suponiendo que sea el Sistema Operativo anfitrión—, en el *script* de arranque se lanzará la aplicación desarrollada, es decir, el sistema del AUV. Su primera fase consiste en arrancar los subsistemas necesarios —creando los procesos e hilos correspondientes—, si bien no habrá planes de la misión definidos inicialmente.

   b) Recepción de los planes de la misón y la configuración del sistema del AUV. Simplemente se recibirán y almacenarán en el sistema de ficheros del Sistema Operativo, de acuerdo con las necesidades del sistema del AUV para que sean identificados por éste. Durante este proceso el AUV podrá estar en *tierra*, por lo que no se debe iniciarse la ejecución de la misión —i. e. de los planes de la misión—, ya que hay que esperar a la botadura del AUV.

   c) Iniciar la ejecución, i. e. lanzar los procesos para los subsistemas y componentes —v. g. sensores, actuadores, escritores, etc.—, para que comience a operar según lo indicado en los planes de la misión y la configuración del sistema del

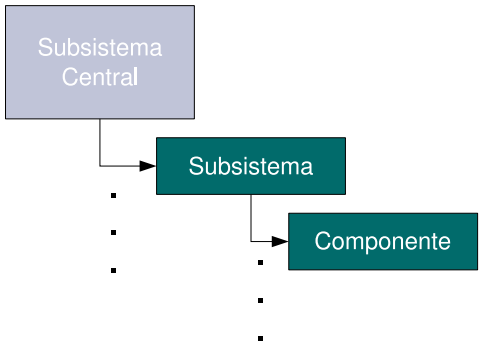


Figura 15.63: Jerarquía de supervisión de la ejecución de los procesos lanzados

AUV. La iniciación de la ejecución se comunicará al AUV una vez esté listo —i. e. en un medio líquido o en una plataforma de prueba.

2. *Fase de control del resto de subsistemas* → Una vez se haya iniciado la ejecución del sistema del AUV, el Subsistema Central quedará encargado de realizar tareas de un índole diferente, orientadas a la supervisión de la correcta ejecución de los procesos lanzados —otra cosa diferente son las tareas que realizará el Subsistema de Supervisión (ver sección 15.8), que son deliberativas y relacionadas con la tipología del AUV y la misión.

   a) El proceso del Subsistema Central se encagará de controlar o velar por que los demás procesos que se han lanzado sigan funcionando y no mueran, como si fuese un *superservidor*, que los relanza cada vez que mueren. En cierto modo se podría plantear una gestión jerárquica, donde a su vez, los subsistemas velan por los procesos que ellos lanzan, como se muestra en la figura 15.63. Sin embargo, esta supervisión sólo se aplicará en casos muy simples, ya que es el Subsistema de Supervisión el que debe gestionar las acciones oportunas. Por ello, el Subsistema Central se limitará a evitar que dejen de funcionar subsistemas críticos, como precisamente el Subsistema de Supervisión —y quizás otros.

3. Al terminar la misión se deberá volver a un estado inicial, i. e. antes de iniciar la ejecución de los planes, tras la inicialización del Sistema Operativo y la carga de los planes de la misión. Siempre será posible cambiar los planes de la misión e incluso gestionar una lista de planes, indicando cuál se desea que sea el activo.

Finalmente, hay que indicar que el sistema del AUV nunca debe terminar o morir, pues en tal caso el AUV quedaría inutilizable, i. e. simplemente sería un equipo con un Sistema Operativo ejecutándose, pero sin las funcionalidades propias del AUV. Para que no muera, nunca se usarán las herramientas que el Sistema Operativo proporciona para estos casos, por si se produjera un error que terminara su ejecución y se tuviera que reiniciar. Además, tras terminarse la misión, el sistema del AUV no se termina, sino que esperará por nuevas misiones o la indicación de que comience una nueva ejecución, como se mencionó previamente; o simplemente para realizar un apagado normal del sistema

del AUV, que también realizará el apagado del equipo —se enviará la orden de apagado `halt` al Sistema Operativo, en el caso de *Linux*.

Parte IV

Resultados

Capítulo 16

Especificación de la Misión

Nuestra recompensa se encuentra en el esfuerzo y no en el resultado.
Un esfuerzo total es una victoria completa.

— Mahatma Gandhi
1869-1948 (Político y pensador indio)

Para ilustrar la potencialidad de la especificación de la misión con la arquitectura de **planes de la misión**, se ha elaborado un misión de ejemplo. Esta misión intenta ser simple, a la vez que cubre todos los planes y aspectos que puede tener una misión de exploración oceanográfica típica. Se mostrarán las especificaciones XML realizadas para cada elemento de la misión, al mismo tiempo que se comenta que se pretende hacer en la misma.

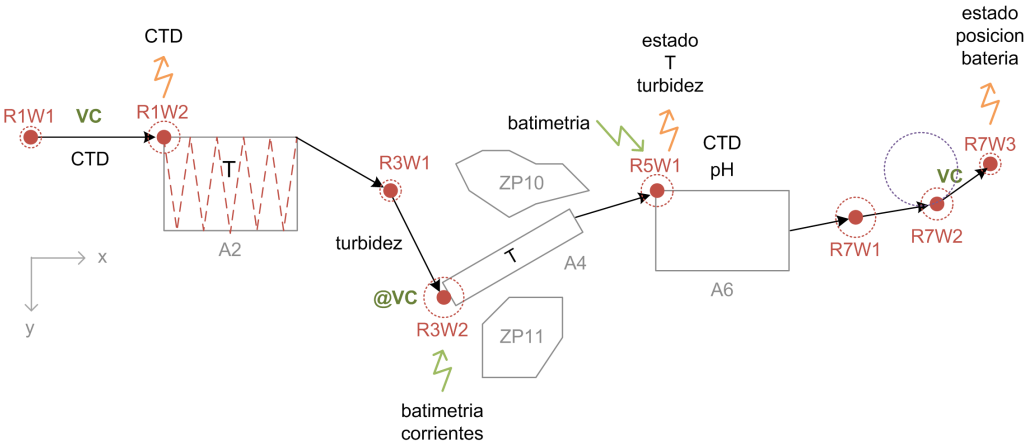
Antes de entrar en los detalles de la especificación de la misión se mostrará cómo puede definirse ésta. Esta tarea puede simplificarse o facilitarse mediante el uso de herramientas gráficas de especificación de misiones, como es el caso del Proyecto Fin de Carrera denominado *planificador* (véase el Apéndice A). Como no se ha dispuesto de esta herramienta durante el desarrollo de este PFC, se ha realizado un diagrama ilustrativo equivalente, con una representación similar a la que suelen usar este tipo de herramientas.

En esta especificación de la misión también se incluye el plan de *log* (PdL), encargado de configurar el registro del sistema, el cual se ha implementado con *log4cxx*. Aunque este elmento se incluye en la misión, en realidad se trata de una especificación interna, cuya definición la hará el desarrollador del sistema. La misión debe contener el PdL para que éste sea enviado al vehículo y se configure adecuadamente el registro del sistema. Por ello, el *planificador* no ofrecerá herramientas para su especificación y ni siquiera hará referencia a él durante la edición de la misión. Simplemente, internamente dispondrá de un modelo básico de este plan, que añadirá por defecto a la misión; si el usuario quiere modificarlo y seleccionar un PdL concreto de una lista, el *planificador* podrá ofrecer estas posibilidades mediante un modo avanzado, orientados para desarrolladores del sistema

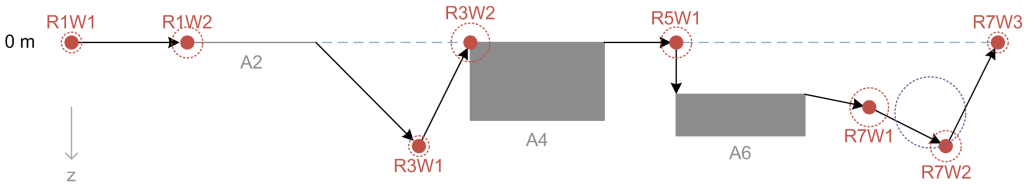
del vehículo que ejecutará la misión.

16.1. Edición de la Misión sobre un Mapa

La edición de la misión se realiza con una herramienta gráfica sobre un mapa de la zona de exploración. La Figura 16.1 muestra la visión cenital (a) y el perfil (b) del terreno, sobre el que se edita la misión. Lo más habitual es comenzar definiendo las rutas, áreas y zonas prohibidas del plan de navegación (PdN), y seguidamente definir el resto de planes sobre ellas. Esto permite indicar gráficamente y de forma muy intuitiva, cierta información de otros planes, v. g. las medidas que debe tomar el vehículo a lo largo de la misión, las medidas o datos que debe enviar o recibir en determinados puntos, etc.



(a) Especificación del PdN y parte del PdM y PdC; el resto se define mediante cuadros de diálogo



(b) Perfil de profundidad del PdN, correspondiente a la figura (a)

Figura 16.1: Edición de la Misión sobre un Mapa

Sin embargo, este tipo de interfaz no permite la especificación de todas las características de los planes de la misión. Por ello, se dispondrá de cuadros de diálogo para tomar dicha información, que aquí sólo veremos directamente especificada en los ficheros XML de cada plan. En la Figura 16.1, aparte del PdN, se ha especificado parte de las tareas de los planes de comunicación y medición (PdC y PdM, respectivamente). Pero éstos, como el resto de planes e incluso el propio PdN, requieren de otros mecanismos adicionales para terminar de dar soporte a toda la potencialidad de representación de misiones que ofrece la especificación diseñada.

A modo de resumen y de acuerdo con la arquitectura de los **planes de la misión**, habitualmente se especificarán los siguientes elementos:

Misión Se indicarán los diferentes planes y tablas de parámetros que forman la misión

completa. En el Algoritmo 16.1 se puede ver como esto se reduce a indicar las referencias de los ficheros XML de cada uno de ellos. Evidentemente, es responsabilidad del **planificador** facilitar esta tarea, la cual puede llegar a ser transparente al usuario, manejando los nombres de los planes en lugar de las rutas de los ficheros XML.

PdA En el plan de almacenamiento basta con indicar las medidas que se desea que el vehículo almacene, con la posibilidad de indicar el número máximo de muestras. Esto se hará una vez definida toda la lista de medidas disponibles en el sistema y será habitual que el **planificador** facilite esta tarea ofreciendo la posibilidad de activar el almacenamiento de todas aquellas medidas que se hayan especificado en el PdM —para su medición.

PdC Sobre la especificación gráfica del PdN en el mapa se pueden indicar los puntos donde se desea que el vehículo se comunique. A continuación, para cada uno de estos puntos se facilitará la información sobre el nombre de la medida o el dato y el destinatario o la fuente, según sea el tipo de comunicación indicado (envío o recepción, respectivamente). También es común especificar la comunicación en base a determinadas condiciones o excepciones, lo cual se hará al margen del mapa.

PdM Sobre la especificación gráfica del PdN en el mapa se pueden indicar las mediciones que deben hacerse en cada punto. Para cada medición indicada se introducirá la información de la misma, i. e. frecuencia y resolución de muestreo, número máximo de muestras a tomar (si procede) y, opcionalmente, la lista de sensores que puede usarse. También es común especificar las mediciones en base a determinadas condiciones o intervalos, lo cual se hará al margen del mapa.

PdN Las rutas, áreas y zonas prohibidas se dibujan sobre el mapa y para cada una de ellas se pueden configurar gráficamente algunos de sus parámetros, v. g. incertidumbre de los *waypoints*, modo de recorrido de las áreas, etc. Esto podrá apoyarse en cuadros de diálogo para aquellos parámetros difíciles de representar gráficamente, v. g. tipo de seguimiento de una medida, número de transectos en que se divide el recorrido de un área, etc.

PdS El plan de supervisión requiere unos conocimientos más profundos del sistema, pues en su especificación se manejan todas las acciones soportadas en la misión, usando una interfaz homogénea y menos intuitiva que en el resto de planes. También permite especificaar la ejecución de determinados planes de contingencia o, incluso, planes de la misión. Es común que se definan determinadas condiciones o excepciones, que constituyen el proceso de supervisión del sistema, y las acciones a realizar bajo las mismas.

Parámetros Los planes de la misión se complementan con la elaboración de tablas de parámetros que facilitan la modificación de los elementos especificados. El *planificador* podrá facilitar enormemente la definición de parámetros, ya que permitirá seleccionar estos elementos y asignarles un parámetro, con un nombre y valor por defecto. Estos parámetros podrían visualizarse en el mapa, como es el caso del parámetro **VC** que aparece en la Figura 16.1 (a).

```
<?xml version="1.0"?>
<mision id="1" nombre="Ejemplo Exploracion"
                xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                xmlns="http://www.mision.com"
                xsi:schemaLocation="http://www.mision.com
                                    ../esquemas/mision.xsd">

  <pda fichero="../planes/pda/pda.xml"/>
  <pdc fichero="../planes/pdc/pdc.xml"/>
  <pdm fichero="../planes/pdm/pdm.xml"/>
  <pdn fichero="../planes/pdn/pdn.xml"/>
  <pds fichero="../planes/pds/pds.xml"/>
  <pdl fichero="../log/cfg/auv.xml"/>

  <tablaParametros fichero="../tablaparametros/tablaparametros.xml"/>
</mision>
```

Algoritmo 16.1: Misión: Ejemplo de Exploración

En el diagrama de flujo de la Figura 16.2 se muestra la secuencia de edición de una misión típica. Como resultado se obtendría una misión completa y lista para validarse y enviarse al vehículo que la ejecutará. Este flujo de edición es orientativo, de modo que es posible realizar la especificación de la misión en otro orden, siendo el resultado final el mismo.

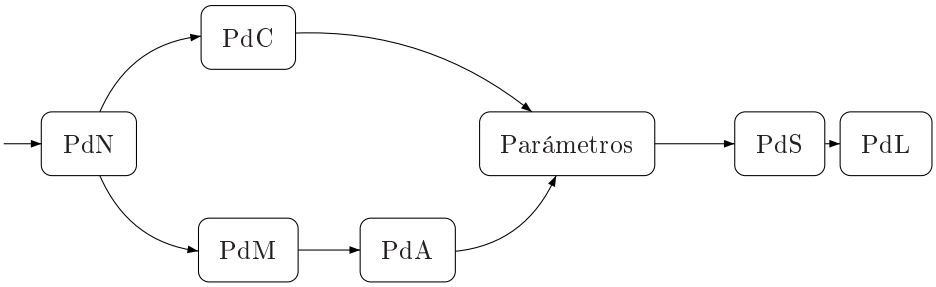


Figura 16.2: Flujo del proceso de edición de una misión de exploración oceanográfica típica

A modo de resumen, con la misión mostrada en la Figura 16.1 se intenta especificar lo siguiente:

1. Se indican las rutas y áreas por las que debe ir pasando el vehículo. Se incluyen dos zonas prohibidas (**ZP10** y **ZP11**) por las que el vehículo de evitar pasar. En la Sección 16.5 se explica en detalle el PdN, para indicar que navegación se pretende conseguir durante cada fase de la misión de exploración.

2. A largo de los transectos de las rutas y dentro de las áreas, se indican las medidas que el vehículo debe muestrear, como parte de la especificación del PdM (véase la Sección 16.4 para más detalles).

3. En determinados puntos se indica que el vehículo debe comunicarse. Esto se especifica indicando los datos que intervienen en la comunicación y el tipo de comunicación, i. e. envío o recepción (ilustrado gráficamente).

En las siguientes secciones, para cada plan de la misión y las tablas de parámetros, se explica lo que se pretende definir en la misión de ejemplo, haciendo referencia a la

especificación gráfica de la Figura 16.1 si en ella se ha indicado algo, y se muestra el fichero XML resultante.

16.2. Plan de Almacenamiento

En el PdA se indica una lista de tareas de almacenamiento para indicar aquellas medidas que deben almacenar bajo ciertas condiciones. De acuerdo con las mediciones indicadas en el PdM, hay varias tareas de almacenamiento de las mismas. Además, también se tienen en cuenta los datos que hay que comunicar, según lo indicado en el PdC, pues éstos deben haber sido almacenados; en caso contrario el proceso de validación de la misión detectará esta incompatibilidad, avisando que el PdC no podrá realizarse correctamente. Precisamente, las tareas del PdA cuyo identificador (**id**) va del 1 al 5, se encargan de evitar este problema.

En el Algoritmo 16.2 se muestra el fichero XML resultante de la siguiente lista de tareas de almacenamiento especificadas:

Tarea 1 Almacenamiento de las medidas **temperatura** y **posicion** durante toda la misión. Se usa un período de inhibición infinito (**INF**) porque, al no haber disparadores, la tarea estará siempre activa.

Tarea 2 Almacenamiento de la **conductividad** y la **profundidad** en la ruta 1. Esto permite que el vehículo pueda enviar estos datos posteriormente. Se usa un período de inhibición de 1min en base a la dinámica de navegación por la ruta.

Tarea 3 Almacenamiento de la **conductividad** y la **profundidad** en el área 6. Esto permite que el vehículo pueda enviar estos datos posteriormente. Se usa un período de inhibición de 10min en base a la dinámica de navegación por el área. Las tareas 2 y 3, definen en su conjunto el almacenamiento de la conductividad y la profundidad, como un **o** lógico.

Tarea 4 Almacenamiento de la **turbidez** en la ruta 1. Esto permite que el vehículo pueda enviar este dato posteriormente. Se usa un período de inhibición de 1min en base a la dinámica de navegación por la ruta.

Tarea 5 Almacenamiento del **pH** en el área 6. Esto permite que el vehículo disponga de estos datos almacenados para cuando se recupere el vehículo después de terminar la misión. Se usa un período de inhibición de 10min en base a la dinámica de navegación por el área.

Tarea 6 Almacenamiento de la **bateria** si su valor baja del 25 %. Además, se define un intervalo para que sólo se almacene cada 10min. Se usa un período de inhibición de 10min en base al período del intervalo contenido en los disparadores de esta tarea.

Tarea 7 Almacenamiento de la **velocidad** y el **estado** de la misión cuando el vehículo ha entrado en una zona prohibida, lo cual se especifica con la excepción **enZonaProhibida**. Se usa un período de inhibición de 2min en base a la dinámica de navegación del vehículo para salir de la zona prohibida.

```
1 <?xml version="1.0"?>
2 <planDeAlmacenamiento id="1" nombre="almacenamiento islotes"
3                       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

```
4                       xmlns:disparadores="http://www.disparadores.com"
5                       xmlns="http://www.pda.com"
6                       xsi:schemaLocation="http://www.pda.com
7                                           ../../esquemas/pda.xsd">
8
9   <tarea id="1" nombre="almacenamiento de temperatura y posicion">
10    <disparadores:disparadores/>
11    <acciones>
12      <almacenar id="1" medida="temperatura"/>
13      <almacenar id="2" medida="posicion"/>
14    </acciones>
15    <periodoInhibicion valor="INF" unidad="minuto"/>
16  </tarea>
17
18  <!-- 'O' logico, entre la tarea 2 y 3 -->
19  <tarea id="2" nombre="almacenamiento de conductividad y profundidad
20                        en la ruta 1">
21    <disparadores:disparadores>
22      <disparadores:condicion id="1" medida="ruta"
23                              operador="eq" valor="1" unidad="ruta"/>
24    </disparadores:disparadores>
25    <acciones>
26      <almacenar id="1" medida="conductividad"/>
27      <almacenar id="2" medida="profundidad"/>
28    </acciones>
29    <periodoInhibicion valor="1" unidad="minuto"/>
30  </tarea>
31
32  <tarea id="3" nombre="almacenamiento de conductividad y profundidad
33                        en el area 6">
34    <disparadores:disparadores>
35      <disparadores:condicion id="1" medida="area"
36                              operador="eq" valor="6" unidad="area"/>
37    </disparadores:disparadores>
38    <acciones>
39      <almacenar id="1" medida="conductividad"/>
40      <almacenar id="2" medida="profundidad"/>
41    </acciones>
42    <periodoInhibicion valor="10" unidad="minuto"/>
43  </tarea>
44
45  <tarea id="4" nombre="almacenamiento de turbidez">
46    <disparadores:disparadores>
47      <disparadores:condicion id="1" medida="ruta"
48                              operador="eq" valor="3" unidad="ruta"/>
49    </disparadores:disparadores>
50    <acciones>
51      <almacenar id="1" medida="turbidez"/>
52    </acciones>
53    <periodoInhibicion valor="1" unidad="minuto"/>
54  </tarea>
55
56  <tarea id="5" nombre="almacenamiento de pH">
57    <disparadores:disparadores>
58      <disparadores:condicion id="1" medida="area"
59                              operador="eq" valor="6" unidad="area"/>
60    </disparadores:disparadores>
61    <acciones>
62      <almacenar id="1" medida="pH"/>
63    </acciones>
64    <periodoInhibicion valor="10" unidad="minuto"/>
65  </tarea>
66
67  <tarea id="6" nombre="almacenamiento de bateria baja">
68    <disparadores:disparadores>
69      <disparadores:condicion id="1" medida="bateria"
70                              operador="lt" valor="25" unidad="%"/>
71      <disparadores:intervalo id="2" medida="tiempo"
72                              inicio="0" fin="INF" periodo="10"
73                              unidad="minuto"/>
74    </disparadores:disparadores>
75    <acciones>
76      <almacenar id="1" medida="bateria"/>
77    </acciones>
78    <periodoInhibicion valor="10" unidad="minuto"/>
79  </tarea>
80
81  <tarea id="7" nombre="almacenamiento de velocidad y estado
82                        en zona prohibida">
83    <disparadores:disparadores>
```

```
84        <disparadores:excepcion id="1" nombre="enZonaProhibida"/>
85      </disparadores:disparadores>
86      <acciones>
87        <almacenar id="1" medida="velocidad"/>
88        <almacenar id="1" medida="estado"/>
89      </acciones>
90      <periodoInhibicion valor="2" unidad="minuto"/>
91    </tarea>
92  </planDeAlmacenamiento>
```

Algoritmo 16.2: PdA (Ejemplo de Exploración)

16.3. Plan de Comunicación

En el PdC se indica una lista de tareas de comunicación para indicar las medidas o datos que deben comunicarse bajo ciertas condiciones. En la Figura 16.1 se observa como se indican los puntos dónde debe realizarse la comunicación, así como el tipo de comunicación (envío o recepción) y los nombres de las medidas o datos. Se utilizan los elementos de la especificación del PdN para indicar dichos puntos, de modo que el **planificador** podrá validar con mayor facilidad que los dispositivos de comunicación pueden operar en el lugar indicado —v. g. en ocasiones el equipamiento del vehículo sólo permite la comunicación en superficie. Las tares con identificador del 1 al 4 aparecen especificadas en el mapa. En el Algoritmo 16.3 se muestra el fichero XML resultante de la siguiente lista de tareas de comunición especificadas:

Tarea 1 Envío de los datos de **conductividad**, **temperatura** y **profundidad** en el *waypoint* 2 de la ruta 1; estos datos han sido medidos previamente, de acuerdo con la especificación del PdM. Estos tres datos son enviados al **planificador**.

Tarea 2 Recepción de los datos de **batimetria** y **corrientes** en el *waypoint* 2 de la ruta 3. La fuente de estos datos es el **servidorInformacion**, que para las corrientes emplea el puerto 2001. Esta información corresponderá al área 4 y se almacenará en el sistema para su uso posterior.

Tarea 3 Envío de las medidas del **estado** de la misión y la **temperatura** en el *waypoint* 1 de la ruta 5. También se envía el dato de **turbidez** (medido en la ruta 3). Estas medidas y datos se envían al **planificador**. También se recibe el dato de **batimetria**, del **servidorInformacion**; esta información batimétrica corresponderá al área 6.

Tarea 4 Envío de las medidas del **estado** de la misión, **posicion** y **bateria** en el *waypoint* 2 de la ruta 7. Estas medidas se envían al **planificador**, y en el caso del estado de la misión y la posición también se envían a la dirección IP **192.168.1.35**, al puerto 1000 y 1001, respectivamente.

Tarea 5 Envío de la **temperatura** y **posicion** cuando la **temperatura** es estrictamente mayor de 22°C, estando en el área 2.

Tarea 6 Envío de las medidas del **estado** de la misión, **posicion** y **bateria** si la batería baja del 10 %. Los destinatarios de estas medidas son los mismos que en la tarea 4.

Los períodos de inhibición utilizados en las tareas de comunicación suelen cubrir el tiempo de envío o recepción. No obstante, en casos como las comunicaciones en un

waypoint, normalmente el vehículo ya no estará en el *waypoint* cuando se vuelvan a comprobar los disparadores. Esto hace que el tiempo de inhibición resulte irrelevante en estos casos.

```
<?xml version="1.0"?>
<planDeComunicacion id="1" nombre="comunicacion islotes"
                    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                    xmlns:disparadores="http://www.disparadores.com"
                    xmlns="http://www.pdc.com"
                    xsi:schemaLocation="http://www.pdc.com
                                        ../../esquemas/pdc.xsd">
  <tarea id="1" nombre="envio de datos de CTD">
    <disparadores:disparadores>
      <disparadores:condicion id="1" medida="ruta"
                              operador="eq" valor="1" unidad="ruta"/>
      <disparadores:condicion id="2" medida="waypoint"
                              operador="eq" valor="2" unidad="waypoint"/>
    </disparadores:disparadores>
    <acciones>
      <enviarDato id="1" dato="conductividad" destinatario="planificador"/>
      <enviarDato id="2" dato="temperatura" destinatario="planificador"/>
      <enviarDato id="3" dato="presion" destinatario="planificador"/>
    </acciones>
    <periodoInhibicion valor="5" unidad="minuto" />
  </tarea>

  <tarea id="2" nombre="recepcion batimetria y corrientes de estrecho">
    <disparadores:disparadores>
      <disparadores:condicion id="1" medida="ruta"
                              operador="eq" valor="3" unidad="ruta"/>
      <disparadores:condicion id="2" medida="waypoint"
                              operador="eq" valor="2" unidad="waypoint"/>
    </disparadores:disparadores>
    <acciones>
      <recibirDato id="1" dato="batimetria" fuente="servidorInformacion"/>
      <recibirDato id="2" dato="corrientes" fuente="servidorInformacion:2001"/>
    </acciones>
    <periodoInhibicion valor="10" unidad="minuto" />
  </tarea>

  <tarea id="3" nombre="envio de datos de turbidez y
                        recepcion batimetria de mar abierto">
    <disparadores:disparadores>
      <disparadores:condicion id="1" medida="ruta"
                              operador="eq" valor="5" unidad="ruta"/>
      <disparadores:condicion id="2" medida="waypoint"
                              operador="eq" valor="1" unidad="waypoint"/>
    </disparadores:disparadores>
    <acciones>
      <enviarMedida id="1" medida="estado" destinatario="planificador"/>
      <enviarMedida id="2" medida="temperatura" destinatario="planificador"/>
      <enviarDato id="3" dato="turbidez" destinatario="planificador"/>
      <recibirDato id="4" dato="batimetria" fuente="servidorInformacion"/>
    </acciones>
    <periodoInhibicion valor="10" unidad="minuto" />
  </tarea>

  <tarea id="4" nombre="finalizacion de la mision">
    <disparadores:disparadores>
      <disparadores:condicion id="1" medida="ruta"
                              operador="eq" valor="7" unidad="ruta"/>
      <disparadores:condicion id="2" medida="waypoint"
                              operador="eq" valor="2" unidad="waypoint"/>
    </disparadores:disparadores>
    <acciones>
      <enviarMedida id="1" medida="estado"
                    destinatario="planificador, 192.168.1.35:1000"/>
      <enviarMedida id="2" medida="posicion"
                    destinatario="planificador, 192.168.1.35:1001"/>
      <enviarMedida id="3" medida="bateria" destinatario="planificador"/>
    </acciones>
    <periodoInhibicion valor="1" unidad="minuto" />
  </tarea>

  <!-- No especificado en el mapa -->
  <tarea id="5" nombre="aguas calidas">
    <disparadores:disparadores>
      <disparadores:condicion id="1" medida="area"
                              operador="eq" valor="2" unidad="ruta"/>
```

```
76        <disparadores:condicion id="2" medida="temperatura"
77                                operador="gt" valor="22"
78                                unidad="grado centigrado"/>
79      </disparadores:disparadores>
80      <acciones>
81        <enviarMedida id="1" medida="temperatura" destinatario="planificador"/>
82        <enviarMedida id="2" medida="posicion" destinatario="planificador"/>
83      </acciones>
84      <periodoInhibicion valor="30" unidad="segundo" />
85    </tarea>
86
87    <tarea id="6" nombre="bateria baja">
88      <disparadores:disparadores>
89        <disparadores:condicion id="1" medida="bateria"
90                                operador="lt" valor="10" unidad="%"/>
91      </disparadores:disparadores>
92      <acciones>
93        <!-- estado, indica SOS -->
94        <enviarMedida id="1" medida="estado"
95                      destinatario="planificador, 192.168.1.35:1000"/>
96        <enviarMedida id="2" medida="posicion"
97                      destinatario="planificador, 192.168.1.35:1001"/>
98        <enviarMedida id="3" medida="bateria" destinatario="planificador"/>
99      </acciones>
100     <periodoInhibicion valor="30" unidad="minuto" />
101   </tarea>
102 </planDeComunicacion>
```

Algoritmo 16.3: PdC (Ejemplo de Exploración)

16.4. Plan de Medición

En el PdM se indica una lista de tareas de medición para indicar las medidas que deben muestrearse bajo ciertas condiciones. En la Figura 16.1 se observa como se indican las medidas a tomar a lo largo de diversos transectos de las rutas y en las áreas especificadas en el PdN. En el Algoritmo 16.4 se muestra el fichero XML resultante de la siguiente lista de tareas de medición especificadas:

Tarea 1 Medición de **conductividad**, **temperatura** y **profundidad** en el transecto 1 de la ruta 1. Se utiliza una frecuencia de muestreo de 1/60Hz en todas las mediciones. Las resoluciones requeridas son 1mS, 0.1°C y 1m, respectivamente.

Tarea 2 Medición de **temperatura** en el área 2. Se especifica el sensor **SBE 37SIP** y la configuración **sync.cfg**, que son las que se desea usar. Se utiliza una frecuencia de muestreo de 1/600Hz y una resolución de 0.01°C.

Tarea 3 Medición de **turbidez** en el transecto 1 de la ruta 3. Se utiliza una frecuencia de muestreo de 1/300Hz y una resolución de 0.01NTU.

Tarea 4 Medición de **conductividad**, **temperatura** y **profundidad** en el área 6. Se utiliza una frecuencia de muestreo de 1/1800Hz en todas las mediciones. Las resoluciones requeridas son 0.1S, 1°C y 5dm, respectivamente. También se mide el **pH** a 1/900Hz y 0.1 de resolución.

Tarea 5 Medición de **pH** cuando éste es ácido o básico, i. e. menor que 4 o mayor que 8, respectivamente. Adicionalmente se mide la **posicion** y **temperatura** cuando ocurre esto. Sólo se toman 10 muestras.

Tarea 6 Medición de **temperatura** cuando es menor de 15.5°C en el área 5.

Tarea 7 Medición de **salinidad** cuando la temperatura es menor de 15.5°C en el área 2. Además, sólo se medirá 1 muestra cada 10m, desde que el vehículo haya recorrido 500m y hasta que alcance los 2000m, tal y como se especifica con un **intervalo**.

Tarea 8 Medición de 5 muestras de **temperatura** y **posicion** cuando se produce la excepción de **agotamientoBaterias**. Con ello se intenta tomar una última ráfaga de muestras en la misión.

Cuando se especifica la medición en base a los elementos del PdN, se indica un período de inhibición de acuerdo con la dinámica de recorrido del vehículo en las rutas o áreas, para dejar de medir cuando se hayan terminado. En el resto de casos dependen del número de muestras tomadas y lo que desee especificarse.

```
<?xml version="1.0"?>
<planDeMedicion id="1" nombre="medicion islotes"
                xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                xmlns:disparadores="http://www.disparadores.com"
                xmlns:equipamiento="http://www.equipamiento.com"
                xmlns="http://www.pdm.com"
                xsi:schemaLocation="http://www.pdm.com
                                    ../../esquemas/pdm.xsd">

  <tarea id="1" nombre="medicion de CTD">
    <disparadores:disparadores>
      <disparadores:condicion id="1" medida="ruta"
                              operador="eq" valor="1" unidad="ruta"/>
      <disparadores:condicion id="2" medida="transecto"
                              operador="eq" valor="1" unidad="transecto"/>
    </disparadores:disparadores>
    <acciones>
      <medir id="1" medida="conductividad">
        <frecuencia valor="1/60" unidad="Hz"/>
        <resolucion valor="1" unidad="mS"/>
      </medir>
      <medir id="2" medida="temperatura">
        <frecuencia valor="1/60" unidad="Hz"/>
        <resolucion valor="0.1" unidad="grado centigrado"/>
      </medir>
      <medir id="3" medida="profundidad">
        <frecuencia valor="1/60" unidad="Hz"/>
        <resolucion valor="1" unidad="m"/>
      </medir>
    </acciones>
    <periodoInhibicion valor="1" unidad="minuto" />
  </tarea>

  <tarea id="2" nombre="medicion de temperatura">
    <disparadores:disparadores>
      <disparadores:condicion id="1" medida="area"
                              operador="eq" valor="2" unidad="area"/>
    </disparadores:disparadores>
    <acciones>
      <medir id="1" medida="temperatura">
        <frecuencia valor="1/600" unidad="Hz"/>
        <resolucion valor="0.01" unidad="grado centigrado"/>
        <sensor id="1" nombre="SBE 37SIP">
          <equipamiento:configuracion nombre="sync.cfg"/>
        </sensor>
      </medir>
    </acciones>
    <periodoInhibicion valor="2" unidad="minuto"/>
  </tarea>

  <tarea id="3" nombre="medicion de turbidez">
    <disparadores:disparadores>
      <disparadores:condicion id="1" medida="ruta"
                              operador="eq" valor="3" unidad="area"/>
      <disparadores:condicion id="2" medida="transecto"
                              operador="eq" valor="1" unidad="transecto"/>
    </disparadores:disparadores>
    <acciones>
      <medir id="1" medida="turbidez">
        <frecuencia valor="1/300" unidad="Hz"/>
        <resolucion valor="0.01" unidad="NTU"/>
```

```
62        </medir>
63      </acciones>
64      <periodoInhibicion valor="2" unidad="minuto"/>
65    </tarea>
66
67    <tarea id="4" nombre="medicion de CTD y pH">
68      <disparadores:disparadores>
69        <disparadores:condicion id="1" medida="area"
70                                operador="eq" valor="6" unidad="area"/>
71      </disparadores:disparadores>
72      <acciones>
73        <medir id="1" medida="conductividad">
74          <frecuencia valor="1/1800" unidad="Hz"/>
75          <resolucion valor="0.1" unidad="S"/>
76        </medir>
77        <medir id="2" medida="temperatura">
78          <frecuencia valor="1/1800" unidad="Hz"/>
79          <resolucion valor="1" unidad="grado centigrado"/>
80        </medir>
81        <medir id="3" medida="profundidad">
82          <frecuencia valor="1/1800" unidad="Hz"/>
83          <resolucion valor="5" unidad="dm"/>
84        </medir>
85        <medir id="4" medida="pH">
86          <frecuencia valor="1/900" unidad="Hz"/>
87          <resolucion valor="0.1" unidad="pH"/>
88        </medir>
89      </acciones>
90      <periodoInhibicion valor="10" unidad="minuto"/>
91    </tarea>
92
93    <!-- No especificado en el mapa -->
94    <tarea id="5" nombre="medicion pH acido/basico">
95      <disparadores:disparadores>
96        <disparadores:condicion id="1" medida="pH"
97                                operador="lt" valor="4" unidad="pH"/>
98        <disparadores:condicion id="1" medida="pH"
99                                operador="gt" valor="8" unidad="pH"/>
100     </disparadores:disparadores>
101     <acciones>
102       <medir id="1" medida="pH" muestras="10">
103         <frecuencia valor="1" unidad="Hz"/>
104         <resolucion valor="0.1" unidad="pH"/>
105       </medir>
106       <medir id="2" medida="posicion" muestras="10">
107         <frecuencia valor="1" unidad="Hz"/>
108         <resolucion valor="1" unidad="cm"/>
109       </medir>
110       <medir id="3" medida="temperatura" muestras="10">
111         <frecuencia valor="1" unidad="Hz"/>
112         <resolucion valor="1" unidad="miligrado centigrado"/>
113       </medir>
114     </acciones>
115     <periodoInhibicion valor="30" unidad="segundo"/>
116   </tarea>
117
118   <tarea id="6" nombre="medicion aguas frias">
119     <disparadores:disparadores>
120       <disparadores:condicion id="1" medida="area"
121                               operador="lt" valor="4" unidad="area"/>
122       <disparadores:condicion id="2" medida="temperatura"
123                               operador="lt" valor="15.5"
124                               unidad="grado centigrado"/>
125     </disparadores:disparadores>
126     <acciones>
127       <medir id="1" medida="temperatura">
128         <frecuencia valor="1/60" unidad="Hz"/>
129         <resolucion valor="0.1" unidad="grado centigrado"/>
130       </medir>
131     </acciones>
132     <periodoInhibicion valor="20" unidad="segundo"/>
133   </tarea>
134
135   <tarea id="7" nombre="medicion salinidad de aguas frias">
136     <disparadores:disparadores>
137       <disparadores:condicion id="1" medida="area"
138                               operador="lt" valor="2" unidad="area"/>
139       <disparadores:condicion id="2" medida="temperatura"
140                               operador="lt" valor="15.5"
141                               unidad="grado centigrado"/>
```

```
142         <disparadores:intervalo id="3" medida="odometria"
143                                 inicio="500" fin="2000" periodo="10" unidad="m"/>
144       </disparadores:disparadores>
145       <acciones>
146         <medir id="1" medida="salinidad" muestras="1">
147           <frecuencia valor="1" unidad="Hz"/>
148           <resolucion valor="0.01" unidad="PSU"/>
149         </medir>
150       </acciones>
151       <periodoInhibicion valor="15" unidad="segundo"/>
152     </tarea>
153
154     <tarea id="8" nombre="medicion con bateria baja">
155       <disparadores:disparadores>
156         <disparadores:excepcion id="1" nombre="agotamientoBaterias"/>
157       </disparadores:disparadores>
158       <acciones>
159         <medir id="1" medida="temperatura" muestras="5">
160           <frecuencia valor="1" unidad="Hz"/>
161           <resolucion valor="0.1" unidad="grado centigrado"/>
162         </medir>
163         <medir id="2" medida="posicion" muestras="5">
164           <frecuencia valor="2" unidad="Hz"/>
165           <resolucion valor="20" unidad="cm"/>
166         </medir>
167       </acciones>
168       <periodoInhibicion valor="10" unidad="minuto"/>
169     </tarea>
170 </planDeMedicion>
```

Algoritmo 16.4: PdM (Ejemplo de Exploración)

16.5. Plan de Navegación

La especificación del PdN es la mostrada en la Figura 16.1, aunque algunos parámetros se especificarán con cuadros de diálogo, para obtener el resultado del fichero XML mostrado en el Algoritmo 16.5. En éste aparece la especificación de las siguientes rutas, áreas y zonas prohibidas:

Ruta 1 Ruta de salida del vehículo, compuesta por los *waypoints* 1 (**R1W1**) y 2 (**R1W2**). Se define la posición de los mismos y el transecto que los une, donde se indica la velocidad crucero deseada.

Área 2 Área **A2** de muestreo de temperatura, que se recorre en *zigzag* utilizando 10 transectos. Se sigue un rumbo de 270° al describir los transectos. Este área es a profundidad constante, en superficie. Se estima una duración de $1.5\text{h} \pm 15\,\%$ para que se exlpore el área.

Ruta 3 Ruta de muestreo de turbidez, compuesta por los *waypoints* 1 (**R3W1**) y 2 (**R3W2**). Se define la posición de los mismos y el transecto que los une. También se controla la orientación que debe tener el vehículo en los *waypoints* de cara a estar correctamente orientado para acometer la navegación de los siguientes elementos del PdN. Se usa el parámetro de la mision **VC** para indicar la velocidad que debe tenerse en el *waypoint* 2.

Área 4 Área **A4** de seguimiento de la medida **temperatura**, durante $30\text{min} \pm 10\,\%$. Se definen las tres fases habituales en la especificación del seguimiento de medidas, **búsqueda**, **mantenimiento** y **finalización**, con el recorrido 1 describiendo transectos en *zigzag*, el recorrido 2 siguiendo la **temperatura** con una función de gradiente y el recorrido 3. Se busca la medida mientras la temperatura sea estrictamente menor de 20°C, se mantiene mientras esté entre 20°C y sea estrictamente

menor de 32°C, y se termina el seguimiento cuando se superan los 32°C. Se observa como estas condiciones son disjuntas y totales (véase la Propiedad 13.3).

Ruta 5 Ruta para abandonar el estrecho que forma las zonas prohibidas, compuesta por un único *waypoint* **R5W1**, de modo que no se especifica ningún transecto.

Área 6 Área **A6** de muestreo de pH, que se recorre a la deriva. Se define para una profundidad que va de 80m a 110m y con una duración de $2\text{h} \pm 5\,\%$.

Ruta 7 Ruta para alcanzar el punto de encuentro, donde termina la misión. Está compuesta por los *waypoints* 1 (**R7W1**), 2 (**R7W2**) y 3 (**R7W3**). Se definen dos transectos y en el *waypoint* 2 se proporciona el radio de interpolación, que determina cómo pasar del primer transecto al segundo.

Zona Prohibida 10 Zona prohibida **ZP10**, con los vértices que la definen.

Zona Prohibida 11 Zona prohibida **ZP11**, con los vértices que la definen.

```
<?xml version="1.0"?>
<planDeNavegacion id="1" nombre="exploracion islotes" repeticion="1"
                  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                  xmlns:disparadores="http://www.disparadores.com"
                  xmlns="http://www.pdn.com"
                  xsi:schemaLocation="http://www.pdn.com
                                      ../../esquemas/pdn.xsd">
  <ruta id="1" nombre="salida">
    <waypoint id="1">
      <pose>
        <posicion x="100" y="200" z="0" unidad="m"/>
        <orientacion roll="0" pitch="0" yaw="0" unidad="grado"/>
        <incertidumbre valor="1" unidad="m"/>
      </pose>
    </waypoint>
    <waypoint id="2">
      <pose>
        <posicion x="200" y="200" z="0" unidad="m"/>
        <orientacion roll="0" pitch="0" yaw="270" unidad="grado"/>
        <incertidumbre valor="2" unidad="m"/>
      </pose>
      <velocidad>
        <posicion x="0" y="3" z="0" unidad="nudo"/>
      </velocidad>
    </waypoint>
    <transecto id="1" inicio="1" fin="2">
      <velocidad>
        <posicion x="3" y="0" z="0" unidad="nudo"/>
      </velocidad>
    </transecto>
  </ruta>

  <area id="2" nombre="muestreo de temperatura">
    <vertice id="1" x="200" y="100" z="0" unidad="m"/>
    <vertice id="2" x="200" y="200" z="0" unidad="m"/>
    <vertice id="3" x="300" y="200" z="0" unidad="m"/>
    <vertice id="4" x="300" y="100" z="0" unidad="m"/>
    <tiempo valor="1.5" unidad="hora" holgura="15"/>
    <recorrido id="1">
      <disparadores:disparadores/>
      <transectos modo="zigzag" cantidad="10" ciclos="0">
        <tiempo valor="10" unidad="minuto"/>
        <profundidad valor="0" unidad="m"/>
        <angulo valor="270" unidad="grado"/>
      </transectos>
    </recorrido>
  </area>

  <ruta id="3" nombre="muestreo de turbidez">
    <waypoint id="1">
      <pose>
        <posicion x="400" y="250" z="120" unidad="m"/>
        <orientacion roll="0" pitch="70" yaw="315" unidad="grado"/>
```

```
54          <incertidumbre valor="1" unidad="m"/>
55        </pose>
56      </waypoint>
57      <waypoint id="2">
58        <pose>
59          <posicion x="450" y="400" z="0" unidad="m"/>
60          <orientacion roll="0" pitch="0" yaw="30" unidad="grado"/>
61          <incertidumbre valor="4" unidad="m"/>
62        </pose>
63        <velocidad>
64          <posicion x="@VC" y="-1" z="0" unidad="nudo"/>
65        </velocidad>
66      </waypoint>
67      <transecto id="1" inicio="1" fin="2">
68        <velocidad>
69          <tiempo valor="20" unidad="minuto"/>
70        </velocidad>
71      </transecto>
72    </ruta>
73
74    <area id="4" nombre="seguimiento de temperatura">
75      <vertice id="1" x="450" y="400" z="0" unidad="m"/>
76      <vertice id="2" x="460" y="420" z="0" unidad="m"/>
77      <vertice id="3" x="550" y="300" z="0" unidad="m"/>
78      <vertice id="4" x="540" y="280" z="0" unidad="m"/>
79      <profundidad minima="0" maxima="100" unidad="m"/>
80      <tiempo valor="30" unidad="minuto" holgura="25"/>
81      <!-- Busqueda -->
82      <recorrido id="1">
83        <disparadores:disparadores>
84          <disparadores:condicion id="1" medida="temperatura"
85                                  operador="lt" valor="20"
86                                  unidad="grado centigrado"/>
87        </disparadores:disparadores>
88        <transectos modo="zigzag" cantidad="50" ciclos="0">
89          <tiempo valor="1.5" unidad="minuto"/>
90          <profundidad valor="2" unidad="m"/>
91          <angulo valor="330" unidad="grado"/>
92        </transectos>
93      </recorrido>
94      <!-- Mantenimiento -->
95      <recorrido id="2">
96        <disparadores:disparadores>
97          <disparadores:condicion id="1" medida="temperatura"
98                                  operador="ge" valor="20"
99                                  unidad="grado centigrado"/>
100         <disparadores:condicion id="2" medida="temperatura"
101                                 operador="lt" valor="32"
102                                 unidad="grado centigrado"/>
103       </disparadores:disparadores>
104       <seguimiento>
105         <funcion medida="temperatura" modo="gradiente"/>
106       </seguimiento>
107     </recorrido>
108     <!-- Finalizacion -->
109     <recorrido id="3">
110       <disparadores:disparadores>
111         <disparadores:condicion id="1" medida="temperatura"
112                                 operador="ge" valor="32"
113                                 unidad="grado centigrado"/>
114       </disparadores:disparadores>
115     </recorrido>
116   </area>
117
118   <ruta id="5" nombre="abandonar estrecho">
119     <waypoint id="1">
120       <pose>
121         <posicion x="600" y="250" z="0" unidad="m"/>
122         <orientacion roll="0" pitch="270" yaw="0" unidad="grado"/>
123         <incertidumbre valor="2" unidad="m"/>
124       </pose>
125     </waypoint>
126   </ruta>
127
128   <area id="6" nombre="muestreo de pH">
129     <vertice id="1" x="600" y="250" z="80" unidad="m"/>
130     <vertice id="2" x="600" y="350" z="80" unidad="m"/>
131     <vertice id="3" x="700" y="350" z="80" unidad="m"/>
132     <vertice id="4" x="700" y="250" z="80" unidad="m"/>
133
```

```
134       <profundidad minima="80" maxima="110" unidad="m"/>
135       <tiempo valor="2" unidad="hora" holgura="5"/>
136
137       <recorrido id="1">
138         <disparadores:disparadores/>
139         <deriva/>
140       </recorrido>
141     </area>
142
143     <ruta id="7" nombre="alcanzar punto de encuentro">
144       <waypoint id="1">
145         <pose>
146           <posicion x="750" y="280" z="100" unidad="m"/>
147           <orientacion roll="0" pitch="330" yaw="10" unidad="grado"/>
148           <incertidumbre valor="4" unidad="m"/>
149         </pose>
150       </waypoint>
151       <waypoint id="2">
152         <pose>
153           <posicion x="800" y="260" z="125" unidad="m"/>
154           <orientacion roll="0" pitch="60" yaw="35" unidad="grado"/>
155           <incertidumbre valor="2" unidad="m"/>
156         </pose>
157         <interpolacion valor="10" unidad="m"/>
158         <velocidad>
159           <posicion x="3" y="3" z="6" unidad="nudo"/>
160         </velocidad>
161       </waypoint>
162       <waypoint id="3">
163         <pose>
164           <posicion x="850" y="220" z="0" unidad="m"/>
165           <orientacion roll="0" pitch="0" yaw="0" unidad="grado"/>
166           <incertidumbre valor="1" unidad="m"/>
167         </pose>
168         <velocidad>
169           <posicion x="0" y="0" z="0" unidad="nudo"/>
170         </velocidad>
171       </waypoint>
172       <transecto id="1" inicio="1" fin="2">
173         <velocidad>
174           <tiempo valor="5" unidad="minuto"/>
175         </velocidad>
176       </transecto>
177       <transecto id="2" inicio="2" fin="3">
178         <velocidad>
179           <posicion x="5" y="2" z="4" unidad="nudo"/>
180         </velocidad>
181       </transecto>
182     </ruta>
183
184     <zonaProhibida id="10" nombre="islote 1">
185       <vertice id="1" x="460" y="225" z="0" unidad="m"/>
186       <vertice id="2" x="470" y="220" z="0" unidad="m"/>
187       <vertice id="3" x="485" y="220" z="0" unidad="m"/>
188       <vertice id="4" x="500" y="230" z="0" unidad="m"/>
189       <vertice id="5" x="540" y="230" z="0" unidad="m"/>
190       <vertice id="6" x="565" y="250" z="0" unidad="m"/>
191       <vertice id="7" x="500" y="270" z="0" unidad="m"/>
192       <vertice id="8" x="480" y="280" z="0" unidad="m"/>
193       <vertice id="9" x="465" y="250" z="0" unidad="m"/>
194     </zonaProhibida>
195     <zonaProhibida id="11" nombre="islote 2">
196       <vertice id="1" x="480" y="450" z="0" unidad="m"/>
197       <vertice id="2" x="520" y="450" z="0" unidad="m"/>
198       <vertice id="3" x="520" y="470" z="0" unidad="m"/>
199       <vertice id="1" x="500" y="465" z="0" unidad="m"/>
200       <vertice id="2" x="465" y="500" z="0" unidad="m"/>
201       <vertice id="3" x="465" y="500" z="0" unidad="m"/>
202     </zonaProhibida>
203
204 </planDeNavegacion>
```

Algoritmo 16.5: PdN (Ejemplo de Exploración)

16.6. Plan de Supervisión

En el PdS se indica una lista de tareas de supervisión para indicar las acciones que deben realizarse bajo ciertas condiciones críticas o excepcionales. En el Algoritmo 16.6 se muestra el fichero XML resultante de la siguiente lista de tareas de supervisión especificadas:

Tarea 1 Salir de un zona prohibida, lo cual se detecta con la excepción **enZonaProhibida**. Para salir se ejecuta el plan **salirDeZonaProhibida**, donde estará especificado cómo debe hacerse esto. Este plan recibe la velocidad a la que se desea salir.

Tarea 2 Liberar memoria cuando se detecta la excepción **agotamientoMemoria**. Se ejecutan comandos de envío de datos, para que así se consiga espacio libre.

Tarea 3 Aumentar la velocidad de crucero cuando el tiempo de la misión supera las 5 horas. Esto se consigue aumentando el valor del parámetro de la misión **VC**.

Tarea 4 Recargar las baterías cuando se detecta la excepción **agotamientoBaterias**. Esto es posible si el vehículo dispone de paneles solares y baterías recargables. En tal caso, se detiene el PdN y se hace emerger el vehículo. El período de inhibición hace que, al menos, durante 20min se estén recargando las baterías.

Tarea 5 Reducir la temperatura interna del vehículo si se detecta una **temperaturaCPU** mayor o igual a 70°C. Se ejecuta el plan **reducirTemperaturaCPU**, al que se le pasa la temperatura que debe alcanzarse.

```
<?xml version="1.0"?>
 <planDeSupervision id="1" nombre="supervision mision"
                    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                    xmlns:disparadores="http://www.disparadores.com"
                    xmlns="http://www.pds.com"
                    xsi:schemaLocation="http://www.pds.com
                                        ../../esquemas/pds.xsd">

  <tarea id="1" nombre="salir de zona prohibida">
    <disparadores:disparadores>
      <disparadores:excepcion id="1" nombre="enZonaProhibida"/>
    </disparadores:disparadores>
    <acciones>
      <ejecutarPlan id="1" plan="salirDeZonaProhibida">
        <!-- velocidad -->
        <parametro id="1" valor="10"/>
      </ejecutarPlan>
    </acciones>
    <periodoInhibicion valor="1" unidad="minuto"/>
  </tarea>

  <tarea id="2" nombre="liberar memoria">
    <disparadores:disparadores>
      <disparadores:excepcion id="1" nombre="agotamientoMemoria"/>
    </disparadores:disparadores>
    <acciones>
      <ejecutarComando id="1" comando="enviarDato">
        <!-- dato -->
        <parametro id="1" valor="conductividad"/>
        <!-- destinatario -->
        <parametro id="2" valor="planificador"/>
      </ejecutarComando>
      <ejecutarComando id="2" comando="enviarDato">
        <!-- dato -->
        <parametro id="1" valor="temperatura"/>
        <!-- destinatario -->
        <parametro id="2" valor="planificador"/>
      </ejecutarComando>
      <ejecutarComando id="3" comando="enviarDato">
        <!-- dato -->
```

```
41          <parametro id="1" valor="profundidad"/>
42          <!-- destinatario -->
43          <parametro id="2" valor="planificador"/>
44        </ejecutarComando>
45        <ejecutarComando id="4" comando="enviarDato">
46          <!-- dato -->
47          <parametro id="1" valor="posicion"/>
48          <!-- destinatario -->
49          <parametro id="2" valor="planificador"/>
50        </ejecutarComando>
51      </acciones>
52      <periodoInhibicion valor="20" unidad="minuto"/>
53    </tarea>
54
55    <tarea id="3" nombre="aumentar velocidad">
56      <disparadores:disparadores>
57        <disparadores:condicion id="1" medida="tiempo"
58                                operador="gt" valor="5" unidad="hora"/>
59      </disparadores:disparadores>
60      <acciones>
61        <ejecutarComando id="1" comando="CambiarVariableMision">
62          <!-- variable -->
63          <parametro id="1" valor="@VC"/>
64          <!-- valor -->
65          <parametro id="2" valor="6"/>
66        </ejecutarComando>
67      </acciones>
68      <periodoInhibicion valor="INF" unidad="minuto"/>
69    </tarea>
70
71    <tarea id="4" nombre="recargar baterias">
72      <disparadores:disparadores>
73        <disparadores:excepcion id="1" nombre="agotamientoBaterias"/>
74      </disparadores:disparadores>
75      <acciones>
76        <ejecutarComando id="1" comando="detenerPlan">
77          <!-- tipo -->
78          <parametro id="1" valor="PdN"/>
79          <!-- plan (id) -->
80          <parametro id="2" valor="1"/>
81        </ejecutarComando>
82        <ejecutarComando id="2" comando="emerger"/>
83      </acciones>
84      <periodoInhibicion valor="20" unidad="minuto"/>
85    </tarea>
86
87    <tarea id="5" nombre="reducir temperatura interna">
88      <disparadores:disparadores>
89        <disparadores:condicion id="1" medida="temperaturaCPU"
90                                operador="ge" valor="70"
91                                unidad="grado centigrado"/>
92      </disparadores:disparadores>
93      <acciones>
94        <ejecutarPlan id="1" plan="reducirTemperaturaCPU">
95          <!-- temperatura -->
96          <parametro id="1" valor="50"/>
97        </ejecutarPlan>
98      </acciones>
99      <periodoInhibicion valor="30" unidad="segundo"/>
100   </tarea>
101 </planDeSupervision>
```

Algoritmo 16.6: PdS (Ejemplo de Exploración)

16.7. Plan de *Log*. Registro del Sistema

En el Algoritmo 16.7 se muestra la especificación del PdL usando la sintaxis de *log4cxx* [Gülkü, 2002]. Esta especificación muestra detalles del sistema *Sick***AUV**, que ejecutará la misión. Se indica que se registre información de determinados componentes y la forma en que debe registrarse.

```
1 <?xml version="1.0" encoding="UTF-8" ?>
2 <!DOCTYPE log4j:configuration SYSTEM "log4j.dtd">
3
```

```
<log4j:configuration xmlns:log4j='http://jakarta.apache.org/log4j/'>
  <appender name="stdout" class="org.apache.log4j.ConsoleAppender">
    <layout class="org.apache.log4j.PatternLayout">
      <param name="ConversionPattern" value="%-4r [%t] %-5p %c - %m%n"/>
    </layout>
  </appender>
  <appender name="ficheroHTMLInterpretePlan"
            class="org.apache.log4j.FileAppender">
    <param name="File" value="data/log/interpreteplan.html"/>
    <param name="Append" value="false"/>
    <layout class="org.apache.log4j.HtmlLayout">
      <param name="Title" value="Log del Interprete de Planes"/>
    </layout>
  </appender>
  <appender name="ficheroHTMLGestorDisparadores"
            class="org.apache.log4j.FileAppender">
    <param name="File" value="data/log/gestordisparadores.html"/>
    <param name="Append" value="false"/>
    <layout class="org.apache.log4j.HtmlLayout">
      <param name="Title" value="Log del Gestor de Disparadores"/>
    </layout>
  </appender>
  <appender name="ficheroHTMLComunicador"
            class="org.apache.log4j.FileAppender">
    <param name="File" value="data/log/comunicador.html"/>
    <param name="Append" value="false"/>
    <layout class="org.apache.log4j.HtmlLayout">
      <param name="Title" value="Log del Comunicador"/>
    </layout>
  </appender>
  <appender name="ficheroHTMLSubsistemaSensorial"
            class="org.apache.log4j.FileAppender">
    <param name="File" value="data/log/subsistemasensorial.html"/>
    <param name="Append" value="false"/>
    <layout class="org.apache.log4j.HtmlLayout">
      <param name="Title" value="Log del Subsistema de Sensorial"/>
    </layout>
  </appender>

  <logger name="InterpretePlan" additivity="false">
    <level value="warn"/>
    <appender-ref ref="stdout"/>
    <appender-ref ref="ficheroHTMLInterpretePlan"/>
  </logger>
  <logger name="GestorDisparadores" additivity="false">
    <level value="debug"/>
    <appender-ref ref="stdout"/>
    <appender-ref ref="ficheroHTMLGestorDisparadores"/>
  </logger>
  <logger name="Comunicador" additivity="false">
    <level value="info"/>
    <appender-ref ref="stdout"/>
    <appender-ref ref="ficheroHTMLComunicador"/>
    <appender-ref ref="ficheroHTMLSubsistemaComunicacion"/>
  </logger>
  <logger name="SubsistemaSensorial" additivity="false">
    <level value="debug"/>
    <appender-ref ref="stdout"/>
    <appender-ref ref="ficheroHTMLSubsistemaSensorial"/>
  </logger>

  <root>
    <level value ="debug"/>
    <appender-ref ref="stdout"/>
  </root>
</log4j:configuration>
```

Algoritmo 16.7: PdL (Registro del Sistema)

16.8. Parámetros de la misión

La misión se acompaña de una tabla de parámetros, que se muestra en el Algoritmo 16.8 y cuyos parámetros son:

VC Indica la velocidad de crucero del vehículo. Modifica la componente x de la velocidad que debe mantener en el transecto 1 de la ruta 1 y el transecto 2 de la ruta 7. Tiene el valor 4 por defecto, para el que no se requiere la indicación de unidades, pues ya aparece en la especificación de la velocidad en el transecto. En la Figura 16.1 se muestra como este parámetro se usa en los transectos antes indicados. También se muestra que se usa como variable en el *waypoint* **R3W2**.

```
<?xml version="1.0"?>
<tablaParametros id="1" nombre="Tabla Parametros 1"
                 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                 xmlns:parametros="http://www.parametros.com"
                 xmlns="http://www.tablaparametros.com"
                 xsi:schemaLocation="http://www.tablaparametros.com
                                     ../esquemas/tablaparametros.xsd">

  <parametro nombre="VC" valor="4">
    <parametros:elemento nombre="/planDeNavegacion[@id='1']/ruta[@id='1']
                                 /transecto[@id='1']/velocidad/posicion/@x"/>
    <parametros:elemento nombre="/planDeNavegacion[@id='1']/ruta[@id='7']
                                 /transecto[@id='2']/velocidad/posicion/@x"/>
  </parametro>

</tablaParametros>
```

Algoritmo 16.8: PdA (Ejemplo de Exploración)

Capítulo 17

Conclusiones

La vida es el arte de sacar conclusiones suficientes a partir de datos insuficientes

— Samuel Butler
1612–1680 (Poeta inglés)

Con este capítulo se concluye el trabajo presentado, resumiendo los contenidos tratados a los largo del documento de acuerdo a la consecución de los objetivos enumerados en la Sección 1.1. Esto nos permite reformular el objetivo principal del proyecto afirmando que:

> Se ha realizado el diseño de un sistema integrado de control para un AUV, denominado *Sick***AUV**, que incluye mecanismos para la definición del equipamiento disponible a bordo del vehículo y de la misión a desarrollar.

Seguidamente se enumeran las contribuciones que pueden extraerse de las propuestas y resultados obtenidos. Los aspectos que no han podido ser cumplimentados, así como otros posibles trabajos de investigación futura, se comentan en el Capítulo 18.

En la introducción se ilustró la estructura del documento mediante la Figura 1.3, la cual abarca los tres principales elementos de un AUV y que constituyen, a su vez, los objetivos de este proyecto: **equipamiento**, **misión** y **sistema**. Para cada uno de estos elementos se ha elaborado un estudio y se ha propuesto un diseño que se ve reflejado en la Figura 17.1, la cual rellena el espacio que nos planteábamos cubrir en este proyecto.

De forma más detallada, se resume el contenido de este documento para resaltar el tratamiento realizado para cada uno de los siguientes objetivos:

Equipamiento A lo largo de la Parte I se ha realizado un estudio de los vehículos de exploración submarina y los dispositivos e información que éstos suelen manejar durante las misiones que realizan. Este estudio se centra en los AUVs, de los cuales se definen sus características y posibles parámetros de configuración, tanto del vehículo como del sistema embebido del que disponen. Se ha propuesto un formato

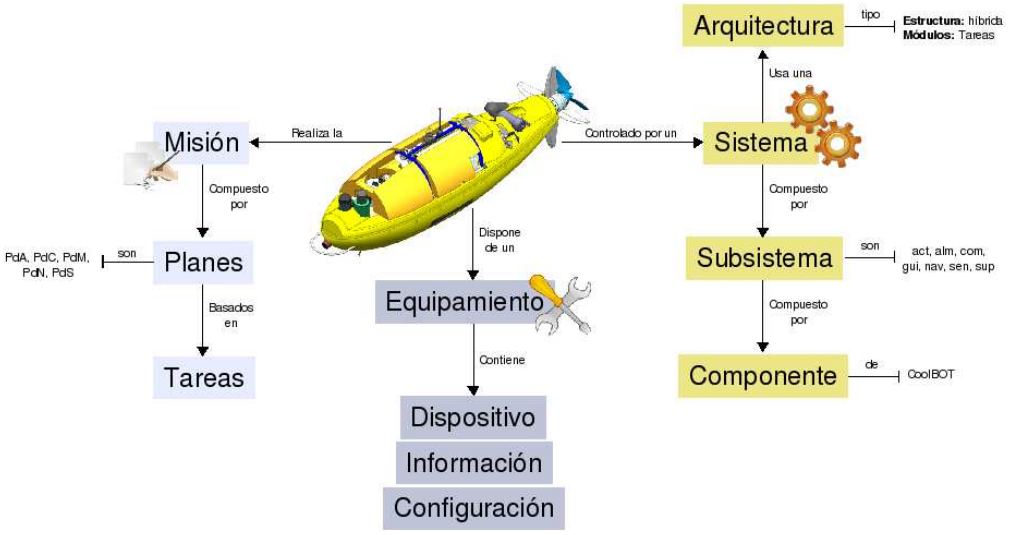


Figura 17.1: Mapa conceptual de la relación del AUV con el Equipamiento, la Misión y el Sistema propuestos. Detalles de la estructura de los mismos, para los diseños propuestos

de especificación del equipamiento que cubre los dispositivos, la información y la configuración del vehículo y sistema, usando XML como lenguaje. Se han organizado los diferentes elementos que forman parte de esta especificación y se han elaborado esquemas XML (XSD) para definir formalmente la sintaxis de la especificación. A modo de ejemplo se ha elaborado una especificación completa de un hipotético AUV.

Misión En la Parte II se ha analizado la tipología de misiones que acostumbran a realizar los vehículos de exploración oceanográfica, con un enfoque especial en los AUVs. A continuación se han estudiado diferentes arquitecturas de especificación de misiones y se han propuesto los **planes de la misión** como una arquitectura basada en tareas y compuesta por planes. Las principales virtudes de esta arquitectura son la modularidad, flexibilidad, facilidad de definición, portabilidad y reconfiguración (véase el Cuadro 12.4). La especificación de la misión que se ha propuesto dispondrá de parámetros que facilitarán la reconfiguración dinámica de la misma. Se usa XML como lenguaje de especificación y se han elaborado esquemas XML (XSD) para definir formalmente la sintaxis. Se ha definido una organización de los planes y las tablas de parámetros, y se han realizado algunos ejemplos de misiones representativas (véase el Capítulo 16).

Sistema En la Parte III se ha realizado un estudio de las arquitecturas usadas en los sistemas de diversos AUVs. Se ha analizado tanto la estructura como los módulos de la arquitectura y se ha propuesto *Sick***AUV** como sistema. *Sick***AUV** dispone de una arquitectura híbrida (deliberativa y reactiva) cuyos módulos se basan en tareas. El sistema se ha dividido en varios subsistemas, que a su vez tienen una división en componentes internos. Estos elementos se han diseñado usando componentes CoolBOT, tanto para los componentes internos, como para los propios subsistemas, que serán componentes compuestos. Esto ha dotado al sistema de una

jerarquización y modularidad importantes, que facilitan su diseño y distribuyen la carga computacional apropiadamente. Determinados componentes son los encargados de la ejecución de las acciones de las tareas, mediante la creación de un **pedido**, que permite monitorizar la evolución o estado de ejecución de la acción.
La comunicación entre los subsistemas y componentes del sistema está resuelta por los mecanismos de comunicación que ofrece CoolBOT. No obstante, *Sick***AUV** también incorpora herramientas que facilitan tanto la comunicación interna del sistema, como la externa: manejo de *sockets* como *streams* de C++, serialización mediante XDR, factoría de clases donde se dispone de la interfaz **Empaquetable** para homogeneizar el tratamiento de la información, sistema de mensajes de solicitud de servicios y notificación de informes del estado de realización de pedidos, etc.
Con el estado actual de desarrollo de *Sick***AUV**, el sistema no dispone de soporte para manejar la especificación del equipamiento del vehículo, pero sí es capaz de manejar la especificación de la misión y ejecutar misiones básicas de determinados planes. Para ello dispone de un **intérprete** y un **gestor de disparadores** que aprovechan todos los subsistemas que manejan planes de la misión. Respecto al PdL o **registro** del sistema, se ha utilizado la librería **log4cxx**. Esto permite monitorizar fácilmente el estado de todo el sistema controlando el nivel de detalle deseado.

17.1. Contribuciones

Este proyecto ha logrado el objetivo principal propuesto mediante el diseño del sistema *Sick***AUV** y la elaboración de un formato de especificación del equipamiento y de la misión. A lo largo del desarrollo realizado para alcanzar este objetivo, se han conseguido diversas contribuciones de interés, que se enumeran y detallan brevemente a continuación:

Especificación del Equipamiento Se ha realizado un estudio y análisis de los dispositivos y la información con la que suelen equiparse los vehículos de exploración submarina. Estos tipos de equipamiento se denominan físico y lógico, respectivamente, y permiten la elaboración de una clasificación que, a su vez, diferencia tres categorías según su finalidad:

1. Dispositivos de apoyo a la navegación, tanto la instrumentación como los elementos que forman el sistema de impulsión físico.
2. Carga útil de la misión, que suele estar formada por sensores que muestrean parámetros físico-químicos de los océanos.
3. Soporte de dispositivos internos para la monitorización del estado del sistema y el vehículo.

En base a todo el equipamiento que puede emplearse y la configuración del mismo y el propio sistema del vehículo, se ha diseñado un formato de especificación en el Capítulo 9. Se ha establecido la organización del mismo y se ha implementado usando XML, de acuerdo a una serie de esquemas XML (XSD) que definen la sintaxis del lenguaje de especificación resultante.

Planes de la Misión El estudio de las arquitecturas de especificación de misiones para vehículos de exploración oceanográfica muestra tres casos fundamentales, donde

se usan **redes de Petri**, **tareas** o **comportamientos** como elementos básicos de definición de la misión. En base a este análisis se ha realizado una propuesta basada en **tareas**, pero que además incluye una división lógica mediante planes (véase la Definición 12.5).
La arquitectura de los **planes de la misión** permite una definición modular de la misión donde cada plan abarca una tipología de tareas bien diferenciada. La arquitectura de especificación de misiones propuesta consta de los siguientes planes:

PdA Plan de Almacenamiento, encargado de registrar de forma permanente los datos de la misión que sean de interés.

PdC Plan de Comunicación, donde se indica la información que el sistema debe enviar o recibir en un momento dado o bajo determinadas condiciones.

PdM Plan de Medición, que recoge la lista de medidas que deben muestrearse, así como la configuración del proceso de muestreo de cada una.

PdN Plan de Navegación, que contiene la secuencia de tareas de navegación *ad hoc*, como son el seguimiento de rutas, exploración de áreas y seguimiento de medidas, así como la definición de zonas prohibidas.

PdS Plan de Supervisión, donde se definen las tareas de supervisión que deben llevarse a cabo durante el transcurso de la misión, así como las acciones que deben efectuarse bajo determinadas condiciones excepcionales —v. g. agotamiento de baterías.

Parámetros de la Misión Para permitir la posibilidad de reconfiguración dinámica de la misión se han propuesto los **parámetros de la misión**. Se trata de un mecanismo de asociación entre los nombres de los parámetros y elementos concretos de la especificación de la misión. El sistema de reconfiguración implantado en la especificación de la misión también permite la modificación directa de cualquier elemento de ésta, gracias a las capacidades del lenguaje de consulta sobre ficheros XML conocido como XPath, que permite referenciar los atributos o elementos XML que desean modificarse.

Especificación de la Misión Se ha diseñado un formato de especificación de la misión para la arquitectura de los **planes de la misión**, que incluye el soporte para los **parámetros de la misión**. Se ha establecido la organización de los diferentes tipos de planes y las tablas de parámetros de la misión. Todo esto se ha implementado usando XML, de acuerdo a una serie de esquemas XML (XSD) que definen la sintaxis del lenguaje de especificación resultante. También se han empleado sentencias XPath para la asociación de los nombres de los parámetros con los atributos o elementos XML a los que referencian.

Sistema *Sick*AUV Se ha propuesto *Sick***AUV** como un sistema distribuido, pero no necesariamente paralelo, que emplea una arquitectura híbrida (deliberativa y reactiva) con módulos de ejecución de tareas modelados mediante componentes CoolBOT. Internamente está formado por varios subsistemas, que a su vez contienen componentes. Dentro del desarrollo del sistema podemos identificar las siguientes contribuciones:

Subsistemas y Componentes Tanto los subsistemas como los componentes de *Sick***AUV** están modelados e implementados mediante componentes CoolBOT. Se han propuesto los siguientes subsistemas:

Subsistema Actuador Controla los actuadores con los que vaya equipado el vehículo. Esto incluye el sistema de impulsión. Recibirá comandos de control para actuar sobre los dispositivos y también puede recibir la configuración que se desea que éstos usen. Por norma general, simplemente toma los comandos de actuación del subsistema de navegación, lo que permite comandar los motores y las superficies de control para conseguir que el vehículo alcance la posición deseada. Por tanto, este subsistema constituye la capa reactiva de la arquitectura de control del subsistema.

Subsistema de Almacenamiento Gestiona el **inventario** de datos almacenados en el sistema y se encarga de almacenar las muestras de datos sensoriales o de otro tipo. Almacenará aquellas medidas que vengan indicadas en el PdA, de acuerdo a las condiciones especificadas. Dispone de los componentes de interpretación y gestión de disparadores para poder ejecutar correctamente el plan.

Subsistema de Comunicación Se encarga de las comunicaciones con el exterior. Para ello dispone de un **comunicador** que se encarga de enviar y recibir datos en forma de muestras o ficheros, de acuerdo con las especificaciones del PdC. Dispone de los componentes de interpretación y gestión de disparadores para poder ejecutar correctamente el plan. Además, mediante un componente de **acceso remoto** permite las comunicaciones externas cuando el vehículo está en línea. Este módulo ofrece la posibilidad de controlar remotamente el vehículo, como si se tratara de un ROV, pero también ofrece la posibilidad de reconfigurar el sistema o modificar la misión dinámicamente.

Subsistema de Guiado A partir de la especificación del PdN, indica la pose y velocidad que debe mantener el vehículo, para navegar de acuerdo a las tareas de navegación definidas. Dispone de los componentes de interpretación y gestión de disparadores para poder ejecutar correctamente el plan. Enviará comandos al subsistema de navegación para indicar cómo debe desplazarse el vehículo, de modo que se trata de la capa deliberativa de la arquitectura de contol del sistema de navegación. Por tanto, determinará el camino a seguir según las restricciones o condiciones especificadas — v. g. paso por los *waypoints* de una ruta, camino a seguir en la exploración de un área, indicación de la dirección a llevar durante el seguimiento de una medida, etc.

Subsistema de Navegación Recibe los comandos del subsistema de guiado, que indican la pose y velocidad que el vehículo debe mantener en cada instante, para navegar según lo indicado en la misión. Este subsistema será el *piloto automático* del vehículo, que resolverá las ecuaciones hidrodinámicas y aplicará algoritmos de control en base a la información de los instrumentos de navegación, para determinar cómo comandar los actuadores del sistema de impulsión. Estos comandos resultantes se enviarán al subsistema actuador para que sea éste el que finalmente efectúe las

acciones. Con este subsistema se completan las tres capas que constituyen la arquitectura de control para la navegación en el sistema *Sick***AUV**.

Subsistema Sensorial Gestiona el equipamiento sensorial y se encarga de medir de acuerdo a la configuración y condiciones especificadas en el PdM —v. g. frecuencia de muestreo, resolución, etc. Dispone de los componentes de interpretación y gestión de disparadores para poder ejecutar correctamente el plan. Con los valores obtenidos de las medidas se construye una muestra con un formato autodefinido que se envía al resto de subsistemas, para que realicen con ella las tareas oportunas.

Subsistema de Supervisión Aunque cada subsistema realizará labores de supervisión sobre sus componentes internos, el sistema dispondrá de un módulo de supervisión específico. Este subsistema usará las especificaciones del PdS, para ejecutar las acciones apropiadas bajo determinadas condiciones excepcionales. Dispone de los componentes de interpretación y gestión de disparadores para poder ejecutar correctamente el plan. Las posibles acciones que pueden acometer engloban todo el universo de comandos soportados por el sistema y, además, se recoge la posibilidad de ejecutar **planes de contingencia**.

Aunque se han diseñado todos estos subsistemas, sólo se dispone de una implementación funcional del subsistema de almacenamiento y sensorial, si bien es bastante básica. El diseño del proceso de interpretación de la misión mediante la gestión de disparadores ha sido implementado y probado para la ejecución del PdM, como se ilustra en el **??**. También se dispone de la implementación del sistema de comunicación de mensajes de control entre los diferentes subsistemas y componentes, lo cual incluye los modelos de datos y algoritmos que facilitan comunicación interna y externa.

HAL Capa de Abstracción Hardware, o *Hardware Abstraction Layer*, la cual desacopla los modelos físicos de los dispositivos manejados por el sistema de la interfaz de control de los mismos. Esto permite al sistema establecer una clasificación de los dispositivos y facilitar la integración de nuevos modelos, si ya está soportado por la HAL.
El diseño de *Sick***AUV** recoge dos alternativas concretas para dar soporte a los dispositivos del equipamiento del vehículo:

Player El *framework* para sistemas robóticos móviles *Player* ofrece soporte para un gran número de dispositivos, tanto sensoriales como actuadores. Por este motivo resulta una herramienta útil para integrar los dispositivos físicos disponibles. Además, permite el desarrollo de nuevos dispositivos y su integración de forma transparente dentro de toda la arquitectura.
Esto permite la fácil integración de dispositivos físicos y también es posible crear sensores virtuales que ofrezcan una determinada información a través de la interfaz de dicho sensor —v. g. la batimetría ofrece la profundidad para una determinada latitud y longitud, de modo que puede emplearse como si se tratase de un sensor de profundidad (profundímetro) ubicado en el fondo del mar.

Dispositivos Virtualizados del Simulador La herramienta de simulación de AUVs ofrece la posibilidad de usar dispositivos virtuales. Los disposi-

tivos se manejarán mediante la comunicación por TCP/IP, a través de la cual fluirán los datos con un formato específico diseñado pero no implementado en el sistema (véase la Sección 15.7). Esto ofrece la posibilidad de desarrollar simulaciones *Hardware In Loop* (HIL) donde se virtualiza el equipamiento sensorial no disponible en el vehículo.

En ambos casos, se realizará una adaptación de las interfaces de los dispositivos proporcionados, para garantizar la homogeneidad en el tratamiento de *Sick***AUV**. Esto no sólo incluye los mecanismos de comando, sino también el formato de representación de los datos, v. g. para las muestras se hará uso de un modelo autodefinido y universal.

Dispositivos adaptados a *Player* Varios dispositivos de instrumentación han sido integrados en *Player*. Para ello se ha adoptado la arquitectura de este *framework* y se han definido los *drivers* y empleado las interfaces más apropiadas, dentro del repertorio disponible en *Player*. El desarrollo de los *drivers* se ha hecho mediante la adaptación de software existente para varios dispositivos físicos concretos, como son una brújula con inclinómetros y magnetómetro, un giróscopo y un GPS (véase el Capítulo 8). También se han hecho pruebas con otro tipo de dispositivos, tales como cámaras web, información batimétrica disponible en ficheros *netCDF*, sintetizadores de voz (*festival*), etc.
El uso de *Player* en el sistema desarrollado se centra en facilitar la integración de dispositivos, mediante la definición de sus interfaces y la disponibilidad de una infraestructura de comunicación por *sockets* y serialización con XDR, lo cual permite la abstracción del hardware.

Manejo de ficheros *netCDF* La información batimétrica, así como otro tipo de información de gran volumen —v. g. datos meteorológicos, corrientes marinas, etc.—, suelen estar contenidas en ficheros binarios con un formato especialmente diseñado para la recuperación eficiente de los datos en él almacenados. Uno de los formatos más comunes es *netCDF*. Se ha partido una librería de manejo de este tipo de ficheros, desarrollada en C++, para preparar una interfaz de recuperación de datos adecuada a las necesidades del sistema. De hecho, se ha modelado como un sensor virtual que se integra dentro de *Player*.

Generador de Componentes CoolBOT Aunque no aparece entre los objetivos iniciales del proyecto, dado que se ha usado el *framework* CoolBOT como base para el diseño e implementación del sistema *Sick***AUV**, se ha desarrollado un generador de componentes CoolBOT con el fin de facilitar la construcción de los esqueletos de declaración y definición de los mismos. Este generador construye los componentes CoolBOT a partir de una especificación de las características de los mismos indicada en XML. Mediante un *script* desarrollado en Perl, se obtiene como resultado un esqueleto para C++, compuesto por un fichero de cabecera y otro de implementación, listos para rellenar con la lógica de control propia del componente dentro del sistema *Sick***AUV**. En la Sección E.1.3 se comentan algunos detalles de este desarrollo, así como un generador alternativo que no estaba disponible al inicio de este proyecto.

A la luz del trabajo realizado queda patente que no sólo se ha alcanzado el objetivo del proyecto, sino que además se dispone de una implementación parcial del sistema y varias

contribuciones adicionales. Además, se dispone de numerosas contribuciones en forma de estudios, diseños e implementaciones, que constituyen un punto de partida para nuevos desarrollos. De hecho, en varios casos sólo ha faltado realizar la integración en el sistema, como es el caso de la HAL para *Player*, los dispositivos adaptados a *Player*, el manejo de ficheros *netCDF*, etc., que quedan como propuestas de trabajo futuro.

Mención especial merece el caso del generador de esqueletos de componentes CoolBOT realizado en el transcurso del proyecto para su aprovechamiento en la implementación de *Sick***AUV**. Se trata de un software completamente independiente de *Sick***AUV**, que puede reutilizarse en otros desarrollos software (véase la Sección E.1.3 para más información).

La implementación realizada del sistema abarca su núcleo, lo que incluye la interpretación de los planes de la misión y la gestión de las tareas, especialmente sus disparadores. Esto constituye la base para el desarrollo de los subsistemas, los cuales están implementados parcialmente o como *cajas vacías*. No obstante, esto es suficiente para demostrar la viabilidad del diseño en líneas generales, tal y como ilustra el **??** con los resultados de la interpretación de un PdM simple, mediante el subsistema sensorial.

Capítulo 18

Trabajo Futuro

Con todo software sucede que en el momento en que lo dominas completamente,
aparece una nueva versión.

— Bahaman
2006 (Ley de Bahaman)

Como resultado del desarrollo de este proyecto se abre un abanico de mejoras, extensiones y áreas de investigación nuevas. También se han encontrado numerosos problemas que pueden ser objeto de un estudio más detallado. En este sentido, se muestra a continuación, una lista de los principales proyectos de continuación o complementarios que resultaría interesante abordar como trabajo futuro.

Soporte para el Equipamiento Entre las contribuciones de este proyecto se incluye la **especificación del equipamiento**. Sin embargo, sólo cubre el análisis y diseño, de modo que se plantea como trabajo futuro la implementación del analizador y el desarrollo del modelo de datos para mantener la información del equipamiento en el sistema.

Respecto a la implementación del analizador, dado que la especificación está realizada en XML y se dispone de los esquemas XML (XSD), el proceso de desarrollo se simplifica enormemente. La validación XML permite determinar que la sintaxis de especificación del equipamiento es correcta y para la implementación del analizador se pueden usar herramientas como las comentadas en la Sección E.7. En concreto se propone el uso de SAX2 (*Simple API for XML* 2), que es la técnica empleada para el proceso de análisis de la misión.

Implementación del núcleo de *Sick*AUV Continuar y terminar la implementación del núcleo de *Sick*AUV. Esto incluye principalmente los mecanismos de comunicación, la HAL, el soporte para manejar la configuración del sistema, la integración del diseño de los parámetros de la misión y la posibilidad de reconfiguración de los

atributos y elementos XML de los planes de la misión, la integración de las excepciones, aparte de otras tareas específicas de determinados subsistemas o módulos.

Navegación Aunque el enfoque del diseño del sistema cubre los AUVs, la implementación no considera la navegación. Por este motivo, será necesario desarrollar los subsistemas de guiado, navegación y actuador, así como la integración de información sensorial de los instrumentos de navegación, el soporte de la hidrodinámica, los algoritmos de control y el diseño del *piloto automático* y demás mecanismos necesarios para la navegación. Este trabajo requiere la adopción de una arquitectura de control apropiada y no es en absoluto trivial.

Supervisión Los mecanismos de supervisión, tanto a nivel de sistema como de subsistemas, no están implementados en *Sick***AUV**. Se plantea la implementación de los mismos, junto con el control y recuperación frente a excepciones, la gestión y ejecución de **planes de contingencia**, así como cualquier otra medida que resulte útil para monitorizar y supervisar el estado del sistema.

Implementación de Servicios La mayoría de los servicios que ofrecen los subsistemas y los componentes de éstos, sólo han sido diseñados, de modo que una tarea pendiente consiste en la implementación de los mismos. En el caso de los servicios para las acciones del PdM sí se dispone de la implementación, i. e. la acción **medir**. Pero esta implementación sólo es parcial y por ello también se considera parte de esta tarea la finalización de las implementaciones parciales de servicios.

Integración de Dispositivos Una de las contribuciones más portables del proyecto es la adaptación de dispositivos a *Player*. La implementación actual de *Sick***AUV** requiere la integración de *Player* para aprovechar los dispositivos que han sido adaptados a *Player* como parte del desarrollo de este proyecto. Esta tarea debe coordinarse con el desarrollo de la Capa de Abstracción Hardware (HAL) y queda pendiente como trabajo futuro.

Planificación y Monitorización de Misiones El sistema del AUV puede complementarse con el desarrollo de una aplicación gráfica para la planificación y monitorización de misiones (véase la Figura 18.1 (a)). En realidad, esta tarea está siendo llevada a cabo por el Proyecto Fin de Carrera (PFC) complementario denominado **planificador** en el Apéndice A.

Simulación El sistema del AUV puede complementarse con el desarrollo de un simulador del entorno submarino, con modelos matemáticos de parámetros físico-químicos, capaz de soportar AUVs virtuales con su equipamiento virtualizado (véase la Figura 18.1 (b)). Este tipo de herramienta permitirá la ejecución simulada de las misiones, empleando el mismo formato de especificación de la misión y el equipamiento que con los AUVs reales y el sistema *Sick***AUV**. Esta tarea está siendo llevada a cabo por el PFC complementario denominado **simulador** en el Apéndice A.

Prototipo de AUV La elaboración del prototipo de un AUV para facilitar la realización de pruebas reales del sistema es también una posibilidad de continuación de este proyecto. Hay que apuntar que durante el desarrollo del proyecto se planteó esta posibilidad y actualmente se dispone de un prototipo cuya estructura está

completamente diseñada y construida. No obstante, aún se está trabajando en la equipación del sistema de potencia y hay diversas tareas que deben realizarse antes de disponer de un prototipo completamente funcional y listo donde instalar el sistema.

Comunicación con el planificador En *Sick***AUV** ya se dispone de una implementación inicial de los mecanismos de comunicación con la herramienta de planificación de misiones, pero se plantea su finalización y la realización de pruebas como trabajo futuro. Esto incluye el proceso de envío de la misión, comentado en la Sección 12.3, la comunicación de datos en forma de muestras o ficheros, tanto el envío como la recepción, y el **acceso remoto** al sistema.

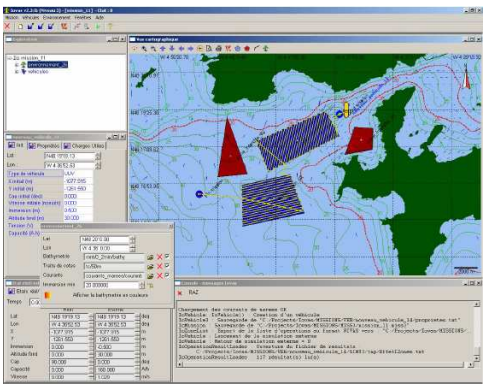
Batería de Pruebas Cuando se disponga de un sistema funcional, con una implementación que cubra prácticamente todos los subsistemas, se podrán efectuar diversas pruebas. En este sentido, se antoja interesante la elaboración de un estudio de las pruebas y mecanismos de chequeo más apropiados, tomando como base el análisis de la Sección 11.6. Como resultado podría obtenerse una batería de pruebas que verifiquen el correcto funcionamiento del sistema.

Simulación HIL Aprovechando la HAL del sistema y el **simulador**, junto con su capacidad para ofrecer dispositivos virtualizados, se pueden realizar simulaciones *Hardware In Loop* (HIL). Este tipo de simulaciones son de gran utilidad porque permiten probar el vehículo y su hardware en un entorno simulado y controlado. Además, se pueden plantear variantes de simulación en las que se haga el estudio del sistema aún sin disponer de todo el equipamiento físico real, incluyendo dispositivos virtualizados. Además, se puede realizar el estudio de determinados componentes, mientras otros se simulan.

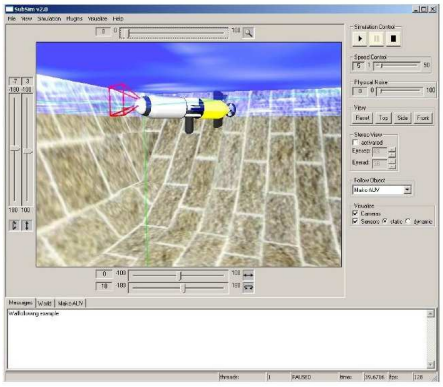
Coordinación Como proyecto de investigación a largo plazo se propone el desarrollo de mecanismos de coordinación que se integren en el sistema y permitan la realización de misiones multi-AUV o entre múltiples vehículos o sistemas de exploración oceanográfica, tal y como se comenta en la Sección 2.3. Evidentemente, para que esto sea posible es necesario disponer de varios vehículos de estas características completamente operativos y con los dispositivos de comunicación adecuados, i. e. con la tecnología subyacente necesaria para permitir la comunicación.

Generación de Componentes CoolBOT Debido al hecho de que se usa CoolBOT para el diseño e implementación del sistema *Sick***AUV**, resulta interesante avanzar en la realización de herramientas para la plataforma CoolBOT. Por tanto, se plantea la mejora de las herramientas de generación de esqueletos de componentes CoolBOT. Para ello puede partirse del generador desarrollado como parte de este proyecto y mejorarlo (véase la Sección E.1.3), o bien adaptar el PFC titulado **Compilador/Generador de esqueletos C++ para componentes CoolBOT**, realizado específicamente con esta finalidad y que se comenta en el Apéndice A [Santana Jorge, 2007].

Aparte de los proyectos antes mencionados, hay una serie de aspectos de menor importancia a los que se puede dar solución como continuación de este proyecto, orientados a la consecución de una versión completamente operativa del sistema *Sick***AUV**.



(a) Planificador IOVAS [Ayreault et al., 2006]. Interfaz de preparación de la misión



(b) Simulador SubSim [Bräunl et al., 2006]

Figura 18.1: Ejemplos de herramientas de planificación y simulación de misiones para AUVs

Multiplicidad de planes Se plantea la posibilidad de que una misión disponga de varios planes de un mismo tipo. Esto es fácilmente implementable y daría mayor flexibilidad al proceso de construcción de misiones, permitiendo un mejor aprovechamiento de planes de otras misiones. No obstante, durante el proceso de validación de la misión será necesario comprobar que estos planes no sean incompatibles entre sí (véase la Sección 12.3.1).

Comandos nuevos Aparte de implementar los comandos propuestos en el Capítulo 15, pero aún no soportados por *Sick***AUV**, se plantea la posibilidad de definir nuevos comandos o aumentar el número de parámetros de los existentes. El sistema está diseñado de forma que esta labor resulte simple y sistemática. Además la integración de los comandos no genera problemas ni la necesidad de modificaciones en la arquitectura del sistema. Se trata, por tanto, de un sistema flexible preparado para el desarrollo e incorporación de comandos adicionales.

Análisis Dimensional En los sistemas de muestreo de parámetros físico-químicos, como es el caso de los vehículos de exploración ocenográfica, resulta interesante representar las muestras con su valor y unidades. El Análisis Dimensional es una poderosa herramienta que permite simplificar el estudio de cualquier fenómeno en el que estén involucradas muchas magnitudes físicas en forma de variables independientes. Esto permite aplicar formalismos matemáticos como el teorema de Vaschy-Buckingham (teorema Π) para verificar que los cálculos que se realizan son correctos [Muñoz Andrés, 1991]. Además, disponiendo de un sistema donde los valores van acompañados de las unidades se evitan errores típicos debidos al uso de unidades incorrectas. *Sick***AUV** ya dispone de un modelo de datos que almacena la unidad de las muestras, pero se plantea como trabajo futuro la implementación de un sistema de conversión de unidades y una aritmética dimensional, que permita operar muestras con unidades de forma transparente.

Soporte para la Coordinación Con vistas a la coordinación entre vehículos o sistemas, se plantea ofrecer el soporte de comunicaciones necesario para la misma

mediante el diseño e implementación de nuevos comandos. Estos comandos podrían usarse y probarse a través del PdC durante el desarrollo de esta extensión del sistema.

Disparadores: Operadores relacionales El diseño e implementación de *Sick***AUV** soporta operadores relacionales para los disparadores de tipo **condición**. La lista de operadores relacionales es la habitual: $\neq$, $<$, $\leq$, $=$, $\geq$ y $>$. Sin embargo, a la hora de especificar condiciones en las tareas de la misión, es común el uso de construcciones semánticas más potentes o con cierta incertidumbre, v. g. aproximadamente igual, mucho mayor que, etc. Por este motivo se plantea la incorporación de un conjunto de operadores relacionales extendidos, que incluyen comparaciones con una aproximación o margen de error y comparaciones relativas —i. e. dependientes del valor. Como punto de partida se propone la siguiente lista: $\lesssim$, $\approx$, $\gtrsim$, $\lll$, $\ll$, $\gg$, $\ggg$. El sistema usará un valor de precisión para las comparaciones con margen de error y un factor de escala para las comparaciones relativas; ambos serán parámetros configurables del sistema.

Mnemotécnicos de puertos El formato de especificación de la dirección que se indica en las acciones de comunicación del PdC se comenta en la Sección E.13.1. Se admiten nombres de dominio como alternativa a las direcciones IP. Sin embargo, sólo se admiten valores numérios para los puertos. Se propone, por tanto, permitir el uso de nombres o mnemotécnicos de puertos, v. g. **ftp** para el puerto 21, **http** para el 80. Se trata de una asociación entre los nombres de los puertos estándar y sus valores numéricos, lo cual sería bastante sencillo de implementar e integrar en el sistema.

Dispositivos de diferente HAL Se plantea la posibilidad de que coexistan dispositivos implementados por HALs diferentes. Esto permitiría que el sistema pueda usar dispositivos físicos reales y virtualizados simultáneamente, v. g. uso de un giróscopo real a través de la HAL de *Player* junto con un termómetro virtualizado a través de la HAL del simulador.

Bibliografía

[Alexandrescu, 2001] Alexandrescu, A. (2001).
Modern C++ Design: Generic Programming and Design Patterns Applied.
Addison Wesley.
ISBN: 0-201-70431-5
Pages: 352.
[citado en la pág. 29]

[Ataz López, 2006] Ataz López, J. (2006).
Guía casi completa de BibTEX.
1 edition.
[citado en la pág. 27]

[Ayreault et al., 2006] Ayreault, H., Dabe, F., Barbier, M., Nicolas, S., and Kermet, G. (2006).
Goal driven planning and adaptivity for auvs.
First Workshop on Control Architectures of Robots, pages 1–8.
GESMA, Groupe d'Etudes Sous-Marines de l'Atlantique. BP42 — 29240 Brest Armées — France
ONERA, Office National d'Etudes et de Recherches Aérospatiales, Toulouse Center. BP 4025 — 31055 Toulouse cedex 4 — France
PROLEXIA, 865 avenue de Bruxelles — 83500 La Seyne-sur-mer — France.
[citado en la pág. 264]

[Balch and Arkin, 1989] Balch, T. and Arkin, R. (1989).
Motor schema-based formation control for multiagent robot teams.
First International Conference on Multiagent Systems, San Francisco, pages 1–7.
[citado en la pág. 166]

[Barbier et al., 2001] Barbier, M., Lemaire, J., and Toumelin, N. (2001).
Procedures planner for an auv.
12^{th} International Symposium on Unmanned Untethered Submersible Technology.
Durham.
[citado en la pág. 171]

[Barrouil and Lemaire, 1998] Barrouil, C. and Lemaire, J. (1998).
An integrated navigation system for a long range auv.
IEEE Oceanic Engineering Society, OCEANS'98: "Engineering for Sustainable Use of the Oceans"(1):1–5.
[citado en la pág. 88, 93, 94, 95, 106, 169, 171, 296]

[Borges de Sousa and Lobo Pereira, 2002] Borges de Sousa, J. and Lobo Pereira, F. (2002).
Coordinated control strategies for networked vehicles: an application to autonomous underwater vehicles.

Departamento de Engenharia Electrotécnica e de Computadores. Faculdade de Engenharia da Universidade do Porto.
Rua Dr. Roberto Frias. 4200-465, Porto. Portugal.
[citado en la pág. 13]

[Bray et al., 2004] Bray, T., Paoli, J., Sperberg-McQueen, C. M., Maler, E., Yergeau, F., and Cowan, J. (2004).
Extensible Markup Language (XML) 1.1.
W3C (World Wide Web Consortium), W3C (http://www.w3.org), w3c recommendation 04 february 2004 edition.
Enlace web: http://www.w3.org/TR/2004/REC-xml11-20040204.
[citado en la pág. 118]

[Brooks, 1985] Brooks, R. (1985).
A robust layered contol system for a mobile robot.
Massachusetts Institute of Technology. Artificial Intelligence Laboratory, pages 1–26.
AI Memo 864.
[citado en la pág. 166]

[Bräunl et al., 2006] Bräunl, T., Boeing, Koestler, Petitt, Ruehl, Bielohlawek, and Haas (2006).
Subsim — an autonomous submarine simulation system.
http://robotics.ee.uwa.edu.au/auv/subsim.html.
[citado en la pág. 264]

[Carreras Pérez, 2003] Carreras Pérez, M. (2003).
A Proposal of a Behavior-based Control Architecture with Reinforcement Learning for an Autonomous Underwater Robot.
PhD thesis, University of Girona, Girona.
Departament of Electronics, Informatics and Automation.
[citado en la pág. 12, 102, 104, 106, 164, 165, 167, 172, 173, 174, 366]

[Carreras Pérez et al., 2003] Carreras Pérez, M., Batlle, J., and Ridao, P. (2003).
Reactive control of an auv using motor schemas.
Computer Vision and Robotics Group, Institute of Informatics and Applications (University of Girona)(1):6.
Edifici Politècnica II, Campus Montilivi, 17071 Girona, Spain.
[citado en la pág. 12, 172, 173]

[Castellucci et al., 2006] Castellucci, F., Prud'homme, C., and Dulimarta, H. (2006).
Corelinux++: The corelinux consortium.
http://corelinux.sourceforge.net/index.php (Fuentes, API y Documentación).
Versión libcorelinux++ 0.4.32 (01-15-2004).
[citado en la pág. 347]

[Cline et al., 1998] Cline, M., Lomow, G., and Girou, M. (1998).
C++ FAQs.
Addison Wesley, 2 edition.
ISBN: 0-201-30983-1.
[citado en la pág. 19]

[Costello, 2004] Costello, R. (2004).
Xml schemas.
XML Technologies Course.
Presentaciones basadas en:
http://www.w3.org/TR/xmlschema-0 (Primer),
http://www.w3.org/TR/xmlschema-1 (Structures),
http://www.w3.org/TR/xmlschema-2 (Datatypes).

[citado en la pág. 118]

[Dawes et al., 2006] Dawes, B., Abrahams, D., and Rivera, R. (2006).
Boost c++ libraries.
http://www.boost.org (Librerías y Documentación).
[citado en la pág. 24, 347]

[Domínguez Brito, 2003] Domínguez Brito, A. (2003).
CoolBOT: un Marco de Programación Orientado a Componentes para Robótica.
PhD thesis, Universidad de Las Palmas de Gran Canaria, Las Palmas de Gran Canaria. Departamento de Informática y Sistemas.
[citado en la pág. 24, 161, 162, 174, 175, 192, 217, 218, 220, 299, 348]

[Drepper, 2006] Drepper, U. (2006).
How to write Shared Libraries.
Red Hat Inc.
[citado en la pág. 22]

[Duarte Oliveira, 2003] Duarte Oliveira, R. (2003).
Supervisão e Controlo da Missão de Veículos Autónomos.
PhD thesis, Universidade Técnica de Lisboa. Instituto Superior Técnico (IFT), Lisboa. Dissertação para obtenção do grau de mestre em engenharia electrotécnica e de computadores.
[citado en la pág. 171, 287, 289, 290, 291, 292, 294]

[Díaz, 2007] Díaz, A. A. (2007).
Tecnologías marinas para la exploración del océano.
IMEDEA, pages 1–8.
[citado en la pág. 5, 44]

[Díaz, 2002] Díaz, D. (2002).
GNU Prolog. A Native Prolog Compiler with Constraint Solving over Finite Domains.
INRIA (The French National Institute for Research in Computer Science and Control) y Panthéon Sorbonne Université Paris 1, edición 1.7, para gnu prolog versión 1.2.16 (gprolog-1.2.16) edition.
Enlace web: http://gnu-prolog.inria.fr; dentro de la sección de documentación.
[citado en la pág. 339]

[Eckel, 2007] Eckel, B. (2007).
Pensar en C++, volume 1.
Traducción inacabada de Thinking in C++.
[citado en la pág. 19]

[Evans et al., 2003] Evans, J., Petillot, Y., Redmond, P., Reed, S., and Lane, D. (2003).
Autotracker: Real-time pipeline and cable tracking technologies for auvs.
In G.Ñ. Roberts, R. S. and Allen, R., editors, *Guidances and Control of Underwater Vehicles 2003*. ELSEVIER IFAC Publications, Ocean Systems Laboratory, Heriot-Watt University, Edinburgh, EH14 4AS, Scotland, UK.
[citado en la pág. 76]

[Fallside and Walmsley, 2004] Fallside, D. C. and Walmsley, P. (2004).
XML Schema Part 0, XML Schema Part 1 y XML Schema Part 2.
W3C (World Wide Web Consortium), W3C (http://www.w3.org), w3c recommendation 28 october 2004 edition.
Parte 0: http://www.w3.org/TR/2004/REC-xmlschema-0-20041028,
Parte 1: http://www.w3.org/TR/2004/REC-xmlschema-1-20041028,
Parte 2: http://www.w3.org/TR/2004/REC-xmlschema-2-20041028.
[citado en la pág. 118]

[Fossen, 1994] Fossen, T. (1994).
Guidance and Control of Ocean Vehicles.
John Wiley & Sons Ltd., University of Trondheim, Norway, 1 edition.
[citado en la pág. 41]

[Fossen, 2002] Fossen, T. (2002).
Marine Control Systems. Guidance, Navigation and Control of Ships, Rigs and Underwater Vehicles.
Marine Cybernetics AS, Trondheim, Norway, 1 - 3ª impresión edition.
Enlace web: http://www.marinecybernetics.com.
[citado en la pág. 29, 41, 140]

[Gamma et al., 2003] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (2003).
Patrones de Diseño. Elementos de software orientado a objetos reutilizable.
Addison Wesley, Madrid.
Traducción a español por César Fernández Acebal (Universidad de Oviedo).
[citado en la pág. 19, 29, 219, 284, 330]

[Gayo, 2006] Gayo, J. L. (2006).
Aplicaciones avanzadas en xml: Web semántica.
Curso de Doctorado. Universidad Pontificia de Salamanca.
[citado en la pág. 118]

[Gerkey et al., 2006] Gerkey, B., Vaughan, R., Howard, A., Koenig, N., et al. (2006).
The player project: Player, stage, gazebo.
http://playerstage.sourceforge.net (Fuentes, API, Ejemplos y Referencias).
Versión Player 2.0.3 (26-09-2006).
[citado en la pág. 24, 177, 217, 300, 304, 321, 322, 348]

[Group, 1987] Group, N. W. (1987).
XDR: eXternal Data Representation standard.
Sun Microsystems, Inc., rfc 1014 - xdr edition.
Enlace web: http://www.faqs.org/rfcs/rfc1014.html.
[citado en la pág. 303, 317]

[Gülkü, 2002] Gülkü, C. (2002).
The Complete log4j Manual.
Draft. A reliable, fast and flexible logging framework for Java.
Manual for log4j version 1.2 and later.
Web oficial: http://logging.apache.org.
[citado en la pág. 118, 155, 249, 335]

[Hanrahan and Bellingham, 2008] Hanrahan, C. and Bellingham, J. (2008).
Autonomous ocean sampling network (aosn).
http://www.mbari.org/aosn.
[citado en la pág. 14]

[Hedström, 2006] Hedström, A. (2006).
C++ sockets library.
http://www.alhem.net/Sockets/index.html (Fuentes, Tutorial, Ejemplos, Foro y Licencias).
Versión C++ Sockets 2.0.6 (28-08-2006).
[citado en la pág. 338]

[Hernández Sosa, 2003] Hernández Sosa, J. (2003).
Adaptación computacional en sistema percepto-efectores. Propuesta de arquitectura y políticas de control.

PhD thesis, Universidad de Las Palmas de Gran Canaria, Las Palmas de Gran Canaria. Departamento de Informática y Sistemas.
[citado en la pág. 162, 164, 168, 174]

[Issac et al., 2005] Issac, M., Adams, S., He, M., Bose, N., Williams, C., and Bachmayer, R. (2005).
Manoeuvering experiments using the mun explorer auv.
Memorial University of Newfoundland (MUN), St. John's, NL, A1B 3X5, Canada.
[citado en la pág. 70, 71, 72]

[Johnson et al., 2006] Johnson, A., de Vienne, C., and Cumming, M. (2006).
libxml++ - c++ wrapper for the libxml xml parser library (libxml2).
http://libxmlplusplus.sourceforge.net (Fuentes, API, Documentación, etc.).
Versión libxml++-2.6 - API 2.14 (13-03-2006).
[citado en la pág. 334]

[Karlsson et al., 2005] Karlsson, N., Di Bernardo, E., Ostrowski, J., Goncalves, L., Pirjanian, P., and Munich, M. (2005).
The vslam algorithm for robust localization and mapping.
Proc. of Int. Conf. on Robotics and Automation (ICRA) 2005, Evolution Robotics, Inc. Pasadena, California, USA.
[citado en la pág. 44]

[Laboratory, 2006] Laboratory, I. S. (2006).
SICStus Prolog User's Manual.
Swedish Institute of Computer Science, PO Box 1263. SE-164 29 Kista, Sweden, release 3.12.5 edition.
Enlace web: http://www.sics.se/sicstus/; dentro de la sección de documentación.
[citado en la pág. 339]

[Lamport, 2000] Lamport, L. (2000).
LaTeX. A Document Preparation System. User's Guide and Reference Manual.
Addison Wesley, 2 edition.
Updated for LaTeX 2ε.
[citado en la pág. 27]

[Madhan et al., 2005] Madhan, R., Desa, E., Prabhudesai, S., Sebastiao, L., Pascoal, A., Desa, E., Mascarenhas, A., Maurya, P., Navelkar, G., Afzulpurkar, S., and Khalap, S. (2005).
Mechanical design and development aspects of a small auv — maya.
National Institute of Oceanography, Dona Paula, Goa 403 004, India, pages 1–6.
[citado en la pág. 5]

[Martínez de Sousa, 1995] Martínez de Sousa, J. (1995).
Diccionario de ortografía técnica.
Fundación Germán Sánchez Ruipérez.
[citado en la pág. 27]

[Muñoz Andrés, 1991] Muñoz Andrés, V. (1991).
Química Técnica, volume Tomo I.
UNED, Madrid, 1 edition.
Análisis Dimensional.
[citado en la pág. 264]

[Newman and Leonard, 2002] Newman, P. and Leonard, J. (2002).
Pure range-only sub-sea slam.
Massachusetts Institute of Technology, pages 1–7.
[citado en la pág. 44]

[Oetiker et al., 2006] Oetiker, T., Partl, H., Hyna, I., and Schlegl, E. (2006).
The Not So Short Introduction to LaTeX 2ε.
4.20 edition.
Or LaTeX 2ε in 139 minutes.
[citado en la pág. 27]

[Petillot et al., 2003] Petillot, Y., Reed, S., and Bell, J. (2003).
Real time auv pipeline detection and tracking using side scan sonar and multi-beam echo-sounder.
Oceans Systems Laboratory, Heriot Watt University. School of EPS.
RICCARTON Campus. Edinburgh, EH14 4AS, UK.
[citado en la pág. 76]

[Petri and Reisig, 2008] Petri, C. and Reisig, W. (2008).
Petri net.
Scholarpedia, 3(4):6477.
The free peer reviewed encyclopedia. Revision 37208.
[citado en la pág. 287, 294, 295]

[Pressman, 1993] Pressman, R. (1993).
Ingeniería del Software: un enfoque práctico.
McGraw-Hill, 5 edition.
ISBN: 8448132149.
[citado en la pág. 15]

[Ramalho Oliveira et al., 1996] Ramalho Oliveira, P., Pascoal, A., Silva, V., and Silvestre, C. (1996).
Design, development and testing of a mission control system for the marius auv.
Department of Electrical Engineering. Institute for Systems and Robotics, Instituto Superior Técnico (IST)(1):20.
Av. Rovisco Pais, 1096 Lisboa Codex, Portugal.
[citado en la pág. 11, 93, 94, 95, 106, 169, 171, 295, 296]

[Ramalho Oliveira et al., 1998] Ramalho Oliveira, P., Pascoal, A., Silva, V., and Silvestre, C. (1998).
The mission control system of the marius auv: System design, implementation and tests at sea.
International Journal of Systems Science, 29(10):1065–1080.
Special Issue on Underwater Robotics.
[citado en la pág. 11, 171, 295]

[Ridao, 2006] Ridao, P. (2006).
Airsub — aplicaciones industriales de los robots submarinos.
http://eia.udg.es/~pere/airsub/index.htm.
[citado en la pág. 74]

[Ridao et al., 2005] Ridao, P., Yuh, J., Batlle, J., and Sugihara, K. (2005).
On auv control architecture.
Informatics and Applications Institute (University of Girona). Mechanical Engineering Dept., Information and Computer Science Dept. (University of Hawaii at Manoa), ITOCA (Intelligent Task-Oriented Control Architecture)(1):6.
Avda. Lluis Santaló s/n Girona CP. 17007 Spain. Honolulu, Hawaii 96822, USA.
[citado en la pág. 11, 99, 169, 171, 172]

[Roberts et al., 2003] Roberts, G., Sutton, R., and Allen, R. (2003).
Guidance and control of underwater vehicles.
Elsevier Science and Technology, IFAC Proceedings Volumes(1):1–40.

[citado en la pág. 98, 99, 101, 102, 171, 172]

[Rudnick and Perry, 2003] Rudnick, D. and Perry, M. (2003).
Autonomous and lagrangian platforms and sensors. alps workshop report.
http://www.geo-prose.com/ALPS.
31 marzo – 2 abril.
[citado en la pág. 5]

[Santana Jorge, 2007] Santana Jorge, F. (2007).
Compilador/generador de esqueletos c++ para componentes coolbot.
Master's thesis, Universidad de Las Palmas de Gran Canaria, Las Palmas de Gran Canaria. Facultad de Informática.
[citado en la pág. 263, 278]

[Sattar, 2005] Sattar, J. (2005).
A visual servoing system for an amphibious legged robot.
Master's thesis, School of Computer Science. McGill University, Montréal, Québec.
[citado en la pág. 44]

[Schmidt, 2006] Schmidt, D. (2006).
C++ Network Programming with Patterns, Frameworks, and ACE.
DOC group, Department of EECS. Vanderbilt University.
Tlf. (615) 343-8197.
[citado en la pág. 29]

[Simmons and Apfelbaum, 1998] Simmons, R. and Apfelbaum, D. (1998).
A task description language for robot control.
Proceedings of Conference on Intelligent Robotics and Systems, pages 1–7.
[citado en la pág. 99]

[Simmons and Apfelbaum, 2003] Simmons, R. and Apfelbaum, D. (2003).
Tdl. task description language.
http://www.cs.cmu.edu/~tdl.
[citado en la pág. 99]

[Spinellis, 2001] Spinellis, D. (2001).
Notable design patterns for domain specific languages.
Journal of Systems and Software, 56(1):91–99.
[citado en la pág. 16, 281, 282, 283, 284, 285]

[Sutter, 2000] Sutter, H. (2000).
Pimples — beauty marks you can depend on.
More C++ Gems. Cambridge University Press, C++ Report(1).
[citado en la pág. 19]

[Turner and Mailman, 1999] Turner, R. and Mailman, J. (1999).
Interfacing the orca auv controller to the nps uvw and to a land robot.
In *11th International Symposium on Unmanned Untethered Submersible Technology (UUST99)*. Department of Computer Science, University of Maine, Orono, ME 04469-5752.
http://cdps.umcs.maine.edu/Papers/1999/UUST99.
[citado en la pág. 164]

[Usuarios Wikipedia, 2008a] Usuarios Wikipedia (2008a).
Domain-specific language (wikipedia).
http://en.wikipedia.org/wiki/Domain-specific_language.
[citado en la pág. 16, 279]

[Usuarios Wikipedia, 2008b] Usuarios Wikipedia (2008b).
Petri net (wikipedia).
http://en.wikipedia.org/wiki/Petri_net.
[citado en la pág. 287, 289, 294, 295]

[Usuarios Wikipedia, 2008c] Usuarios Wikipedia (2008c).
Red de petri (wikipedia).
http://es.wikipedia.org/wiki/Red_de_Petri.
[citado en la pág. 287, 294, 295]

[Valavanis et al., 1997] Valavanis, K., Gracanin, D., Matijasevic, M., Kolluru, R., and Demetriou, G. (1997).
Control architectures for autonomous underwater vehicles.
IEEE Control Systems, pages 48–64.
[citado en la pág. 9, 162, 163, 166, 167, 169, 174]

[Vaughan et al., 2000] Vaughan, G., Elliston, B., Tromey, T., and Taylor, I. (2000).
GNU AUTOCONF, AUTOMAKE AND LIBTOOL — The Goat Book, new riders (sams publishing, subsidiaria de pearson education) edition.
[citado en la pág. 22]

[Veillard, 2006] Veillard, D. (2006).
libxml2 - the xml c parser and toolkit of gnome (libxml).
http://xmlsoft.org (Fuentes, API, Ejemplos y Licencia MIT).
Versión libxml2-2.6.26 (06-06-2006).
[citado en la pág. 334]

[Wang, 2007] Wang, D. (2007).
Autonomous Underwater Vehicle (AUV) Path Planning and Adaptive On-board Routing for Adaptive Rapid Environmental Assessment.
PhD thesis, University of Girona, Massachusetts Institute of Technology.
[citado en la pág. 71, 72]

[Wielemaker, 2004] Wielemaker, J. (2004).
SWI-Prolog 5.5. Reference Manual.
Universidad de Amsterdam, Dept. of Social Science Informatics (SWI). Roeterstraat 15, 1018 WB Amsterdam. Holanda. Tlf. (+31) 20 5256121, versión 5.5.2 edition.
Enlace web: http://www.swi-prolog.org; dentro de la sección de documentación.
[citado en la pág. 339, 341]

Parte V

Apéndices

Apéndice A

Proyectos de Fin de Carrera complementarios

Este proyecto, titulado Sistema Integrado de Control para un Vehículo Submarino Autónomo, se intregra dentro de una terna de Proyectos de Fin de Carrera (PFC) que cubren el análisis, diseño e implementación[1] de diferentes productos relacionados con un escenario de gestión de AUVs.

En resumen, estos proyectos consisten en (se incluye este mismo proyecto por completitud):

Sistema de Control Bajo el título de **Sistema Integrado de Control para un Vehículo Submarino Autónomo**, consiste en el análisis de arquitecturas típicas para AUVs, adopción y diseño de una arquitectura propia, definición de formatos de especificación del equipamiento y misiones ejecutables por un AUV. Opcionalmente, implementación básica de la arquitectura del sistema del AUV propuesta (*Sick***AUV**) para demostrar la bondad de la misma con algunas pruebas y resultados. Soporte de comunicaciones por TCP/IP y un protocolo de comunicaciones común, basado en XDR. Soporte opcional para coordinación de AUVs en entornos multi-AUV y para simulación HIL (Hardware In Loop). Soporte opcional para diferentes interfaces de comunicación. Soporte opcional para supervisión del sistema, control de excepciones y ejecución de planes de contingencia. Algoritmos de control para un módulo de navegación integrando las medidas sensoriales de instrumentos de navegación.

Planificador Bajo el título de **Sistema Integrado de Planificación y Control de Misiones con Vehículos Submarinos Autónomos**, se trata de una interfaz para la definición y validación de misiones, tras la definición de formatos de especificación del equipamiento y misiones ejecutables por un AUV. Monitorización y control remoto de AUVs operativos, mediante comunicación por TCP/IP y un protocolo de comunicaciones común, basado en XDR. Realización del análisis, diseño

[1] En principio, sólo se realizará una implementación básica, ya que los proyectos empiezan desde cero, lo que requiere un estudio o análisis y diseño inicial bastante profundo.

y prototipo de la interfaz, junto con una implementación básica. Soporte opcional para entornos multi-AUV.

Simulador Bajo el título de **Entorno de Simulación para el Desarrollo de Misiones con Vehículos Submarinos Autónomos**, se trata de un simulador del entorno y de AUVs. Análisis de arquitecturas de simuladores, especialmente orientados a entornos de operación de AUVs, consideran un espacio tridimensional. Adopción y diseño de una arquitectura propia. Definición de formatos de especificación del equipamiento y misiones ejecutables por un AUV. Diseño de servidor de simulación de modelos físico-químicos del entorno y de clientes AUV, con implementación básica. Soporte opcional para sensores virtuales integrables en simulaciones HIL, mediante comunicación por TCP/IP y un protocolo de comunicaciones común, basado en XDR.

Se puede observar como determinadas tareas son comunes a varios proyectos. Por este motivo se han realizado de forma coordinadas. Se trata de las siguientes tareas:

Definición de equipamiento Estudio de características de AUVs y vehículos similares para la definición del equipamiento de los mismos. Diseño del formato de especificación en XML con esquemas de validación XML y ejemplos de definición para un hipotético AUV.

Definición de misiones Estudio de misiones típicas de AUVs (véase el Capítulo 11) y vehículos similares (véase el Capítulo 10 para algunos ejemplos). Diseño del formato de especificación en XML con esquemas de validación XML (véase el Capítulo 13) y ejemplos de definición para un hipotético AUV, siguiendo la arquitectura de definición de misiones adoptada (véase la Sección 12.2), basada en planes.

Protocolo de comunicaciones Protocolo de comunicación entre interfaz de planificación, AUV y simulador. Tiene como finalidad permitir el control remoto del AUV, el envío de misiones y la monitorización. El AUV tendrá la capacidad de envío de muestras. Protocolo de comunicación con el simulador para uso de sensores virtuales o entorno virtual, permitiendo simulación HIL. Las pruebas de comunicación también forman parte importante de esta tarea.

Serialización de datos Serialización de datos en un formato independiente del lenguaje de programación. Se ha optado por XDR (véase el Sección E.9) y se ha creado una librería en el lenguaje de programación necesario según el proyecto.

Existe otra serie de proyectos relacionados con éste:

Generador de componentes CoolBOT Se trata de una contribución realizada en un PFC [Santana Jorge, 2007].

Prototipo de AUV Se ha diseñado un prototipo de AUV con vistas a dar continuidad a esta línea de investigación.

Boyas Oceanográficas Se ha desarrollado un software similar al formado por los proyectos del **sistema de control** y **planificador**, como parte de la colaboración en una beca de la Facultad de Telecomunicaciones.

Apéndice B

Lenguaje Específico de Dominio

> *In all branches of science and engineering one can distinguish between approaches that are* generic *and those that are* specific.
> *A specific approach provides a much better solution for a smaller set of problems.*
> *One of the incarnations of this dichotomy in computer science is:* domain-specific languages *versus* generic programming languages.
>
> — Arie van Deursen et al.
> *1998* (Dutch Telematica Instituut)

Un Lenguaje Específico de Dominio (DSL en lo sucesivo) es cualquier tipo de lenguaje formal —normalmente de programación— que está expresamente orientado a un dominio muy concreto. En la Sección B.1 y en la Definición B.1 se muestra una definición más precisa de un DSL en base a la información de [Usuarios Wikipedia, 2008a].

B.1. Definición

En la actualidad los DSL se han hecho bastante populares en el desarrollo software y forman parte de los lenguajes de 4ª generación (4GL). En la Definición B.1 se muestra la definición de DSL. En contraposición a los DSL tenemos los Lenguajes de programación de Propósito General (GPL en lo sucesivo), que se definen en la Definición B.2. En el Ejemplo B.1 se muestran varios ejemplos de DSL bastante conocidos.

Definición B.1 (DSL). *Un DSL es un lenguaje de programación o de especificación dedicado a un dominio particular de problemas, una técnica de representación de un problema particular o una técnica de una solución particular.*

Definición B.2 (GPL). *Un GPL es un lenguaje de programación aplicable a problemas de un amplio espectro de dominios —normalmente cualquier tipo de problema—, v. g. C o Java. También existen lenguajes de modelado de propósito general, v. g. UML.*

Ejemplo B.1 (Ejemplos de DSL). *A continuación se enumeran los ejemplos más típicos y conocidos de DSL intentando cubrir dominios diferentes para remarcar la especificidad de los mismos:*

Fórmulas de hojas de cálculo

Macros

Gramáticas YACC *Gramáticas BNF o EBNF para crear analizadores sintácticos (para validar las construcciones de un determinado lenguaje generado por la gramática) —i. e.* parsers*— en YACC*[1].

Expresiones regulares *Expresiones regulares para crear analizadores léxicos (para obtener* tokens*) —i. e.* scanners*— en Lex*[2].

GraphViz *Lenguaje de entrada de GraphViz para formatear gráficos.*

XML *Es un lenguaje de marcas extensible que, en realidad, es un metalenguaje que permite definir una gramática de lenguajes específicos. Este DSL ha sido usado para la definición formal de la misión propuesta para el AUV (véase la Sección 12.2 y, para más detalles, el Capítulo 13).*

Otros *Otros ejemplos son LaTeX, CSound, Generic Eclipse Modeling System (GEMS), etc.*

La creación de un DSL puede ser realmente útil si el lenguaje permite expresar un tipo particular de problemas o soluciones a éstos de forma más clara que con lenguajes ya existentes, y el problema en cuestión se repite con la suficiente frecuencia —este es precisamente el caso de los planes de la misión propuesto para un AUV (véase la Sección 12.2). Este proceso es conocido como Programación Orientada al Lenguaje (LOP en lo sucesivo), como se define en la Definición B.3.

Definición B.3 (LOP). *LOP considera la creación de DSLs para expresar problemas como una parte del proceso de solución del problema.*

B.2. Ventajas y desventajas

El uso de un DSL —incluso su creación, i. e. aplicar LOP— frente a un GPL tiene varias ventajas y desventajas. Será el área de aplicación la que determine la elección final (véase la Sección 12.2 para conocer las razones de la elección de un DSL para la especificación de la misión).

La principales ventajes de un DSL son:

[1] Actualmente, en lugar de YACC se suele usar Bison.
[2] Actualmente, en lugar de Lex se suele usar Flex.

1. Permite expresar las soluciones con la semántica, terminología y el nivel de abstracción del problema del dominio. Por tanto, los propios expertos en el dominio pueden comprender, validar, modificar y —habitualmente incluso— desarrollar ellos mismos programas escritos con un DSL.
2. Código auto-documentado.
3. Mejora la calidad, productividad, fiabilidad, mantenibilidad, portabilidad y reusabilidad.
4. Permite la validación en el nivel del dominio. Mientras las construcciones del lenguaje sean seguras cualquier sentencia que se escriba con ellas puede considerarse segura.

Por contra, existen las siguientes desventajas:

1. Coste de diseño, implementación y mantenimiento.
2. Búsqueda, configuración y mantenimiento del alcance adecuado.
3. Dificultad al alcanzar el equilibrio entre la especificidad del dominio y las construcciones del GPL.
4. Posible pérdida de eficiencia en comparación con software escrito a más bajo nivel.
5. Difícil o imposible de depurar.

La anterior enumeración de ventajas y desventajas es aplicable de forma general a cualquier DSL. No obstante, determinados DSL pueden estar diseñados especialmente para aliviar el impacto de determinadas desventajas, v. g. creando un DSL con XML algunas desventajas se suavizan, como el coste de diseño, implementación y mantenimiento, gracias al hecho de que ya existen múltiples herramientas para analizar sintácticamente especificaciones escritas en XML (véase el Sección E.3, especialmente la **??** para conocer un estudio del estado del arte de herramientas de validación de "programas" escritos en el DSL basado en XML).

B.3. Patrones de diseño

> *Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.*
>
> — Christopher Alexander et al.
> *1977* (Architect)

De acuerdo con [Spinellis, 2001] existe un serie de patrones de diseño aplicables a la hora de seguir el paradigma LOP, i. e. al desarrollar un DSL. En esta fuente se describe diseños reusables —i. e. patrones de diseño— para el desarrollo de un DSL.

La Figura B.1 muestra la arquitectura básica de un sistema basado en un DSL. Para la especificación de la misión del AUV se ha desarrollado un DSL —basado en XML— en el que se identifica la aplicación de algunos de los patrones de diseño comentados en [Spinellis, 2001] (véase la Sección 12.2 y el Capítulo 13).

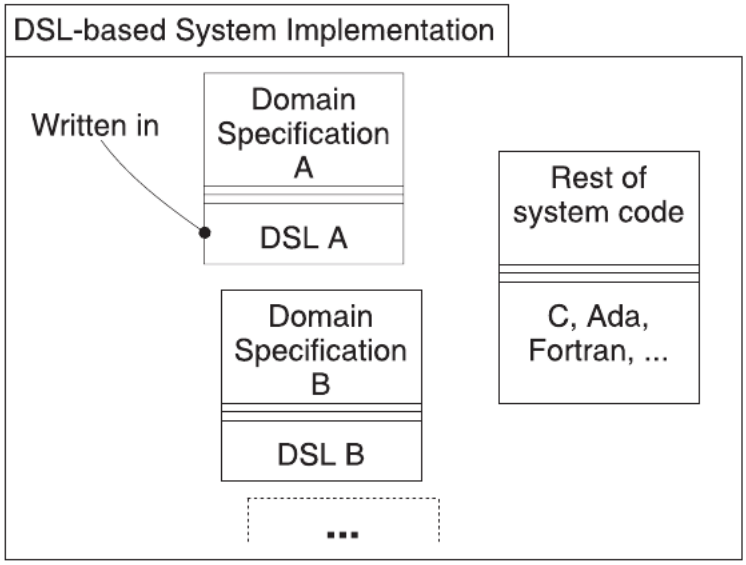


Figura B.1: Diagrama UML de arquitectura de sistema basado en DSL

[Spinellis, 2001] plantea varios patrones de diseño[3] para el desarrollo del DSL, los cuales se explican brevemente a continuación:

Piggyback Lit. succión, piggyback (véase la Figura B.2 (a)) es un patrón estructural en el que se usan las capacidades de un lenguaje existente como base para el nuevo DSL. Al diseñar un DSL sobre un lenguaje existente se consigue directamente el soporte lingüístico de manejo de expresiones, variables, subrutinas o compilación. El ejemplo más típico es *YACC* y *Lex*[4]; tomar XML como base también puede considerarse como ejemplo de aplicación de este patrón, ya que aprovechamos su sintaxis —basadas en etiquetas/marcas— y su tipado —i. e. elementos, atributos y tipos de datos de sus valores.

Pipeline Lit. tubería, pipeline (véase la Figura B.2 (b)) es un patrón de comportamiento que soluciona el problema de la composición de DSLs. Normalmente un sistema se describe mejor usando una familia de DSLs —v. g. DSL1, DSL2, DSL3. Un ejemplo claro son las herramientas de procesado de texto de *troff* —i. e. *eqn*, para procesar ecuaciones, *tbl*, para procesar tabla, etc.— (véase [Spinellis, 2001] para más información).

Lexical processing El patrón de diseño de procesamiento léxico es de creación (véase la Figura B.2 (c)) y ofrece una forma eficiente para el diseño e implementación de

[3] Se mantienen los nombres usados en [Spinellis, 2001], si bien se aporta una traducción libre a español junto con la explicación.

[4] Actualmente se suelen usar versiones más modernas, conocidas como *Bison* y *Flex*.

DSLs. Se trata de un diseño de una forma que se ajuste al procesamiento basado en técnicas de substitución léxica simple, v. g. DSL implementados usando herramientas como *sed*, *awk*, *Perl*, *Python*, *m4* y el preprocesador de C —i. e. CPP— (véase la bibliografía de [Spinellis, 2001] para más información).

Language extension El patrón de diseño de extensión del lenguaje es de creación (véase la Figura B.2 (d)). Se usa para añadir nuevas características a un lenguaje existente. El DSL se diseña e implementa como una extensión de un lenguaje base al que se añaden las nuevas características necesarias —v. g. nuevos tipos de datos, elementos semánticos, *syntactic sugar*[5]— a su núcleo. Algunos casos de aplicación de este patrón de diseño lo son *cfront* (compilador original de C++), extensiones a los tipos de Java, etc. (véase la bibliografía de [Spinellis, 2001] para más información).

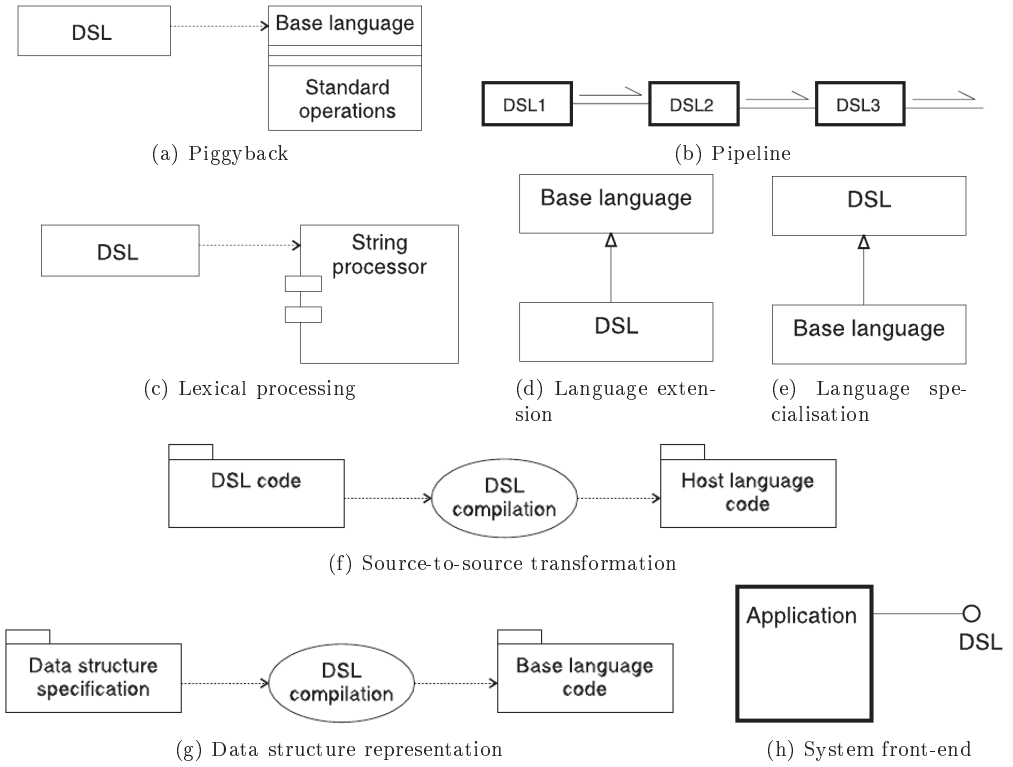
Language specialisation El patrón de diseño de especialización del lenguaje es de creación (véase la Figura B.2 (e)). A diferencia del patrón de extensión del lenguaje, éste elimina —en lugar de añadir— características del lenguaje base para crear el DSL. Este patrón se suele aplicar cuando los requerimientos de seguridad pueden satisfacerse simplemente eliminando algunos aspectos inseguros —v. g. asignación de memoria dinámica, punteros sin límites, hilos— de un lenguaje. Algunos casos de aplicación de este patrón son *Javalight* —que es un subconjunto de *Java* con tipado fuerte—, subconjuntos de C para aplicaciones de automoción, etc. (véase la bibliografía de [Spinellis, 2001] para más información).

Source-to-source transformation Lit. Transformación código-a-código, es un patrón de diseño de creación (véase la Figura B.2 (f)) que permite la implementación eficiente de traductores de DSL. El código fuente del DSL se transforma mediante un proceso de traducción —superficial o profundo— en el código fuente de un lenguaje existente. Este es el caso del generador de componentes CoolBOT desarrollado para facilitar la definición e implementación de éstos (véase el Sección E.1).[6] Aplicando este patrón de diseño las herramientas disponibles para el lenguaje existente permitirán compilar o interpretar el código generado tras el proceso de transformación. Algunos lenguajes, como *C*, ofrecen mecanismos para especificar el fichero y línea de código fuente del DSL que generó una secuencia particular de instrucciones del código final.

Data structure representation El patrón de diseño de representación de estructuras de datos es de creación (véase la Figura B.2 (g)) que permite la especificación declarativa y específica del dominio de datos complejos. El código orientado a datos suele depender de estructuras de datos inicializadas cuya complejidad hace difícil su escritura y mantenimiento. Estas complicadas estructuras de datos se expresan mejor con un lenguaje que con su representación subyacente —v. g. la lista de nodos adyacente de un grafo se puede expresar fácilmente como una lista de conexiones de caminos. Ejemplos de aplicación de este patrón de diseño son *FIDO* —diseñado para expresar de forma concisa conjuntos regulares de ristras y árboles— y las

[5] El *azúcar sintáctico* (*syntactic sugar*) es el hecho de proporcionar una construcción sintáctica equivalente a otra, pero que habitualmente aporta una semántica más clara.

[6] Un desarrollo equivalente —i. e. para la generación de componentes CoolBOT— se ha desarrollado siguiendo otra metodología como parte de un PFC (véase la Apéndice A).



(a) Piggyback (b) Pipeline (c) Lexical processing (d) Language extension (e) Language specialisation (f) Source-to-source transformation (g) Data structure representation (h) System front-end

Figura B.2: Patrones de diseño de DSL

tablas de inicialización de *YACC* y *Lex* (véase la bibliografía de [Spinellis, 2001] para más información).

System front-end La configuración y adaptación de un sistema puede delegarse en un DSL como *front-end*[7]. Los sistemas con cientos de opciones de configuración se pueden beneficiar de hacer que el sistema sea programable mediante un DSL como *front-end*. Este patrón de diseño es estructural (véase la Figura B.2 (h))y ofrece un mecanismo declarativo, mantenible, organizado y abierto para configurar y adaptar el sistema. Destacan como ejemplos los basados en el lenguaje *lisp* —v. g. *Emacs*, *AutoCAD*— y lenguajes como *Tcl*, *Perl*, etc. (véase la bibliografía de [Spinellis, 2001] para más información).

En el Cuadro B.1 se muestran de forma resumida los patrones de diseño para DSL antes comentados. Se indica el nombre del patrón de diseño, su tipo, una descripción corta y algunos ejemplos significativos; por el ejemplo CoolBOT se entiende el generador de componentes CoolBOT (véase el Sección E.1). Respecto al tipo de patrón de diseño, de acuerdo con [Gamma et al., 2003], se distinguen tres grandes grupos de patrones de diseño:

Estructural

de Creación

[7] Elementos con los que tiene contacto el usuario final.

Nombre	Tipo	Descripción	Ejemplo
Piggyback	Estructural	Usa capacidades de lenguaje existente	*YACC*, *Lex*
Pipeline	Comportamiento	Describe problema con familia de DSL	*troff*
Lexical processing	Creación	Procesamiento eficiente basado en técnicas de substitución léxica	*sed*, *awk*, *Perl*, ...
Language extension	Creación	Añade características al lenguaje existente	*cfront*
Language specialisation	Creación	Elimina características del lenguaje existente	*Javalight*
Source-to-source transformation	Creación	Traduce DSL a lenguaje existente	CoolBOT
Data structure representation	Creación	Especificación declarativa y específica del dominio de datos complejos	*FIDO*, *YACC*, *Lex*
System front-end	Estructural	Configuración y adaptación del sistema delegada en DSL	*lisp* (*AutoCAD*), *Tcl*, ...

Cuadro B.1: Patrones de Diseño para DSL

de Comportamiento

Para saber más sobre patrones de diseño se remite al lector a la **??** o a la bibliografía relacionada.

En la Figura B.3 se describen las relaciones entre los diferentes patrones de diseño para DSL. Estas interrelaciones son interesantes a la par que típicas en lenguajes enfocados a un dominio específico, tal y como se explica en [Spinellis, 2001], a donde se remite al lector para más información al respecto.

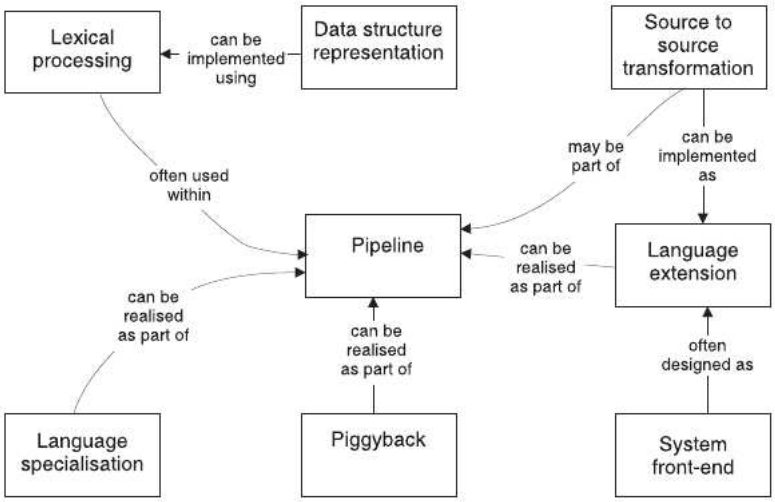


Figura B.3: Relación entre patrones de diseño de DSL

Apéndice C

Redes de Petri

A Petri net is a device that manipulates events according to well-defined rules.
Since rules can be arbitrarily complex, Petri nets are naturally suited to represent a very large class of discrete event systems.

— Paulo J. C. Ramalho Oliveira et al.
1996 (Assistant Professor (IST))

En base a la información general sobre redes de Petri de [Usuarios Wikipedia, 2008b, Usuarios Wikipedia, 2008c, Petri and Reisig, 2008], así como las reseñas indicadas en [Duarte Oliveira, 2003], a continuación se define y explica brevemente en qué consiste una red de Petri. Se ha comentado su viabilidad como arquitectura para la definición de la misión y para el diseño del sistema del AUV en el Capítulo 12 y Capítulo 14, respectivamente.

C.1. Definición

Existen diferentes definiciones equivalentes de las redes de Petri (RdP en lo sucesivo). La definición más común es la mostrada en la Definición C.1; en la Definición C.2 se muestra su representación matemática.

Definición C.1 (Red de Petri). *Una RdP es una herramienta de modelado de sistemas de eventos discretos que consiste en una representación matemática de un sistema distribuido discreto (véase la Definición C.2).*

Definición C.2 (Representación matemática RdP). *Una RdP es un grafo dirigido, bipartito*[1] *y pesado. Se representa por la 5-tupla* $RdP = (P, T, A, w, \mu)$ *donde* P *y* T

[1] En Teoría de grafos se conoce como grafo bipartito a un grafo no dirigido cuyos vértices se pueden separar en dos conjuntos disjuntos V_1 y V_2 y las aristas siempre unen vértices de un conjunto con vértices de otro.

representan los vértices del grafo, conocidos como lugares y transiciones, respectivamente, siendo conjuntos disjuntos.
Representa una relación de flujo entre los lugares y las transiciones: $A \subseteq (P \times T) \cup (T \times P)$.
Esta relación de flujo se denomina arco y se define de modo que los lugares se conecten solamente a las transiciones a través de grafos dirigidos, y del mismo modo las transiciones a los lugares.[2]
w *representa una función de pesos asociada a cada uno de los arcos. Sea* $\mathbb{N}$ *el conjunto de los números naturales, una función de pesos realiza la siguiente correspondencia* $w : A \to \mathbb{N} - \{0\}$.
Una marcación μ *de una RdP es una correspondencia* $\mu : P \to \mathbb{N}$.
Los primeros cuatro elementos de la 5-tupla representan la estructura de la red, mientras que el último elemento representa su estado[3].

Las redes de Petri fueron definidas en los años 1960 por Carl Adam Petri y son una generalización de la teoría de autómatas que permite expresar eventos concurrentes.

Una RdP está formada por una serie de elementos, que pueden verse en la Figura C.1 y son los siguientes:

Lugar El lugar se representa con un círculo. Pueden contener un número cualquiera de marcas —o ninguna.

Transición La transición se representa por un rectángulo. Se puede disparar, i. e. puede consumir marcas de un lugar de inicio y producir marcas en un lugar de llegada. Una transición está habilitada si tiene marcas en todos sus lugares de entrada —i. e. en todos los lugares de los que salen arcos hacia la transición. Se producen marcas en todos los lugares hacia los que la transición tiene arcos.

Arco Se trata de un arco dirigido —i. e. una flecha con punta en un extremo— que conecta un lugar a una transición o viceversa; no puede haber arcos entre lugares ni entre transiciones.

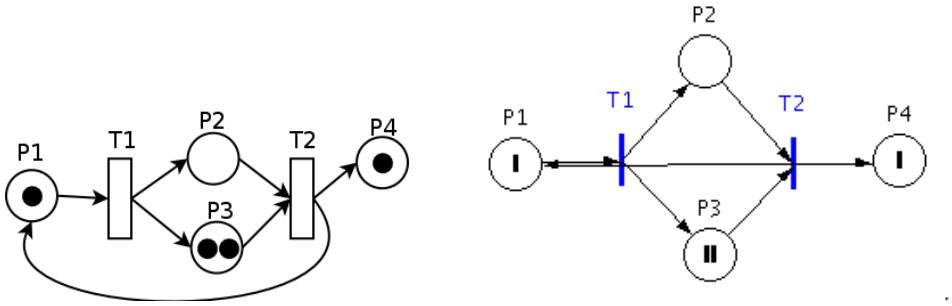
Marca/Ficha La marca o ficha se representa por un punto negro —habitualmente— y puede ocupar un lugar.

A partir de los elementos que forman una RdP se puede construir un diagrama de la misma. Para ello puede usarse software de diagramación o aplicaciones específicas que permiten incluso la simulación de la RdP, como se comenta en la Nota C.1.

Nota C.1 (Software de edición de RdP). *En la Figura C.1 (b) se puede ver el diagrama generado con jPNS para la RdP del ejemplo de la Figura C.1. jPNS es un* applet *de Java para la edición y simulación de redes de Petri. Es bastante completo, pues permite crear el diagrama usando lugares, transiciones, arcos y marcas —denominadas* tokens *en jPNS. Se puede acceder a esta aplicación desde el siguiente enlace web:*
`http://robotics.ee.uwa.edu.au/pns/java/`
También existen otras aplicaciones similares, como otro applet *similar en funcionalidad al anterior, disponible en:*
`http://torguet.free.fr/java/Petri.html`

[2] No existe ningún arco —i. e. relación de flujo— entre dos o más lugares, o entre dos o más transiciones.

[3] El estado de una RdP viene dado por la marcación, i. e. la ubicación de las marcas en los lugares de la RdP.



(a) Diagrama genérico (b) Diagrama creado con jPNS

Figura C.1: Ejemplo de RdP

C.2. Taxonomía

De acuerdo con [Usuarios Wikipedia, 2008b, Duarte Oliveira, 2003] existen siete[4,5] tipos de redes de Petri principales:

1. **Máquina de Estados – State Machine (SM)** Toda transición tiene un arco entrante. Por tanto, no puede haber concurrencia, pero pueden haber conflicto —ie. ¿a dónde debe ir la ficha desde un lugar?, ¿a una transición o a otra?. Matemáticamente:
$$\forall t \in T : |t\bullet| = |\bullet t| = 1 \tag{C.1}$$

2. **Grafo Marcado – Marked Graph (MG)** Todo lugar tiene un arco entrante y uno saliente. Por tanto, no puede haber conflicto, pero puede haber concurrencia. Matemáticamente:
$$\forall p \in S : |p\bullet| = |\bullet p| = 1 \tag{C.2}$$

3. **Elección Libre – Free choice (FC)** Un arco es el único arco saliente del lugar, o bien el único arco entrante a una transición. Por tanto, puede haber tanto concurrencia como conflicto, pero no ambos simultáneamente. Matemáticamente:
$$\forall p \in P : (|p\bullet| \leq 1) \vee (\bullet(p\bullet) = \{p\}) \tag{C.3}$$

4. **Extended free choice (EFC)** Se trata de un red de Petri FC extendida, que puede transformarse en un FC. Todos los lugares que comparten las transiciones de salida deben tener las mimas transiciones de salida. Matemáticamente:
$$\forall p_1, p_2 \in P : p_1\bullet \cap p_2\bullet \neq 0 \rightarrow p_1\bullet = p_2\bullet \tag{C.4}$$

[4] Las redes de Petri AC y MAC pueden considerarse como dos subtipos dentro de uno mismo, i. e. redes de Petri AC (simples o múltiples). En tal caso el número de tipos de redes de Petri principales sería uno menos.

[5] Existen diversas taxonomías de redes de Petri, por lo que el número de tipologías puede variar. De hecho, en [Duarte Oliveira, 2003] se comentan algunos tipos de redes de Petri no comentados por considerarse fuera de los principales. También pueden diferir ligeramente los nombres de las tipologías entre distintas fuentes bibliográficas —se han adoptado los nombres y la terminología más común.

5. **Elección Asimétrica – Asymmetric Choice (AC)** Hay concurrencia y conflicto —en resumen, confusión—, pero no asimétricamente. Matemáticamente:

$$\forall p_1, p_2 \in P : (p_1 \bullet \cap p_2 \bullet \neq 0) \rightarrow [(p_1 \bullet \subseteq p_2 \bullet) \vee (p_2 \bullet \subseteq p_1 \bullet)] \tag{C.5}$$

6. **Múltiple Elección Asimétrica – Multiple Asymmetric Choice (MAC)** Hay múltiples concurrencias y conflictos —en resumen, hay múltiple confusión. Matemáticamente:
   Para un conjunto $|P| = k, \forall t \in T$ existe un subconjunto $\bullet t$, y $\bullet T$ contiene todos los subconjuntos y es el Conjunto Potencia 2^k sin el subconjunto vacío.

7. **Red de Petri – Petri Net (PN)** Se permite *confusión* —i. e. se permite todo.

La taxonomía de redes de Petri diferencia unas tipologías de redes que son conjuntos no disjuntos, donde unos —más generales— contienen a otros, tal y como muestra la Figura C.2.

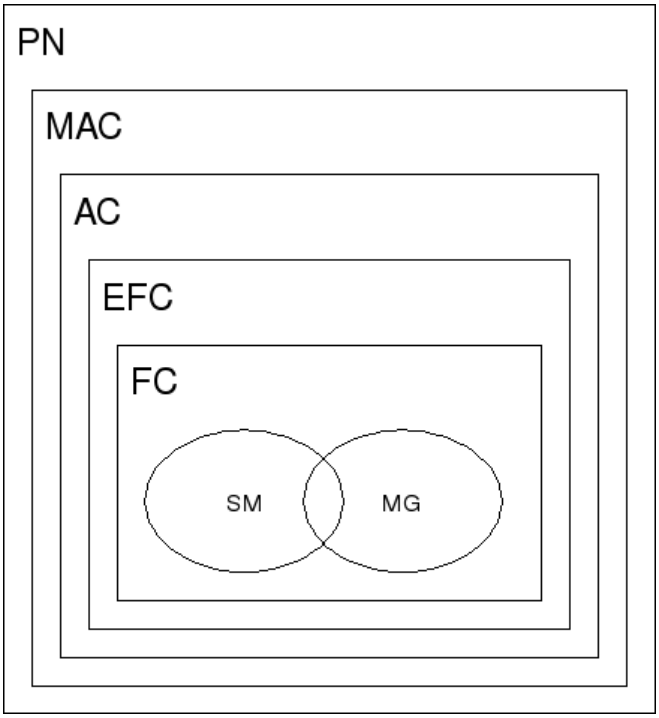


Figura C.2: Taxonomía de redes de Petri. Distribución de conjuntos

C.2.1. Red de Petri Generalizada

De acuerdo con [Duarte Oliveira, 2003] se pueden distinguir entre redes de Petri ordinarias y generalizadas, y éstas se definen tal y como se muestra en la Definición C.3 y Definición C.4, respectivamente.

Definición C.3 (RdP ordinaria). *Una RdP ordinaria sigue la definición indicada en la Definición C.2 y, por tanto, el peso de los arcos definido por w no puede tener un valor superior a 1, i. e. $w \in \{0, 1\}$.*

Definición C.4 (RdP generalizada). *Una RdP generalizada —también denominada RdP lugar-transición— se caracteriza por permitir la existencia de más de un arco entre un lugar y una transición o viceversa. Este tipo de redes tiene el mismo poder de modelado que una RdP ordinaria (véase la Definición C.3); en la representación gráfica es habitual mantener un único arco indicando el peso de éste.*

C.2.2. Red de Petri Modular

De acuerdo con [Duarte Oliveira, 2003] la utilización de RdP de alto nivel permite modelar sistemas con complejidad cada vez mayor. Por ello, una técnica de modelado modular presenta ventajas tales como poder modelar solamente algunas partes del proceso, por ser independientes unas de otras; esto reduce la complejidad del análisis del modelo global. Esta cualidad —modularidad— de las RdP, las hace especialmente útiles para modelar la misión y el sistema de un AUV. Son una herramienta interesante y candidata para su incorporación como arquitectura de la misión y el sistema, por lo que se analiza su viabilidad en la Capítulo 12 y Capítulo 14.

En la Definición C.5 se define en qué consiste una RdP modular, tal y como se comenta en [Duarte Oliveira, 2003], así como la equivalencia entre una RdP modular y una RdP —no modular— enunciada en la Definición C.6.

Definición C.5 (RdP modular). *Una RdP modular es una tripleta de la forma $RdPM = (S, PF, TF)$, la cual satisface los siguientes requisitos:*

1. *S es un conjunto finito de módulos tales que:*
   a) *Cada módulo $s \in S$ es una RdP $s = (P_s, T_s, A_s, w_s, \mu_s)$.*
   b) *Los conjuntos de nodos correspondientes a diferentes módulos son disjuntos a pares:*
   $$\forall s_1, s_2 \in S : \lfloor s_1 \neq s_2 \Rightarrow (P_{s_1} \cup T_{s_2}) \cap (P_{s_2} \cup T_{s_1}) = 0 \rfloor \tag{C.6}$$
2. *$PF \subseteq 2^P$ es un conjunto finito de conjuntos de lugares de fusión tales que:*
   a) *$P = \bigcup_{s \in S} P_s$ es el conjunto de todos los lugares de todos los módulos.*
   b) *Para los nodos $x \in P \cup T$ se usa la notación $S(x)$ para expresar el módulo al cual pertenece x. Para todos los lugares p contenidos en el conjunto P se define $\mu_0(p) = \mu_{0_{s(p)}}(p)$.*
   c) *Los miembros de un conjunto de lugares de fusión pf poseen las mismas marcaciones iniciales:*
   $$\forall pf \in PF : \forall p_1, p_2 \in pf : [\mu_0(p_1) = \mu_0(p_2)] \tag{C.7}$$
3. *$TF \subseteq 2^T$ es el conjunto finito de conjuntos de transiciones de fusión donde $T = \bigcup_{s \in S} T_s$ es el conjunto de todas las transiciones de todos los módulos.*

El Ítem 1 se refiere a que las redes modulares contienen un conjunto finito de módulos, siendo cada uno de ellos una RdP. El Ítem 2 indica que cada conjunto de lugares de fusión pf representa un conjunto de lugares fundidos en uno sólo. 2^P expresa el conjunto de todos los subconjuntos de lugares, aunque todos los elementos pertenecientes a un conjunto de lugares de fusión poseen la mismas marcación inicial. En el Ítem 3 cada conjunto de transiciones fundidas representa el conjunto total de transiciones fundidas.

Definición C.6 (Equivalencia entre RdP modular y RdP). *Considérese una RdP modular* $RdPM = (S, PF, TF)$. *Se define una RdP equivalente como* $RdP' = (P', T', A', w', \mu'_0)$ *donde:*

1. $P' = PG$
2. $T' = TG$
3. $A' \subseteq (P' \times T') \cup (T' \times P')$
4. $\forall (x', y') \in (P' \times T') \cup (T' \times P') : w'(x', y') = w(x', y')$
5. $\forall p' \in P' : \mu'_0(p') = \mu_0(p')$

En base a la Definición C.6 se muestra el ejemplo de [Duarte Oliveira, 2003], donde a partir de la RdP de la Figura C.3 se construyen las redes de Petri modulares equivalentes de la Figura C.4 (a) (b) —se trata de dos redes de Petri modulares diferentes, pero igualmente válidas y equivalentes a la RdP no modular original.

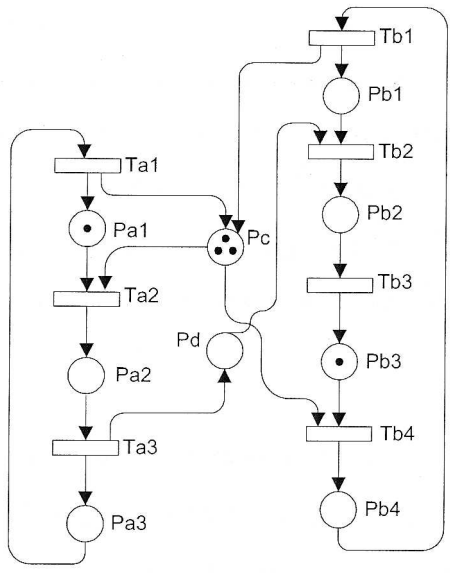
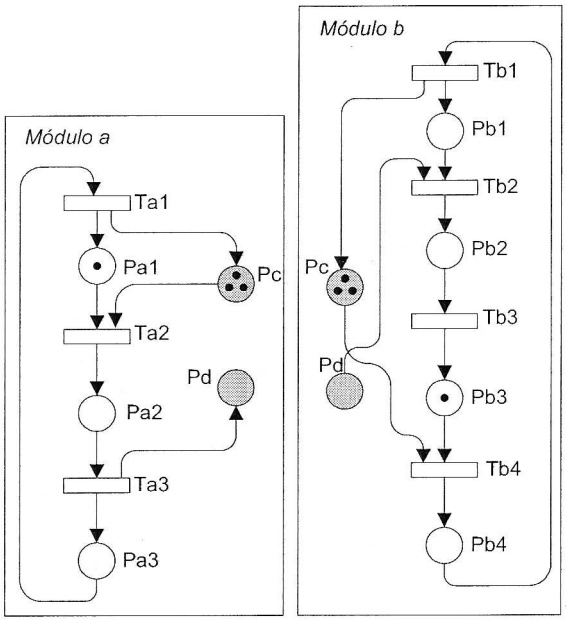
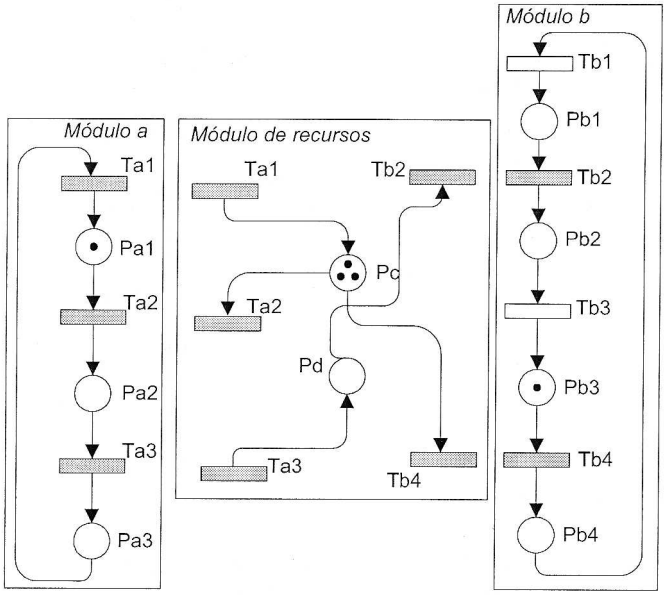


Figura C.3: RdP no modular que modela un ejemplo típico de asignación de recursos, representados por los lugares Pc y Pd



(a) Modelado de la RdP de la Figura C.3 con dos módulos y dos conjuntos de lugares de fusión con dos elementos cada uno



(b) Modelado de la RdP de la Figura C.3 con tres módulos y cinco conjuntos de transiciones de fusión con dos elementos cada uno

Figura C.4: RdP modular en la que se modela la RdP de la Figura C.3 con varios módulos

C.3. Áreas de Aplicación

A partir de lo comentado en [Usuarios Wikipedia, 2008b, Usuarios Wikipedia, 2008c, Petri and Reisig, 2008, Duarte Oliveira, 2003] es posible establecer el marco de aplicabilidad de las redes de Petri. Las características más importantes de éstas son:

1. Permiten el modelado de sistemas de eventos discretos asíncronos y concurrentes, donde la ocurrencia de diversos eventos está permitida de acuerdo a un conjunto de reglas predefinidas.

2. Constituyen una herramienta de modelado de un sistema a través de diferentes planos jerárquicos. Esta característica es particularmente interesante cuando se pretende descomponer o modularizar sistemas complejos —este es el caso de un AUV y en el caso de [Duarte Oliveira, 2003] con el entorno CORAL[6] y el submarino MARIUS se tiene un claro ejemplo de aplicación de las redes de Petri como caso práctico (véase la Sección 14.1).

3. La teoría en que se asientan dispone de herramientas de análisis de comprobada fiabilidad, tales como métodos basados en propiedades estructurales y comportamientos de la red, los cuales predicen el comportamiento de la misma permitiendo la detección de posibles anomalías y previniendo de este modo las posibles situaciones de error.

Dicho esto, las áreas de aplicación más comunes son:

Diseño software El caso de las redes de Petri modulares es muy importante en el diseño de software con redes de Petri para sistemas complejos. Esto se consigue fácilmente porque la posibilidad de modularizar una RdP facilita la jerarquización del sistema. Además, esto repercute en una mayor comprensión del sistema al poder manejar diferentes niveles de abstracción.

Gestión de flujo En aplicaciones controladas por el flujo de datos las redes de Petri resultan una buena herramienta de modelado, ya que permiten definir gráficamente el control de flujo de datos o recursos —según éstos se van procesando.

Análisis de datos Al igual que en la gestión de flujo, en aplicaciones de análisis de datos las redes de Petri son un herramienta de modelado apropiada.

Programación concurrente Las redes de Petri son especialmente indicadas para el diseño e implementación de sistemas de eventos discretos asíncronos y concurrentes. Como herramienta de modelado se ajusta perfectamente al diseño de estos sistemas y la teoría que las sustenta favorece el análisis de la red resultante y la detección temprana de posibles errores.

Sistemas robustos Los sistemas que deben ser robustos o confiables pueden beneficiarse de la cualidad de verificación formal de la red para garantizar su fiabilidad en entornos especialmente críticos, como es precisamente el caso de un AUV.

Diagnosis El formalismo matemático aplicable a las redes de Petri permite su verificación *a priori* y el fácil diagnóstico de errores, si los hubiere.

[6] Motor de interpretación de misiones para AUVs especificadas con redes de Petri (véase la Sección 14.1.5).

En la Sección 14.1 se estudian arquitecturas para diseñar el sistema de un AUV, entre las que se incluyen varias que hacen uso de redes de Petri —con distintos *sabores*— (véase la Sección 14.1.5, entre otras dentro de la Sección 14.1). Sin embargo, como se ha comentado previamente en esta sección, las áreas de aplicación de las redes de Petri no se reducen a los AUVs —que en mayor o menor medida formarían parte de todas las áreas de aplicación antes descritas[7]—, sino que también son útiles en la gestión de flujo de datos con concurrencia como en el caso de la Figura C.5 donde se controla una reacción química a partir de los reactivos para generar los productos mediante una RdP.

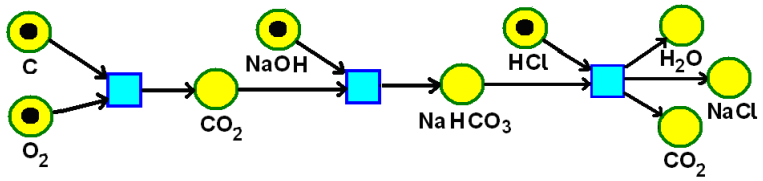


Figura C.5: Ejemplo RdP para reacción química

C.4. Software. Herramientas de construcción y ejecución

Existen diversas herramientas para trabajar con RdP, muchas de las cuales están diseñadas con un enfoque en la programación de sistemas robóticos móviles. De hecho [Ramalho Oliveira et al., 1996, Ramalho Oliveira et al., 1998] es un caso representativo de ello, donde estas herramientas se usan con vehículos de exploración submarina, concretamente AUVs.

C.4.1. Construcción de RdP. Diagramas

Para la construcción de RdP se emplean herramientas para construir diagramas de las mismas. Por lo general pueden aprovecharse aplicaciones de ámbito general. Sin embargo, los componentes primitivos necesarios para la construicción de RdP para AUVs no estarán disponible. En cualquier caso, el manejo general de estas aplicaciones sería el mismo. Se pueden consultar algunas de estas herramientas en [Usuarios Wikipedia, 2008b, Usuarios Wikipedia, 2008c, Petri and Reisig, 2008].

C.4.2. Compilación de RdP. Representación para ejecución

En este sentido sólo cabe destacar el uso de lenguajes de representación que facilita el diseño del compilador que ejecutará las RdP. En [Ramalho Oliveira et al., 1996], que usa el motor de interpretación CORAL, se tiene como ejemplo el uso de Lisp.

[7] El sistema del AUV que se pretende realizar en este proyecto (véase la Apéndice A y Parte III) es fundamentalmente un sistema robusto de eventos discretos asíncronos y concurrentes —i. e. área de aplicación de programación concurrente y sistemas robustos—, pero que también demanda en menor medida el restos de aspectos donde son aplicables las redes de Petri; también demanda otros aspectos que éstas no cubren con igual solvencia (véase el estudio de la Capítulo 12 y Capítulo 14).

C.4.3. Ejecución de RdP. Motor de interpretación

Los motores de RdP estudiados han sido CORAL y ProCoSa. Se remite al lector a las siguientes referencias bibliográficas [Ramalho Oliveira et al., 1996, Barrouil and Lemaire, 1998], respectivamente.

Apéndice D

Herramientas de Desarrollo

D.1. Autotools

Las herramientas **autotools** permite la construcción automática de objetivos de compilación. La documentación oficial se encuentra disponible en la siguiente dirección web: http://sources.redhat.com/autobook

D.2. Entornos de Desarrollo. Herramientas de Modelado y Entornos de Programación

Para el desarrollo de un sistema empotrado es especialmente importante el lenguaje de programación a usar y los entornos de desarrollo. Del mismo modo, para un proyecto de gran embargadura siempre es imprescindible aplicar los paradigmas de la Ingeniería del Software, para conseguir los principales beneficios de su aplicación en los procesos de desarrollo. Esto determina el uso de herramientos de modelado para el diseño de la aplicación, así como el uso de entornos de programación integrados[1] facilitará la fase de codificación o programación, más aún si se integran con las herramientas de diseño en el ciclo de vida del software.

Tanto en el diseño como en la codificación de la aplicación será interesante la aplicación de un paradigma de Orientación a Objetos (OO). Éste facilitará la reutilización y mantenimiento del código, a parte de que facilita el desarrollo, al cargar de semántica al mismo. El modelado con UML será el lenguaje de diagramación apropiado para las representaciones del diseño. El diagrama de clases en UML está directamente relacionado con las clases que se implementarán, en nuestro caso en el lenguaje C++, que está orientado a objetos. Por ello, la forma de conseguir una perfecta intgración entre las dos fases de desarrollo pasa por disponer de una herramienta de modelado que permita la generación de código, así como la ingeniería inversa, es decir, obtener la representación de la implementación en código C++, como diagrama de clases.

En la sección D.2.1 se comentan las herramientas de modelado más apropiadas, con sus ventajas e inconvenientes, no sólo desde el punto de vista del modelado, sino su

[1] Se trata de lo que se denomina como *IDE* (*Integrated Development Enviroment*), que en español se traduce como *Entorno de Desarrollo Integrado*.

disponibilidad en diferentes plataformas. Por otro lado, en la sección D.2.2 se comenta por qué el lenguaje C++ es el más apropiado para la estructura general de la aplicación y cómo es posible que para determinados aspectos de la misma se empleen otros lenguajes de programación o librerías y utilidades.

D.2.1. Diseño Orientado a Objetos mediante Herramientas de Modelado UML

A continuación se indican las posibles herramientas de modelado que se pueden usar para la creación de diagramas UML. Se muestran las más destacadas y se comentan sus ventajas e inconvenientes.

Umbrello

MagicDraw UML

Modelado integrado en Eclipse

D.2.2. IDEs (Entornos de Desarrollo Integrados)

Se ha empleado C++ como lenguaje de desarrollo, utilizando **KDevelop** como Entorno de Desarrollo Integrado (IDE). Además, se ha hecho uso de múltiples librerías y utilidades del sistema. Reflejo de ello es la lista de recursos mostrada en la Sección 4.1.1.

Se ha usado C++ porque ofrece mayor flexibilidad que cualquier otro lenguaje para la integración de diferentes librerías y facilita la interacción con dispositivos físicos. Este es el caso de la realización de *drivers*, así como la adaptación que se ha hecho de algunos de ellos a *Player*.

Apéndice E

Detalles técnicos sobre la Implementación del proyecto

E.1. CoolBOT. A Component-Oriented Programming Framework for Robotics

. . . a software framework called CoolBOT which permits to program robotic systems by integrating software components.
CoolBOT allows to program robotic systems as if they were networks of interconnected sofware components.

— Antonio Carlos Domínguez Brito
2003 (Departamento de Informática y Sistemas. Universidad de Las Palmas de Gran Canaria)

E.1.1. Definición

CoolBOTes un marco de programación orientada a componentes para robótica [Domínguez Brito, 2003]. Se remite al lector a la bibliografía para una consulta detallada del mismo.

E.1.2. Especificación de componentes CoolBOT

Para la especificación de componentes CoolBOT no se ha usado directamente C++, ya que los ficheros cabecera e implementación del mismo son excesivamente largos. En su lugar se ha creado un generador de esqueletos de componentes CoolBOT. A partir de una definición del componente CoolBOT en lenguaje XML se obtiene como resultado los ficheros cabecera e implementación, listos para introducir la lógica de control que sea necesaria.

E.1.3. Generación de componentes CoolBOT

El generador de componentes CoolBOT se ha desarrollado en Perl, utilizando *trozos* de un molde de un componente CoolBOT representativo. A partir de ello y con la informaación de la especificación del componente se generan los ficheros cabecera e implementación en C++.

Hay que indicar que esta aplicación es en modo texto y permite varios parámetros de configuración. Esto permite la generación de los componentes de forma adaptada al sistema *Sick***AUV**. También es posible indicar opciones de depuración y modificadores sobre cómo debe ser el resultado generado.

E.1.3.1. Otros generadores

Al inicio de este Proyecto Fin de Carrera (PFC) no se disponía de un generador de esqueletos de componentes CoolBOT. Sin embargo se comenzó el desarrollo de un PFC con esta finalidad (véase el Apéndice A).

E.2. Player

E.2.1. Arquitectura de *Player*

All the world's a stage,
And all the men and women merely players.

— William Shakespeare, As You Like It
1599/1600 (Epígrafe del proyecto Player)

El *framework* de *Player* [Gerkey et al., 2006] dispone de una arquitectura software en su diseño e implementación. Es interesante conocerla para incorporar nuevos robots o para dar soporte a las interfaces, mediante nuevos *drivers*. Sin embargo, éstas no son las únicas posibilidades. En principio, identificamos las siguientes posibilidades de extensión o interacción con el software de *Player* —que serán útiles o de interés desde el punto de vista del desarrollo del sistema del AUV.

1. *Adaptar la implementación de un* driver *existente para un determinado dispositivo* → La creación o implementación del *driver* como tal se entiende como una tarea al margen de *Player*. Por ello hay que hacer hincapié en que se trata de una labor de adaptación o integración a *Player*. Esto consiste básicamente en dar soporte a una de las interfaces de *Player* —i.e. a la que mejor se ajuste el dispositivo en cuestión. Pero desde el punto de vista de la adaptación del *driver* a *Player* hacemos referencia a realizar la implementación del *driver* —o adaptarla— según la clase Driver que proporciona *Player* y que define cómo debe implementarse de forme genérica todo *driver*.

2. *Dar soporte a una interfaz de* Player → Aunque muy relacionada con la adaptación de un *driver* a *Player*, la labor de dar soporte a una de las interfaces de *Player* se centra en analizar y estudiar las características propias de éstas. Hay bastante

heterogeneidad entre las interfaces, de modo que se requiere un buen conocimiento de aquella interfaz a la que se vaya a dar soporte. Tras la fase inicial de estudio se tendrá que proceder a dar soporte a la interfaz haciendo que se proporcionen todos los datos necesarios interactuando con el *driver* del dispositivo —adaptado a *Player*.

3. *Crear una aplicación de un robot o cliente de* Player → Esta aplicación usará dispositivos que desde este lado del servidor de *Player* de implementan como un *proxy* cada uno. El tipo de aplicaciones que se pueden crear dependen de la finalidad que desee el programador —e.g. un robot, una interfaz gráfica para la visualización de los datos sensoriales, como para los datos de un GPS. Cuando se dé quiera crear este tipo de aplicaciones hay que conocer la API del *proxy* de la interfaz con la cual se va a interactuar. Esta información está disponible desde la documentación de *Player*, en las librerías de los clientes —i.e. *Client Libraries*. Existen tres implementaciones para diferentes lenguajes: C, Python y C++. Nosotros nos centraremos en las desarrolladas en C++, cuya documentación está en el siguiente enlace:

   http://playerstage.sourceforge.net/doc/Player-2.0.0/player/group__player__clientlib__cplusplus.html

4. *Crear una nueva interfaz de* Player → Esta es la tarea más complicada o tediosa. Requiere de la modificación de múltiples ficheros dentro del árbol de directorios del código fuente de la distribución de *Player*. Se tendrán que realizar las siguientes subtareas:

   a) Definir el protocolo de comunicación de la mensaje —i.e. los comandos o mensajes que se enviarán y las estructuras de datos que se manejarán. Esto define la comunicación entre el servidor de *Player* y los *drivers*. Desde el lado del *driver* se rellenará la estructura de datos para comunicar los datos obtenidos del dispositivo, mientras que desde el lado del servidor de *Player* se usará dicha estructura de datos para facilitar la información del dispositivo a través de la API del *proxy* que se proporciona para el desarrollo de aplicaciones en *Player*.

   b) Crear un *driver* de *Player*, que básicamente es la adaptación de un *driver* genérico existente a la nueva interfaz de *Player* creada.

   c) Crear un *proxy* de *Player*. Con ello se define la interfaz a usar en las aplicaciones que se desarrollen en *Player* —i.e. robots u otro tipo de aplicaciones.

   d) Actualizar los ficheros del código fuente para la correcta creación de los ejecutables y librerías, de modo que la interfaz funcione perfectamente.

E.2.2. Estructura de directorios del código fuente

Los principales directorios del código fuente de *Player* se indican a continuación —se considera la versión 2.0.3.

client_libs/ Código fuente de los *proxies*, i.e. las clases herederas de la clase base Proxy, que implementan las interfaces para los diferentes dispositivos soportados por *Player*. Estos *proxies* determinan las API que se usará en el desarrollo de aplicaciones en *Player* —i.e. proporcionan una interfaz apropiada para acceder a

las estructuras de datos recibidas en el proceso de comunicación entre el servidor y los *drivers* de *Player*, así como para enviar comandos, etc. En la carpeta `libplayerc++/` se dispone de la implementación en C++, que será la que se usará en el desarrollo del sistema del AUV. No obstante, en muchas ocaciones se trata de *wrappers* de las implementaciones en C, contenidas en la carpeta `libplayerc/`.

config/ Ficheros de configuración para lanzar el servidor de *Player*. Sirven de ejemplo para lanzar éste con distintos conjuntos de dispositivos, i.e. *drivers*. Como *drivers* se pueden incorparar *drivers* propiamente dichos que controlan dispositivos —físicos o emulados en software—, aplicaciones —robots— y simuladores —e.g. *Stage*—, fundamentalmente.

doc/ Documentación de la distribución del código fuente de *Player*. En cualquier caso, siempre es de obligatoria consulta la documentación de *Player* disponible dentro de su web.

`http://playerstage.sourceforge.net`

examples/ Ejemplos de programas de clientes de *Player* y de *plugin drivers* (ver sección **??**). Se encuentran ubicados en diferentes carpetas:

libplayerc++/ Ejemplos de programas clientes de *Player* desarrollados en C++; en la carpeta `libplayerc/` se tienen versiones equivalentes en C, pero en el sistema del AUV se usará la implementación en C++. Se programan usando el *proxy* de la interfaz de cada dispositivo que se use en la aplicación. La documentación de cada *proxy* —que hereda de la clase base `Proxy`— se encuentra en las librerías de clientes previamente referenciadas.

plugin/ Ejemplos de *plugin drivers* de distintos tipos paradigmáticos:

exampledriver/ Ejemplo de un *driver* simple —en concreto da soporte a la interfaz `position2d` de *Player*.

multidriver/ Ejemplo de un *driver* que da soporte a más de una interfaz de *Player* —en concreto a `position2d` y `laser`. Se usa como ejemplo para documentar la incorparación de este tipo *drivers* en la sección E.2.14.1.

opaquedriver/ Ejemplo de un *driver* que da soporte a la interfaz `opaque` de *Player*. Esta interfaz permite crear *drivers* genéricos no contemplados en *Player*, propocionando mecanismos de comunicación genéricos con el servidor de *Player*; en la sección E.2.7 se comenta cómo dar soporte a esta interfaz y en la sección E.2.14.2 los detalles de implementación para adaptar un *driver*.

libplayercore/ Código fuente del núcleo del servidor, las interfaces y *drivers* de *Player*.

libplayerjpeg/ Código fuente para el manejo de imágenes en formato JPEG, usadas para algunas partes de *Player* —e.g. los escenarios del simulador *Stage*.

libplayertcp/ Código fuente para las comunicación sobre *sockets* TCP/IP entre los *drivers* y el servidor de *Player* —i.e. envía los mensajes entre las colas de *Player* y los *sockets* TCP/IP.

libplayerxdr/ Código fuente para convertir las estructuras de datos C/C++ que se transmitirán, a la representación XDR[1] [Group, 1987] equivalente.

replace/ Utilidades básicas para la implementación de *Player*.

rtk2/ Librerías gráficas —*GUI toolkit*— para robótica.

server/ Código fuente de los *drivers* estáticos (ver sección **??**) implementados en la distribución oficial de *Player*; se encuentran en la carpteta `drivers/`. Sirven de ejemplo para el desarrollo de nuevos *drivers* —e.g. distintos tipos de *drivers*, que dan soporte a diferentes interfaces de *Player*, etc. En la carpeta `mixed/` —dentro de la carpeta `drivers/`— se dispone de los *drivers* que dan soporte a más de una interfaz —e.g. el *driver* de *p2os* (*Pioneer 2 Operative System*, dentro de la subcarpeta `p2os/`.

utils/ Utilidades adicionales para *Player* —e.g. servidor DGPS para aplicar correcciones a los datos de un GPS, clientes gráficos de *Player* como *playerv*, etc.

Desde el punto de vista de las posibilidades de extensión o interacción con el software de *Player*, mencionadas en la sección **??**, para cada una de ellas serán de interés los siguientes directorios por los motivos que se comentan.

1. *Adaptar la implementación de un* driver *existente para un determinado dispositivo* → Se podrán usar los ejemplos del directorio `server/drivers/` para adaptar un nuevo *driver* a *Player*, dando soporte a una de sus interfaces. Del mismo modo, en la carpeta `examples/plugin/` se dispone de ejemplo para el desarrollo de *plugin drivers* (ver sección **??**). En el caso de realizar un *driver* estático, se tendrá que integrar dentro del código fuente de *Player* —i.e. en la carpeta `server/drivers/`, junto con el resto de *drivers* ya existentes. Por contra, los *plugin drivers* no tienen que integrarse en el código fuente de la distribución de *Player*, lo que facilita su desarrollo y uso.

2. *Dar soporte a una interfaz de* Player → A parte de conocer cómo adaptar la implementación de un *driver* —ya que con éstos se da soporte a las interfaces—, se tiene que conocer a qué mensajes responder y qué estructuras de datos enviar al servidor de *Player*, desde el *driver*. Aunque es absolutamente necesario consultar la documentación (ver sección E.2.3), los ejemplos de la carpeta `server/drivers/` son ilustrativos. Según se realice un *driver* estático o un *plugin driver*, será necesario integrar la implementación en el código fuente de la distribución de *Player* o no, respectivamente.

3. *Crear una aplicación de un robot o cliente de* Player → Se debe consultar la documentación (ver sección E.2.3), aunque también puede verse la implementación del *proxy* de la interfaz que se vaya a usar en la aplicación. Ésta se encontrará en la carpeta `client_libs/libplayerc++/`.

4. *Crear una nueva interfaz de* Player → Para crear una interfaz será siemper necesario modificar el código fuente de la distribución de *Player*. Se tendrá que definir el conjunto de mensajes entre el servidor y el *driver* y las estructuras de datos a emplear. Por tanto, se tendrá que crear al menos un *driver* de ejemplo —en la

[1] XDR (eXternal Data Representation) es un estándar para la descripción y codificación de datos independiente del tamaño de la palabra, el orden de los bytes u otros detalles de una arquitectura particular.

carpeta `server/drivers/`— y el *proxy* —en la carpeta `client_libs/libplayerc++/`—, para que se pueda usar la interfaz desde las aplicaciones o robots. Además, se tendrán que actualizar los ficheros de construcción de la distribución de *Player*, para que la nueva interfaz se compile correctamente.

E.2.3. Documentación

Dentro de la web oficial de *Player* encontramos la documentación oficial en el siguiente enlace:

`http://playerstage.sourceforge.net/doc/Player-2.0.0/player/`

Desde el menú de la izquierda se puede acceder a la distinta información. No obstante, es especialmente importante la siguiente, para el desarrollo del sistema del AUV —i.e. para la integración e implementación del equipamiento del AUV.

Interfaces de *Player* Se trata de la lista de interfaces disponibles en *Player*, desde el punto de vista de los *drivers*. Cada una dispone de una documentación resumida en la que se indican los mensajes entre el servidor y el *driver*, así como las estructuras de datos que se manejan.

http://playerstage.sourceforge.net/doc/Player-2.0.0/player/group__interfaces.html

Proxies La documentación de cada uno de los *proxies* de las interfaces de *Player* forma parte de la librerías de los clientes. Permiten crear aplicaciones o robots con *Player* en un determinado lenguaje de programación —i.e. C, Python o C++. En el siguiente enlace se muestra la documentación de los *proxies* para el desarrollo de aplicaciones en C++ —que es la alternativa empleada en el sistema del AUV.

http://playerstage.sourceforge.net/doc/Player-2.0.0/player/group__player__clientlib__cplusplus__proxies.html

Ficheros de Configuración Explicación de la creación de ficheros de configuración. En éstos se indican los *drivers* que deben cargarse en el servidor de *Player*, lo que incluye aplicaciones/robots y simuladores.

http://playerstage.sourceforge.net/doc/Player-2.0.0/player/group__tutorial__config.html

Aplicación en *Player* Se trata de un pequeño ejemplo de aplicación cliente de *Player*. No sólo se muestra su código, sino que igualmente se explica el proceso de compilación.

http://playerstage.sourceforge.net/doc/Player-2.0.0/player/group__cplusplus__example.html

Para cada una de las posibilidades de extensión o interacción con el software de *Player* existe cierta documentación. Sin embargo, no toda esta información se encuentra en la web oficial del *framework* [Gerkey et al., 2006].

1. *Adaptar la implementación de un* driver *existente para un determinado dispositivo* → Actualmente no existe información sobre la adaptación o implementación de *drivers* en *Player*. Sin embargo, consultando los ejemplos de *drivers* de la carpeta `server/drivers/` y los ejemplos de creación de *plugin drivers* de la carpeta `examples/plugin/` se puede determinar cómo debe desarrollarse un *driver* dentro de *Player*, para dar soporte a una de sus interfaces. En la sección **??** se comenta todo esto, en lo que sería una fuente de documentación.

2. *Dar soporte a una interfaz de* Player → La información sobre las interfaces de *Player*, desde el punto de vista de la adaptación o implementación de *drivers* en *Player*, requiere la consulta de la documentación del siguiente enlace:

   http://playerstage.sourceforge.net/doc/Player-2.0.0/player/group__interfaces.html

   Adicionalmente debe consultarse la forma en que debe integrarse el *driver*, lo cual se ha indicado en el punto anterior.

3. *Crear una aplicación de un robot o cliente de* Player → Cuando se crea un aplicación en *Player* se debe conocer la estructura básica del código de la misma y cómo compilarla. Esto está documentado en el siguiente enlace:

   http://playerstage.sourceforge.net/doc/Player-2.0.0/player/group__cplusplus__example.html

   También se tiene que consultar la documentación de las librerías de los clientes, donde se explica la API de los *proxies*, que se tienen para cada una de las interfaces de *Player*. Esto es necesario porque normalmente se usarán dispositivos en la aplicación —e.g. sensores, actuadores, dispositivos de comunicación, etc.

   http://playerstage.sourceforge.net/doc/Player-2.0.0/player/group__player__clientlib__cplusplus__proxies.html

4. *Crear una nueva interfaz de* Player → No se dispone de documentación oficial para la creación de una interfaz nueva en *Player*. No obstante, para ello se puede consultar la documentación de los *proxies* y los *drivers*, para disponer de ejemplos en la implementación de una interfaz. En la sección E.2.9 se explica más en detalle las tareas a realizar para disponer de una nueva interfaz en *Player*.

E.2.4. Soporte para *interfaces* de *Player*

En esta sección se explicarán aquellas interfaces de *Player* a las que se ha dado soporte en el sistema del AUV —adaptando o implementando *drivers* para las mismas. En el cuadro E.1 se muestra la lista completa de la interfaces soportadas por *Player*[2], pero sólo se explicarán las de la lista del cuadro E.2, donde se indican sólo aquéllas empleadas en el sistema del AUV —algunas para integrar más de un dispositivo (ver sección **??**).

Interfaz	Descripción
actarray	Vector de actuadores.
aio	Entrada/Salida —I/O— analógica.

[2]Se considera la versión 2.0.3 de *Player*, donde algunas interfaces están obsoletas y desaparecerán en próximas versiones.

Interfaz	Descripción
audio	Detección y emisión de tonos audibles (**obsoleta**).
audiodsp	Detección y emisión de tonos audibles.
audiomixer	Control de los niveles de audio.
blinkenlight	Luz intermitente —*blinking*.
blobfinder	Sistema visual de detección de formas —*blob-detection*.
bumper	Vector de sensores de tacto/parachoques —*bumpers*.
camera	Interpretación de imágenes de cámara.
dio	Entrada/Salida —I/O— digital.
energy	Almacenamiento y consumo energético (**obsoleta**).
fiducial	Fiducial (marker) detection.
gps	Sistema de Posicionamiento Global —*Global Positioning System* (GPS).
graphics2d	Interfaz gráfica en 2 dimensiones.
graphics3d	Interfaz gráfica en 3 dimensiones.
gripper	An actuated gripper.
ir	Vector de sensores de infrarrojos —*infrared rangers*.
joystick	Control de un *Joystick*.
laser	Laser range-finder.
limb	Brazo robótico (extremidad con múltiples articulaciones) —*multi-jointed limb*.
localize	Multi-hypothesis planar localization system.
log	Control de lectura/escritura en un registro —*log*.
map	Acceso a mapas.
mcom	Cliente de comunicaciones.
opaque	Interfaz genérica para mensajes definidos por el usuario.
planner	A planar path-planner.
player	Player: the meta-device.
position1d	A 1-D linear actuator.
position2d	Robot que se mueve en 2 dimensiones —i.e. sobre un plano.
position3d	Robot que se mueve en 3 dimensiones.
power	Sistema de energía.
ptz	Pan-tilt-zoom unit.
simulation	Simulador de un robot.
sonar	Array of ultrasonic rangers.
sound	Reproducción de secuencias de audio.
speech	Síntesis de voz.
speech_recognition	Reconocimiento de voz.
truth	Access to true state (**obsoleta**).
waveform	Digital waveforms.
wifi	WiFi signal information.
rfid	RFID reader.
wsn	Wireless Sensor Networks.

Cuadro E.1: Descripción de las interfaces de *Player*

Interfaz
actarray gps
opaque
position3d

Cuadro E.2: Lista de interfaces de *Player* usadas

A continuación se explica cada una de las interfaces usadas, listadas en el cuadro E.2. La documentación oficial de las interfaces —todas, no sólo las usadas— se encuentra en la web de *Player*, tal y como se comenta en la sección E.2.3. Para cada interfaz se indicará la fuente de documentación oficial de la misma y las de cada una de las estructuras de datos que se usan en la interfaz.

Toda interfaz contendrá la siguiente información:

Mensajes Los tipos de mensajes que intervendrán en los procesos de comunicación entre el *driver* y el servidor de *Player* o viceversa. Éstos se indican con macros de definición de C/C++, como muestra el algoritmo E.1 —con uno de los mensajes

de la interfaz *gps*. Estas macros permiten indicar qué tipo de mensaje se transmite o se recibe, permitiendo conocer qué estructura de datos asociada tiene.

```
#define PLAYER_GPS_DATA_STATE 1
```

Algoritmo E.1: Definición de un tipo de mensaje

Para cada interfaz se indicará de qué mensajes dispone, su finalidad —indicando qué estructura de datos se usa para el envío de los datos— y el sentido de la comunicación en que se usa.

Estructuras de datos Las estructuras de datos se definen como en el algoritmo E.2. Constarán de un conjunto de campos que se transmiten entre el *driver* y el servidor de *Player* o viceversa. Según el tipo de mensaje se usará una estructura de datos u otra.

```
typedef player_gps_data player_gps_data_t
```

Algoritmo E.2: Definición de una estructura de datos

Tanto para los mensajes como para las estructuras de datos se indicará el sentido en que se produce la comunicación. Es posible que en ciertos casos la comunicación se realice en ambos sentidos.

1. Driver *a Servidor* → El *driver* envía el mensaje o la estructura de datos —tras rellenarla con los datos obtenidos del dispositivo controlado— al servidor de *Player*.

2. *Servidor a* Driver → El *driver* recibe el mensaje o la estructura de datos —e.g. información de control o comandos— desde el servidor de *Player*. Esto podrá afectar en la configuración del dispositivo o cualquier otro tipo de acciones. La información de estas acciones se indicará en las estructuras de datos —cuando sea necesario y no baste con el mensaje.

De acuerdo con la sintaxis usada en la documentación de *Player*, los tres posibles flujos de comunicación son los siguientes:

1. Dato – *Data* → Comunicación Driver *a Servidor*. Se envía una estructura de datos —asociada—, con los datos obtenidos del dispositivo por el *driver*, al servidor de *Player*.

2. Comando – *Command* → Comunicación *Servidor a* Driver. El servidor de *Player* —por iniciativa de una aplicación o robot— envía un comando al *driver* —que puede tener una estructura de datos asociada con la información del comando.

3. Petición/Respuesta – *Request/Reply* → Comunicación Driver *a Servidor* y *Servidor a* Driver. El servidor envía la petición y el *driver* envía la respuesta posteriormente —normalmente con la información asociada en una estructura de datos.

Las estructuras de datos heredan el tipo de comunicación de los mensajes a los que están asociadas —o que tienen asociados.

Por simplicidad al nombrar los mensajes y las estructuras de datos, se indicará el prefijo y luego sólo el sufijo que irá cambiando. Esto es posible por el esquema de nombres usado en *Player*, en el que todos los mensajes y estructuras de datos de una misma

Mensaje (*PLAYER_GPS_*)	Tipo Comunicación	Descripción
DATA_STATE	Dato	Envío de los datos del GPS en la estructura de datos *player_gps_data*.

Cuadro E.3: Mensajes de la interfaz *gps*

Estructura de datos (*player_gps_*)	Mensaje asociado (*PLAYER_GPS_*)	Descripción
data	*DATA_STATE*	Datos del GPS, i.e. el posicionamiento global y la fecha.

Cuadro E.4: Estructuras de datos de la interfaz *gps*

interfaz tienen el mismo prefijo —e.g. para la interfaz *gps* los prefijos son PLAYER_GPS_ y player_gps_ para los mensajes y estructuras de datos, respectivamente.

Por otro lado, para cada interfaz también se mostrará la API del *proxy* que tiene asociado. Con ésta se programarán aplicaciones en *Player*. Se indican los métodos de la API del *proxy* de la interfaz, sin incluir el constructor, porque siempre será igual para todas los *proxies*, con un prototipado como el del algoritmo E.3 —con el ejemplo del *proxy* de la interfaz del *gps*, denominada *GpsProxy*.

```
GpsProxy(PlayerClient *aPc, uint aIndex = 0);
```

Algoritmo E.3: Prototipado del constructor de un *proxy*

E.2.5. Soporte para la interfaz *actarray*

E.2.5.1. *Proxy ActArrayProxy*

E.2.6. Soporte para la interfaz *gps*

La documentación de la interfaz *gps* se encuentra en:

http://playerstage.sourceforge.net/doc/Player-2.0.0/player/group__interface__gps.html

En el cuadro E.3 se describen los mensajes que maneja esta interfaz y en el cuadro E.4 se describen las estructuras de datos que se manejan —sin entrar en el detalle de sus campos.

La documentación de la estructura de datos player_gps_data se encuentra en:

http://playerstage.sourceforge.net/doc/Player-2.0.0/player/structplayer__gps__data.html

En el cuadro E.5 se describe cada uno de los campos de la estructura de datos player_gps_data, indicando su nombre y descripción. En la descripción se incluye la información sobre el tipo de unidades con que deben indicarse los datos. El cuadro E.6 complementa al E.5 mostrando un ejemplo de cumplimentación de las estructura de datos player_gps_data. En este ejemplo se indican los valores almacenados en los campos y su interpretación —con las unidades apropiadas.

A continuación se listan algunas aclaraciones importantes referentes a los campos de la estructura de datos player_gps_data, listados en el cuadro E.5.

Campo		Descripción
Tipo	Nombre	
uint32_t	time_sec	Fecha del GPS (UTC), en segundos desde la época de UNIX.
uint32_t	time_usec	Fecha del GPS (UTC), en microsegundos desde la época de UNIX.
int32_t	latitude	Latitud en grados/1e7 —i.e. se escala para que la resolución sea de 1cm. Un valor positivio indica que se está al norte del ecuador y uno negativo al sur.
int32_t	longitude	Longitud en grados/1e7 —i.e. se escala para que la resolución sea de 1cm. Un valor positivo indica que se está al este del meridiano 0 y uno negativo al oeste.
int32_t	altitud	Altitud en milímetros, respecto del punto de referencia —e.g. el nivel del mar. Un valor negativo indicaría que se está por debajo de éste.
double	utm_e	Valor de *easting* en centímetros, de las coordenadas UTM WGS84.
double	utm_n	Valor de *northing* en centímetros, de las coordenadas UTM WGS84.
uint32_t	quality	Calidad del ajuste —*fix*— del posicionamiento. 0 = no válido, 1 = ajuste GPS, 2 = ajuste DGPS.
uint32_t	num_sats	Número de satélites a la vista o accesibles.
uint32_t	hdop	*Horizontal Dilution Of Position* (HDOP) multiplicado por 10.
uint32_t	vdop	*Vertical Dilution Of Position* (VDOP) multiplicado por 10.
double	err_horz	Error horizontal, en metros.
double	err_vert	Error vertical, en metros.

Cuadro E.5: Campos de la estructura de datos *player_gps_data*

Campo	Valor	
	Almacenado	Interpretado
time_sec	1165707336	1165707336s (2006-12-09 23:35:36 UTC)
time_usec	142432000	142432000μs (142432ms)
latitude	281318629	28.1318629°N
longitude	-??	??°W
altitud	64299	64.299m
utm_e	??	??m
utm_n	??	??m
quality	1	Ajuste GPS
num_sats	5	5 satélites
hdop	??	??HDOP
vdop	??	??VDOP
err_horz	??	??m
err_vert	??	??m

Cuadro E.6: Valores de ejemplo para la estructura de datos *player_gps_data*

1. Los campos `time_sec` y `time_usec` representan conjuntamente la fecha del GPS. El primero proporciona los segundos y el segundo los microsegundos. La fecha que debe proporcionar la interfaz *gps* es la fecha UTC, que se representa como el número de segundos y microsegundos —almacenados en los dos campos antes mencionados— transcurridos desde la época de UNIX[3].

2. En cuanto al número de satélites a la vista (campo `num_sats`) se consideran todos los satélites accesibles. Este valor puede diferir del número de satélites empleados en el cómputo del ajuste del posicionamiento, que podrá ser igual o inferior.

E.2.6.1. *Proxy GpsProxy*

En el cuadro E.7 se muestra la API del *proxy* de la interfaz *gps* —i.e. *GpsProxy*. Proporciona los datos de la estructura de datos `player_gps_data` de la interfaz *gps*, cuyos campos se indican en el cuadro E.5. Sin embargo, las unidades difieren, tal y como se

[3]Por época —*epoch* en inglés— de UNIX entendemos la fecha de referencia empleada en los sistemas UNIX para almacenar la fecha. Se almacenan los segundos y microsegundos transcurridos desde la misma. Esta fecha de referencia es **1970-01-01 00:00:00 UTC** —i.e. el 1 de enero de 1970.

Método Tipo	Método Nombre	Descripción
double	GetLatitude()	Latitud en grados.
double	GetLongitude()	Longitud en grados.
double	GetAltitude()	Altitud en metros.
uint	GetSatellites()	Número de satélites a la vista.
uint	GetQuality()	Calidad del ajuste.
double	GetHdop()	*Horizontal Dilution Of Position* (HDOP).
double	GetVdop()	*Vertical Dilution Of Position* (VDOP).
double	GetUtmEasting()	*Easting* en metros, de las coordenadas UTM WGS84.
double	GetUtmNorthing()	*Northing* en metros, de las coordenadas UTM WGS84.
double	GetTime()	Fecha desde la época de UNIX.
double	GetErrHorizontal()	Error horizontal, en metros.
double	GetErrVertical()	Error vertical, en metros.

Cuadro E.7: API de *GpsProxy*

Mensaje (*PLAYER_OPAQUE_*)	Tipo Comunicación	Descripción
DATA_STATE	Dato	Envío de los datos opacos/genéricos en la estructura de datos *player_opaque_data*.
CMD	Dato	Comando opaco/genérico.
REQ	Dato	Petición opaca/genérica.

Cuadro E.8: Mensajes de la interfaz *opaque*

indica en la descripción de los diferentes métodos de la API del *GpsProxy* del cuadro E.7. Hay que hacer hincapié en las siguientes aclaraciones:

1. El método `GetTime()` devuelve los segundos y microsegundos en una misma variable de tipo `double`, mientras que la estructura de datos `player_gps_data` almacenaba los segundos y microsegundos por separado en sendas variables de tipo `uint32_t`, i.e. en `time_sec` y `time_usec`, respectivamente. Para realizar la conversión —o fusión— se aplica el algoritmo E.4 —obtenido del fichero `client_libs/libplayerc/dev_gps.c` de la distribución de *Player* (ver sección E.2.2)—, almacenando los segundos y milisegundos —se desprecian los valores inferiores a los milisegundos del campo `time_usec` de la estructura de datos `player_gps_data`.

```
1 device->utc_time = gps_data->time_sec + ((double)gps_data->time_usec)/1e6;
```

Algoritmo E.4: Fusión de segundos y microsegundos

E.2.7. Soporte para la interfaz *opaque*

La documentación de la interfaz *opaque* se encuentra en:

http://playerstage.sourceforge.net/doc/Player-2.0.0/player/group__interface__opaque.html

En el cuadro E.8 se describen los mensajes que maneja esta interfaz y en el cuadro E.9 se describen las estructuras de datos que se manejan —sin entrar en el detalle de sus campos.

También se define una macro para indicar el tamaño máximo de un mensaje, fijado en 1MB, como muestra el algoritmo E.5.

```
#define PLAYER_OPAQUE_MAX_SIZE 1024
```

Algoritmo E.5: Tamaño máximo de un mensaje opaco/genérico

Estructura de datos (*player_opaque_*)	Mensaje asociado (*PLAYER_OPAQUE_*)	Descripción
data	*DATA_STATE*	Datos opacos.

Cuadro E.9: Estructuras de datos de la interfaz *opaque*

Campo		Descripción
Tipo	Nombre	
uint32_t	data_count	Tamaño de los datos a enviar.
uint8_t	data[PLAYER_OPAQUE_MAX_SIZE]	Datos a enviar.

Cuadro E.10: Campos de la estructura de datos *player_opaque_data*

La documentación de la estructura de datos `player_opaque_data` se encuentra en:

http://playerstage.sourceforge.net/doc/Player-2.0.0/player/structplayer__opaque__data.html

En el cuadro E.10 se describen los campos de la estructura de datos `player_opaque_data`. En este caso los campos son genéricos —u opacos—, por lo que la forma en que se manejan requiere una explicación especial con vistas a la implementación. Ésta afecta especialmente a la forma en que el campo `data` puede contener cualquier tipo de dato —o estructura de datos—, mientras que en `data_count` simplemente se indica el número de bytes —`uint8_t`— de los datos —`sizeof(data)`. Se diferencian dos aspectos de implementación:

1. *Implementación desde el* driver → En la sección E.2.14.2 se explica como realizar la implementación desde el *driver*, para poner una estructura de datos en el campo `data`.

2. *Implementación desde el* proxy → Para tomar la estructura de datos en un aplicación de *Player* que use el *proxy OpaqueProxy*, se tendrán que seguir los pasos explicados en la sección E.2.7.1.

E.2.7.1. *Proxy OpaqueProxy*

En el cuadro E.11 se muestra la API del *proxy* de la interfaz *opaque* —i.e. *OpaqueProxy*. Hay que prestar especial atención a las siguientes aclaraciones, en las que se incluye la forma en que se debe implementar la obtención de la estructura de datos compartida.

1. En realidad el método `GetCount()` no es necesario para obtener la estructura de datos —desde el punto de vista programático o de implementación—, ya que con el método `GetData()` será suficiente.

2. Con el método `GetData()` se obtienen los datos transmitidos desde el *driver* al servidor de *Player*. En concreto se trata del campo `data` de la estructura de datos `player_opaque_data` (ver cuadro E.10). Como se muestra en el algoritmo E.6, se debe declarar la variable `uint8_t *datosOpacos` para tomar los datos llamando a `GetData()`. Luego simplemente se tiene que hacer un *cast* —con `reinterpret_cast<test_t*>`— para convertir los datos al tipo de la estructura de datos compartida —`test_t` en el ejemplo del algoritmo E.6 (ver sección E.2.14.2 para conocer cómo es esta estructura de datos y de dónde se ha tomado el ejemplo).

Método		Descripción
Tipo	Nombre	
uint	GetCount()	Tamaño de los datos en *bytes*.
void	GetData (uint8_t *aDest)	Toma los datos opacos enviados desde el *driver* al servidor de *Player*.
void	SendCmd (player_opaque_data_t *aData)	Envía un comando desde el servidor de *Player* al *driver*; los datos del comando se envían con la estructura de datos *player_opaque_data* —que es la misma que *player_opaque_data_t*.

Cuadro E.11: API de *OpaqueProxy*

```
1 OpaqueProxy op(&robot, 0);
2 uint8_t *datosOpacos;
3 op.GetData(datosOpacos);
4 test_t mTestStruct = *reinterpret cast<test_t*>(datosOpacos);
```

Algoritmo E.6: Obtención de los datos del *proxy OpaqueProxy*

3. Una vez se tienen los datos en una variable con el tipo de la estructura de datos compartida —test_t mTestStruct en el ejemplo (ver sección E.2.14.2 para conocer sus campos)— se podrá acceder a cualquiera de sus campos para llevar a cabo las tareas que deseamos, como se muestra en el algoritmo E.7.

```
1 cout << mTestStruct.uint8 << endl;
2 // Otros operaciones con los campos
```

Algoritmo E.7: Uso de los datos del *proxy OpaqueProxy*

E.2.8. Soporte para la interfaz *position3d*

La documentación de la interfaz *position3d* se encuentra en:

http://playerstage.sourceforge.net/doc/Player-2.0.0/player/group__interface__position3d.html

En el cuadro E.12 se describen los mensajes que maneja esta interfaz y en el cuadro E.13 se describen las estructuras de datos que se manejan —sin entrar en el detalle de sus campos. Hay que considerar las siguientes aclaraciones:

1. Los datos del estado son básicamente la posición y velocidad en un espacio tridimensional —i.e. tres coordenadas lineales y tres angulares—, y el estado de los motores, i.e. si están en marcha o parados —*stall*.

2. La geometría hace referencia a la pose y dimensiones —caja contenedora o *bounding box*— de la base del robot o vehículo.

3. El estado del motor puede ser apagado (*off*) o encendido (*on*), que se codifica como `false` y `true`, respectivamente. Esto es útil cuando se permite el encendido y apagado de los motores desde software —con el *driver*.

4. Los modos de control son dos: modo velocidad (0) y modo posición (1).

5. En ciertos casos se pueden aplicar diferentes modos de control de velocidad, los cuales serían específicos del *driver*.

Mensaje (*PLAYER_POSITION3D_*)	Tipo Comunicación	Descripción
DATA_STATE	Dato	Estado.
DATA_GEOMETRY	Dato	Geometría.
CMD_SET_VEL	Comando	Control de velocidad, indicando la velocidad deseada.
CMD_SET_POS	Comando	Control de posición, indicando la posición deseada y la velocidad con que alcanzarla.
GET_GEOM	Petición/Respuesta	Obtener la geometría.
MOTOR_POWER	Petición/Respuesta	Potencia de los motores.
VELOCITY_MODE	Petición/Respuesta	Modo velocidad.
POSITION_MODE	Petición/Respuesta	Modo posición.
RESET_ODOM	Petición/Respuesta	Resetea la odometría.
SET_ODOM	Petición/Respuesta	Fija la odometría.
SPEED_PID	Petición/Respuesta	Fija los parámetros PID de velocidad.
POSITION_PID	Petición/Respuesta	Fija los parámetros PID de posición.
SPEED_PROF	Petición/Respuesta	Fija los parámetros del perfil de velocidad.

Cuadro E.12: Mensajes de la interfaz *position3d*

Estructura de datos (*player_position3d_*)	Mensaje asociado (*PLAYER_POSITION3D_*)	Descripción
data	*DATA_STATE*	Datos del estado.
cmd_pos	*CMD_SET_POS*	Posición y velocidad deseada.
cmd_vel	*CMD_SET_VEL*	Velocidad deseada.
geom	*GET_GEOM*	Geometría.
power_config	*MOTOR_POWER*	Estado del motor.
position_mode_req	*POSITION_MODE*	Modo de control.
velocity_mode_config	*VELOCITY_MODE*	Modo de control de velocidad.
set_odom_req	*SET_ODOM*	Posición de odometría deseada.
reset_odom_config	*RESET_ODOM*	Resetea la odometría.
speed_pid_req	*SPEED_PID*	Parámetros PID de velocidad.
position_pid_req	*POSITION_PID*	Parámetros PID de posición.
speed_prof_req	*SPEED_PROF*	Parámetros del perfil de velocidad.

Cuadro E.13: Estructuras de datos de la interfaz *position3d*

6. El reseteo de la odometría consiste en poner a 0 todas las coordenadas lineales y angulares de la pose del robot, tal y como se muestra en la ecuación E.1.

$$(x, y, z, \phi, \theta, \psi) = (0, 0, 0, 0, 0, 0) \tag{E.1}$$

   En cuanto a la estructura de datos `player_position3d_reset_odom_config`, en realidad existe pero no contiene ningún campo, ya que no hay que indicar ninguna información; esto puede comprobarse en el fichero `player.h` de la carpeta `libplayercore/` de la distribución de *Player* (ver sección E.2.2).

7. En los parámetros del perfil de velocidad se definen básicamente la velocidad y aceleración máxima.

En la siguiente lista se enumeran los enlaces de la documentación de cada una de las estructuras de datos de la interfaz *position3d*.

1. `player_position3d_data`

   http://playerstage.sourceforge.net/doc/Player-2.0.0/player/structplayer__position3d__data.html

2. `player_position3d_cmd_pos`

   http://playerstage.sourceforge.net/doc/Player-2.0.0/player/structplayer__position3d__cmd__pos.html

3. player_position3d_cmd_vel

   http://playerstage.sourceforge.net/doc/Player-2.0.0/player/structplayer__position3d__cmd__vel.html

4. player_position3d_geom

   http://playerstage.sourceforge.net/doc/Player-2.0.0/player/structplayer__position3d__geom.html

5. player_position3d_power_config

   http://playerstage.sourceforge.net/doc/Player-2.0.0/player/structplayer__position3d__power__config.html

6. player_position3d_position_mode_req

   http://playerstage.sourceforge.net/doc/Player-2.0.0/player/structplayer__position3d__position__mode__req.html

7. player_position3d_velocidty_mode_config

   http://playerstage.sourceforge.net/doc/Player-2.0.0/player/structplayer__position3d__velocity__mode__config.html

8. player_position3d_set_odom_req

   http://playerstage.sourceforge.net/doc/Player-2.0.0/player/structplayer__position3d__set__odom__req.html

9. player_position3d_reset_odom_config

   http://playerstage.sourceforge.net/doc/Player-2.0.0/player/structplayer__position3d__reset__odom__config.html

10. player_position3d_speed_pid_req

    http://playerstage.sourceforge.net/doc/Player-2.0.0/player/structplayer__position3d__speed__pid__req.html

11. player_position3d_position_pid_req

    http://playerstage.sourceforge.net/doc/Player-2.0.0/player/structplayer__position3d__position__pid__req.html

12. player_position3d_speed_prof_req

    http://playerstage.sourceforge.net/doc/Player-2.0.0/player/structplayer__position3d__speed__prof__req.html

En el cuadro E.14 se describen los campos de la estructura de datos player_position3d_data. En estos campos se hace uso de la estructura de datos player_pose3d (ver cuadro E.15), con la que se define la pose del vehículo o robot en tres dimensiones —i.e. en el espacio. Esta estructura dispone de la documentación en:

Campo Tipo	Nombre	Descripción
player_pose3d_t	pos	Posición tridimensional, i.e. las coordenadas $(x, y, z, \phi, \theta, \psi,)$ con las unidades (m, m, m, rad, rad, rad).
player_pose3d_t	vel	Velocidad tridimensional, i.e. las coordenadas $(x, y, z, \phi, \theta, \psi,)$ con las unidades (m/s, m/s, m/s, rad/s, rad/s, rad/s).
uint8_t	stall	Indica si el motor está parado —*stall*.

Cuadro E.14: Campos de la estructura de datos *player_position3d_data*

Campo Tipo	Nombre	Descripción
float	px	Coordenada x, medida en m.
float	py	Coordenada y, medida en m.
float	pz	Coordenada z, medida en m.
float	proll	Coordenada ϕ, medida en rad.
float	ppitch	Coordenada θ, medida en rad.
float	pyaw	Coordenada ψ, medida en rad.

Cuadro E.15: Campos de la estructura de datos *player_pose3d*

Campo Tipo	Nombre	Descripción
player_pose3d_t	pos	Posición tridimensional, i.e. las coordenadas $(x, y, z, \phi, \theta, \psi,)$ con las unidades (m, m, m, rad, rad, rad).
player_pose3d_t	vel	Velocidad tridimensional, i.e. las coordenadas $(x, y, z, \phi, \theta, \psi,)$ con las unidades (m/s, m/s, m/s, rad/s, rad/s, rad/s).
uint8_t	state	Indica si el motor está apagado o parado/bloqueado —*locked*—, dependiendo del *driver*.

Cuadro E.16: Campos de la estructura de datos *player_position3d_cmd_pos*

http://playerstage.sourceforge.net/doc/Player-2.0.0/player/structplayer__pose3d.html

La simbología usada para las coordenadas lineales y angulares en el espacio —i.e. tridimensionales— se usa la notación SNAME, mostrada en el cuadro **??**, de la sección **??**. Aunque en la documentación oficial no se indican correctamente las unidades de velocidad, se ha supuesto que es el espacio —lineal o angular— recorrido por segundo.

En el cuadro E.16 se describen los campos de la estructura de datos `player_position3d_cmd_pos`. Es practicamente igual a la estructura de datos `player_position3d_data` —tambien usa la estructura de datos `player_pose3d` en sus campos—, pero su uso o finalidad es diferente, como se indica en el cuadro E.13.

NB: Sólo se han comentado las empleadas (*player_position3d_cmd_pos* no se ha usado aún).

Debido a la gran cobertura que ofrece esta interfaz, normalmente hay *drivers* que sólo dan soporte a una parte de la interfaz. Por ejemplo, una brújula sólo proporcionaría el rumbo, i.e. el ángulo de Euler *yaw*, que se indicaría en el campo `pos.pyaw`. A lo sumo podría calcularse la derivada de este valor —i.e. la velocidad— calculando la diferencia entre valores consecutivos y la variación temporal entre la realización de la medición de los mismos —e.g. a partir de las orientaciones o posiciones angulares proporcionadas por un inclinómetro se podrían computar las velocidades derivando. De forma análoga, se pueden integrar las velocidades para conocer las posiciones —e.g. a partir de las velocidades angulares proporcionadas por un giróscopo se podrían computar las posiciones angulares u orientaciones integrando. Estos ejemplo pueden verse en la sección **??**, donde se documentan los intrumentos de navegación, dentro del equipamiento sensorial y

Método		Descripción
Tipo	Nombre	
void	SetSpeed(double aXSpeed, double aYSpeed, double aZSpeed, double aRollSpeed, double aPitchSpeed, double aYawSpeed)	Envía un comando a los motores con la velocidad deseada (6 DOF).
void	SetSpeed(double aXSpeed, double aYSpeed, double aZSpeed, double aYawSpeed)	Comanda los motores con la velocidad deseada (4 DOF).
void	SetSpeed(double aXSpeed, double aYSpeed, double aYawSpeed)	Comanda los motores con la velocidad deseada (3 DOF).
void	SetSpeed(double aXSpeed, double aYawSpeed)	Comanda los motores con la velocidad deseada (2 DOF).
void	SetSpeed(player_pose3d_t vel)	Comanda los motores con la velocidad deseada indicada en *player_pose3d_t* (6 DOF).
void	GoTo(player_pose3d_t aPos, player_pose3d_t aVel)	Control de posición —i.e. posición a alcanzar y velocidad deseada.
void	GoTo(player_pose3d_t aPos)	Control de posición —i.e. sólo la posición a alcanzar.
void	GoTo(double aX, double aY, double aZ, double aRoll, double aPitch, double aYaw)	Control de posición sin usar *player_pose3d_t* —i.e. sólo la posición a alcanzar.
void	SetMotorEnable(bool aEnable)	Habilita/Deshabilita los motores.
void	SelectVelocityControl(int aMode)	Selecciona el modo de control de velocidad.
void	ResetOdometry()	Resetea la odometría.
void	SetOdometry(double aX, double aY, double aZ, double aRoll, double aPitch, double aYaw)	Fija la odometría deseada.
double	GetXPos()	Posición x.
double	GetYPos()	Posición y.
double	GetZPos()	Posición z.
double	GetRoll()	Ángulo de Euler (orientación) ϕ.
double	GetPitch()	Ángulo de Euler (orientación) θ.
double	GetYaw()	Ángulo de Euler (orientación) ψ.
double	GetXSpeed()	Velocidad lineal x.
double	GetYSpeed()	Velocidad lineal y.
double	GetZSpeed()	Velocidad lineal z.
double	GetRollSpeed()	Velocidad angular ϕ.
double	GetPitchSpeed()	Velocidad angular θ.
double	GetYawSpeed()	Velocidad angular ψ.
bool	GetStall()	Indica si los motores están parados.

Cuadro E.17: API de *Position3dProxy*

actuador.

E.2.8.1. *Proxy Position3dProxy*

En el cuadro E.17 se muestra la API del *proxy* de la interfaz *position3d* —i.e. *Position3dProxy*.

Respecto a la API de esta interfaz se enuncian las siguientes aclaraciones:

1. La versión de 2 DOF para comandar la velocidad de los motores está pensada para robots no holonómicos[4], por lo que no se indican las velocidades lineales en las coordenadas y y z.

2. El reseteo de la odometría implica que la posición y orientación pasará a ser la indicada en la ecuación E.1.

[4]En robótica el concepto *holonomicidad* hace referencia a la relación entre grados de libertad (DOF) controlables y totales, para un robot dado. Desde este punto de vista se distinguen tres tipos de robots:

Holonómico DOF controlables >= DOF totales —e.g. un AUV con motores para moverse de forma independiente en los 6 DOF.

No Holonómico DOF controlables < DOF totales —e.g. un AUV tipo torpedo o misil.

Redundante DOF controlables > DOF totales —e.g. un brazo robótico con los mismos DOF que el de un ser humano. Por definición, un robot redundante también será holonómico.

E.2.9. Implementación de una nueva interfaz en *Player*

También será posible añadir una nueva interfaz a *Player*, si bien no se dispone de información oficial de cómo hacerlo. Por ello, en esta sección se explica *grosso modo* en qué zonas del código fuente de la distribución de *Player* (ver sección E.2.2) hay que realizar modificaciones, así como las distintas tareas a cumplimentar.

1. *Definir los mensajes que se transmitirán entre el servidor de* Player *y los* drivers → Esta tarea es puramente de diseño y no se tendrá que implementar nada realmente. No obstante, habrá que especificar las estructuras de datos que se manejen en el proceso de comunicación en lenguaje C. Del mismo modo, se indicará cada tipo de mensaje con una macro de definición. Esta información será importante para el desarrollo de la interfaz, tanto desde el punto de vista del servidor de *Player*, como desde los *drivers*.

2. *Adaptar un* driver *que dé soporte a la interfaz* → Se tiene que adaptar un *driver* existente —o bien implementar uno directamente— a *Player* —heredando de la clase base `Driver`—, que dé soporte a la nueva interfaz. Esto implica que debe responder a los mensajes definidos y rellenar las estructuras de datos diseñadas para la comunicación de información al servidor de *Player*. Si se crea un *driver* estático (ver sección **??**) se tendrá que ubicar dentro de la estructura de directorios de la distribución de *Player* (ver sección E.2.2); se trata de la carpeta `server/drivers/`. En el caso de crear un *plugin driver* esto no será necesario.

3. *Crear la API del* proxy *para la interfaz* → A partir de las estructuras de datos y los mensajes definidos para la interfaz, se diseñará la API del *proxy* para la interfaz. Esta API determina las funciones que se usarán desde la aplicación cliente. Para crear un nuevo *proxy* —que heredará de la clase base `Proxy`— hay que implementar una clase para el mismo —e.g. para la interfaz del *gps* se disponde la clase `GpsProxy` en el fichero `gpsproxy.cc`— y declarar su interfaz en el fichero `playerc++.h`; todos estos ficheros estarán en el directorio `client_libs/libplayerc++/`.

4. *Modificar la función de comunicación con el servidor de* Player → Para la comunicación desde el *driver* al servidor de *Player*, la clase base `Driver` dispone del método `Publish` —con varios parámetros. Al crear una nueva interfaz con sus mensajes y estructuras de datos, será necesario indicar qué nuevos mensajes debe manejar y preparar el soporte para la comunicación de las nuevas estructuras de datos —como *Player* usa XDR [Group, 1987] esta tarea no será compleja.

5. *Modificar los ficheros de compilación* → Se tendrán que actualizar/modificar los ficheros que realizan la compilación de las interfaces —i.e. los *proxies* y los *drivers*— , para que la nueva interfaz esté disponible. Igualmente se podrán crear ficheros de configuración de ejemplo, donde se indique qué parámetros se tienen o pueden indicar al instanciar el *driver* —de la nueva interfaz— en éstos.

E.2.10. Implementación de *drivers* para *Player*

E.2.11. API del *driver*

Todo *driver* para el *framework* de *Player* debe disponer de las siguientes funciones:

Constructor del *driver* Se lee la configuración general del fichero de configuración. El parámetro `interface` indica a qué interfaz da soporte el *driver* —e.g. la interfaz[5] *localise*, *position*, etc. Su prototipo se muestra en el algoritmo E.8.

```
Driver::Driver(ConfigFile *cf, int section, int interface, uint8_t access,
               size_t datasize, size_t commandsize,
               size_t reqqueuelen, size_t repqueuelen);
```

Algoritmo E.8: Constructor del *driver*

Inicialización o Configuración Se hace cualquier tipo de configuración específica del dispositivo que controla el *driver*, como puede ser la apertura de ficheros/buffers o la configuración de puertos seriales. Además, se inicia el hilo del propio *driver*. Su prototipo se muestra en el algoritmo E.9.

```
Driver::Setup()
```

Algoritmo E.9: Función de inicialización del *driver*

Finalización Se llevan a cabo las tareas de limpieza al finalizar el uso del *driver*, como es el caso del cierre de ficheros/buffers. Su prototipo se muestra en el algoritmo E.10.

```
Driver::Shutdown()
```

Algoritmo E.10: Función de finalización del *driver*

Ejecución Se trata de un bucle infinito que actualiza, de forma repetitiva, los datos y envíos para la interfaz a la que se da soporte con el *driver*. Su prototipo se muestra en el algoritmo E.11.

```
Driver::Main()
```

Algoritmo E.11: Función de ejecución del *driver*

Envío de Datos Sirve para preparar datos en el formato especificado por *Player*, para que el sistema que proporciona sea capaz de leer y enviar dichos datos. Su prototipo se muestra en el algoritmo E.12.

```
Driver::SendData()
```

Algoritmo E.12: Función de envío de datos del *driver*

[5] En *Player* el término interfaz hace referencia a la API de los distintos dispositivos que se permiten en *Player*, entendidos como en un Sistema Operativo. En este sentido, dichas interfaces forman parte de la capa de abstracción del hardware —Hardware Abstraction Layer (HAL).

E.2.12. Tipos de *drivers*

En *Player* existen dos tipos diferentes de *drivers*:

1. *Static Drivers* → Se trata de *drivers* estáticos o residentes en la librería de *drivers* de *Player*. Su código reside en la distribución *Player* y se enlazan de forma estática con el servidor de *Player*. En principio, este tipo de *drivers* sólo deben añadirse cuando estén adecuadamente probados y funcionen correctamente. Además, para incluir este tipo de *drivers* hay que adaptar los ficheros de configuración y construcción de *Player* a partir del código fuente. En este sentido, los *static drivers* permiten la distribución de los mismos como parte del sistema de *Player*.

2. *Plugin Drivers* → Son objetos compartidos que se cargan en tiempo de ejecución —en cierto modo son como los módulos cargables en el *kernel* de Linux. Son el método más aconsejable para la creación de *drivers* nuevos, experimentales (en fase de prueba o desarrolllo) o de terceros.

Los *plugin drivers* presentan ciertas ventajas respecto a los *static drivers*:

1. Son más fáciles de construir, ya que no hay que lidiar con herramientas de construcción —*autotools*[6]— ni sumergirse en los detalles internos del servidor de *Player*.

2. Permiten un desarrollo rápido y un ciclo de codificación, compilación y prueba mucho más rápido, dado que no se tiene que recompilar y reenlazar el servidor cuando se cambian estos *drivers*.

3. El código desarrollado para el *driver* se puede mantener de forma independiente. Esto es particularmente útil para usuarios con desarrollos de *drivers* que disponen de su propio código, datos, documentación, etc. fuerta de la estructura de directorios del código fuente de *Player* —desarrollos software independientes, en definitiva.

E.2.13. Construir un *Plugin Driver*

A continuación se describe el proceso de creación de nuevos *plugin drivers*. Se requiere un conocimiento básico de C++, herencia de clases y programación con hilos —threads.

El código de ejemplo —que podría adaptarse para servir de esqueleto en el desarrollo de *plugin drivers*— de un *plugin driver* básico se proporciona en el directorio de ejemplos de la distribución de *Player*; en una instalación por defecto[7] se encontrará en:

```
/usr/local/share/player/examples/plugins/exampledriver/
```

Estos ficheros se pueden copiar a otro directorio y renombrar **Makefile.example** a **Makefile** —en el algoritmo E.13 se muestra el contenido inicial del directorio y después de renombrar este fichero— para luego construir el ejemplo ejecutando la instrucción indicado en el algoritmo E.14. En este algoritmo también se muestran las órdenes que realmente se lanzan —que se comentan en la sección E.2.16— y como se obtiene el driver en el directorio actual, con el nombre `"exampledriver.so"` —que se usa en el campo `plugin` del bloque del *driver*, mostrado en el algoritmo E.16.

[6] Bajo el nombre de *autotools* se hace referencia a las herramientas para la configuración y construcción automática de ficheros ejecutables o librerías a partir del código fuente. Algunas de ellas son **automake**, **autoconf** y otras similares o de apoyo a éstas.

[7] Se considera la instalación de la versión 2.0.3 de *Player*.

```
$ ls
example.cfg                  libexampledriver.a   libexampledriver.so
libexampledriver.so.0.0.0    Makefile.libtool     SConstruct
exampledriver.cc             libexampledriver.la  libexampledriver.so.0
Makefile.example             Makefile.osx.example
$ mv Makefile.example Makefile
$ ls
example.cfg                  libexampledriver.a   libexampledriver.so
libexampledriver.so.0.0.0    Makefile.libtool     SConstruct
exampledriver.cc             libexampledriver.la  libexampledriver.so.0
Makefile                     Makefile.osx.example
```

Algoritmo E.13: Ficheros del *driver* de ejemplo

```
$ make
g++ -Wall -fpic -g3 `pkg-config --cflags playercore` -c exampledriver.cc
g++ -shared -nostartfiles -o libexampledriver.so exampledriver.o
$ ls
example.cfg                  exampledriver.o      libexampledriver.la
libexampledriver.so.0        Makefile             Makefile.osx.example
exampledriver.cc             libexampledriver.a   libexampledriver.so
libexampledriver.so.0.0.0    Makefile.libtool     SConstruct
```

Algoritmo E.14: Comando para construir el *driver* de ejemplo

Esto creará un *plugin driver* con el nombre **exampledriver.so**, que puede probarse usando el fichero de configuración incluido (**example.cfg**), de modo que habrá que lanzar la instrucción del algoritmo E.15, cuya salida muestra como se prepara el *driver* para que escuche en el puerto 6665 —en este caso—, tras pasar por las diferentes fases, que se explican detalladamente en la sección E.2.14. El *driver* se da de baja o termina con **Ctrl**+**C**, momento en que se lanzan las funciones para la finalización del hilo asociado al *driver*, lo cual produce el mensaje `"Quitting"` por la salida estándar —luego se vuelve a tener el *prompt* del intérprete de comandos.

```
$ player example.cfg

* Part of the Player/Stage/Gazebo Project [http://playerstage.sourceforge.net].
* Copyright (C) 2000 - 2006 Brian Gerkey, Richard Vaughan, Andrew Howard,
* Nate Koenig, and contributors. Released under the GNU General Public License.
* Player comes with ABSOLUTELY NO WARRANTY.  This is free software, and you
* are welcome to redistribute it under certain conditions; see COPYING
* for details.

trying to load /driverplayer/./libexampledriver...
success
invoking player_driver_init()...
Example driver initializing
Example driver done
success
Listening on ports: 6665
Quitting.
```

Algoritmo E.15: Comando para probar el *driver* de ejemplo

El bloque del *driver* en el fichero de configuración tiene un campo adicional, denominado `plugin`, para indicar la ruta específica del *plugin driver*, como se muestra en el algoritmo E.16.

```
driver
(
  name "exampledriver"
  plugin "exampledriver.so"
  provides ["position:0"]
  ...
)
```

Algoritmo E.16: Bloque del *driver* de ejemplo

E.2.14. Implementar un *Plugin Driver*

El primer paso en la creación de un nuevo *driver* consiste en decidir a qué interfaz de *Player* dará soporte. Las múltiples interfaces existentes se describen en la documentación de *Player* [Gerkey et al., 2006] y sus múltiples estructuras de mensajes y constantes están definidas en el fichero **player.h**. En la sección **??** se comentan algunas de estas interfaces —en concreto las usadas para el desarrollo de los sensores y actuadores (ver sección **??**) del sistema del AUV— y en la tabla E.1 se muestra la lista de interfaces de *Player*[7].

Aunque es posible crear una nueva interfaz, es aconsejable ajustarse a una interfaz existente —o usar la interfaz genérica *opaque* (ver sección E.2.7). Dando soporte a una interfaz ya existente, se tendrá que realizar menos trabajo de programación en la parte del servidor de *Player* y será más fácil que haya soporte de clientes para el *driver*. Además, la filosofía de *Player* radica en el modelo HAL de los Sistemas Operativos, de modo que en la medida de los posible los *drivers* deben dar soporte a las interfaces ya definidas, evitando crear otras nuevas.

La creación de un nuevo *driver* consiste básicamente en crear una nueva clase para el *driver*, que debe heredar de la clase base `Driver`. Esta clase base define la API[8], parte de la cual debe implementarse en el nuevo *driver* —que serían las funciones virtuales de dicha clase base—, mientras que otras partes podrían sobrecargarse si se desea.

A continuación se describen las características de la clase `Driver`:

Constructor Los *drivers* simples usarán el constructor de la clase base `Driver` indicado en el algoritmo E.8. Este constructor establecerá los buffers y colas que permitirán que el servidor de *Player* pueda comunicarse o acceder a la interfaz del *driver*. Así, para el *driver* de ejemplo tenemos el prototipado del algoritmo E.17.

```
ExampleDriver::ExampleDriver(ConfigFile* cf, int section)
              :Driver(cf, section, false,
                      PLAYER_MSGQUEUE_DEFAULT_MAXLEN,
                      PLAYER_POSITION2D_CODE){
  ...
}
```

Algoritmo E.17: Prototipado del constructor del *driver* de ejemplo

El prototipado anterior indica que este *driver*:

1. Los nuevos comandos no deben sobreescribir a los antiguos, lo cual se indica en el tercer parámetro con `false`.
2. Tiene un buffer de datos lo suficientemente grande como para almacenar un paquete de datos del tamaño máximo por defecto, lo cual se indica con `PLAYER_MSGQUEUE_DEFAULT_MAXLEN`. También es costumbre indicar el tamaño de la estructura de datos asociada a la interfaz —e.g. en el caso de la interfaz *position2d* se trataría de indicar `sizeof(player_position_data_t)`.
3. Da soporte a la interfaz de *Player* *position2d*, lo cual se indica con la macro `PLAYER_POSITION2D_CODE`.

En la API actual de *Player*8 la anterior es la única interfaz admitida para el constructor de la clase base `Driver` —aunque admite la omisión de ciertos argumentos, que adoptarán valores por defecto. Sin embargo, versiones anteriores permitían la indicación de aspectos adicionales, pero ya no deben emplearse.

[8]Se considerará la API 2.0 de *Player*, de acuerdo con la versión 2.0.3 de éste.

El servidor de *Player* pasa los parámetros `cf` y `section`, que permiten el acceso a las opciones específicas del *driver*, almacenadas en el fichero de configuración de *Player*. Así, por ejemplo, el constructor del *driver* puede leer el valor de un parámetro de configuración `foo`, que se tomaría con la función `ReadInt` —si se trata de un parámetro de tipo entero (`int`), que es miembro de la clase `ConfigFile`. En el algoritmo E.18 se observa como se usan los parámetros `cf` y `section` para conseguir el mencionado valor de `foo`; si este no existiera en el fichero de configuración se tomaría el tercer parámetro de la función por defecto —e.g. el 0 en el ejemplo del algoritmo E.18.

```
this->foop = cf->ReadInt(section, "foo", 0);
```

Algoritmo E.18: Obtención del parámetro `foo` del fichero de configuración

El valor de este parámetro podría ser el puerto serial del que se tendrían que leer los datos, por ejemplo —en tal caso, el tipo de datos del parámetro sería una cadena de caracteres (`string`, en *Player*, o `char[]` y `char*`, en C/C++). En la documentación de *Player* [Gerkey et al., 2006] referente a `ConfigFile` se pueden consultar las diferentes opciones que se pueden leer del fichero de configuración.

Inicialización: *Setup* Cuando el primer cliente se subscribe al *driver* se llama al método de inicialización `Setup()` del *driver*; todos los *drivers* deben implementar este método, que de forma general debe encargarse de las siguientes tareas:

1. Realizar la inicializaciones específicas del dispositivo —e.g. abrir un puerto serial.
2. Lanzar el hilo —thread— del *driver*, usando `Driver::StartThread()`.

Player es un sistema multienhebrado[9], con la mayoría *drivers* ejecutándose con su propio hilo. Esto hace que sea fácil implementar *drivers* que leen/escriben datos de puertos seriales, sockets, ficheros en disco, etc.

Después de la inicialización, `Setup()` debe devolver 0 para indicar que el dispositivo se ha inicializado —o configurado— correctamente, o un valor diferente de 0 para indicar que se produjo algún fallo —esto provocará que se cierre la aplicación de *Player*.

Finalización: *Shutdown* Cuando el último cliente se da de baja —de la subscripción que hizo— se llama al método de finalización `Shutdown()` del *driver*; todos los *drivers* deben implementar este método, que de forma general debe encargarse de las siguientes tareas:

1. Parar el hilo del *driver* usando `Driver::StopThread()`.
2. Realizar las finalizaciones específicas del dispositivo —e.g. cerrar un puerto serial.

El orden en estas tareas es importante: hay que parar el hilo del *driver* antes de liberar los recursos que éste usa. Para este fin, `Driver::StopThread()` le indicará al hilo del *driver* que termine y esperará hasta que éste termine antes de retornar.

`Shutdown()` debe devolver 0 para indicar que el dispositivo se finalizó correctamente, o un valor diferente de 0 para indicar que falló la finalización.

[9]Se entiende por sistema multienhebrado o multihilo, aquél en que se lanzan múltiples hilos para realizar las diferentes tareas que se llevan a cabo en el sistema; en inglés este término es *multi-threaded*.

Ejecución El método `Driver::Main()` será invocado como resultado de la creación del nuevo hilo de control; en la API anterior a la actual[8] primero se llamaba a `Driver::Startup()`, pero esto ya no es así. Todos los *drivers* deben sobrecargar este método, que de forma general se encargará de la traducción entre la API específica del dispositivo y las interfaces estándar de *Player*. Los pasos básicos son los siguientes:

1. Para datos entrantes (desde el *driver* al servidor de *Player*):
   a) Leer datos de algún dispositivo externo —e.g. desde un puerto serial.
   b) Almacenar los datos en una de las estructuras de datos de la interfaz de *Player*.
   c) Escribir los datos en el servidor de *Player* usando el método denominado `Driver::Publish()`, que dispone de varios parámetros y lo habitual es que se indiquen los del algoritmo E.19 —obtenido del fichero `driver.h` de la carpeta `libplayercore/` del código fuente de la distribución de *Player* (ver sección E.2.2)—, ya que está sobrecargada con distintos parámetros.

```
void Publish(player_devaddr_t addr,
             MessageQueue *queue,
             uint8_t type,
             uint8_t subtype,
             void *src = NULL,
             size_t len = 0,
             double *timestamp = NULL);
```

Algoritmo E.19: Prototipado de la función `Driver::Publish()`

   Es habitual crear una función denominada `PutData()`, que se encargará de realizar la publicación de todos los datos actualizados al servidor de *Player*, usando la función `Driver::Publish()` internamente en última instancia, con una llamada como la que se muestra en el algoritmo E.20. Se trata de un ejemplo para comunicar los datos de un GPS, dando soporte a la interfaz *gps* de *Player*.

```
Publish(device_addr, NULL,
        PLAYER_MSGTYPE_DATA, PLAYER_GPS_DATA_STATE,
        reinterpret_cast<void*>(&datos), sizeof(player_gps_data_t), NULL);
```

Algoritmo E.20: Llamada a la función `Driver::Publish()`

2. Para datos salientes (desde el servidor de *Player* al *driver*):
   a) Leer los comandos o mensajes del servidor de *Player*. Para detectar estos comandos o mensajes, provinientes del servidor de *Player*, la clase base `Driver` provee un mecanismo asíncrono. Éste consiste en la llamada de forma automática al método `Driver::ProcessMessage()` para cada mensaje recibido. El prototipado de este método se muestra en el algoritmo E.21. Se debe reimplementar para aplicar las acciones oportunas a los mensajes que se reciban. Para determinar qué mensaje se ha recibido se tiene que usar el método `Message::MatchMessage()`, del que se muestra un ejemplo de llamada en el algoritmo E.22 —un ejemplo de un *driver* de la interfaz *laser*. Una vez se determina de qué mensaje se trata —de acuerdo con los mensajes admitidos por la interfaz a la que se da soporte—, se aplicarán las acciones oportunas.

```
int ProcessMessage(MessageQueue *resp_queue,
                   player_msghdr *hdr,
                   void *data);
```

Algoritmo E.21: Prototipado de la función Driver::ProcessMessage()

```
1  if(Message::MatchMessage(hdr, PLAYER_MSGTYPE_REQ,
2                           PLAYER_LASER_REQ_SET_CONFIG,
3                           this->device_addr)){
4    // Acciones para PLAYER_LASER_REQ_SET_CONFIG
5  }else if(Message::MatchMessage(hdr, PLAYER_MSGTYPE_REQ,
6                                 PLAYER_LASER_REQ_GET_CONFIG,
7                                 this->device_addr)){
8    // Acciones para PLAYER_LASER_REQ_GET_CONFIG
9  }else if(Message::MatchMessage(hdr, PLAYER_MSGTYPE_REQ,
10                                PLAYER_LASER_REQ_GET_GEOM,
11                                this->device_addr)){
12   // Acciones para PLAYER_LASER_REQ_GET_GEOM
13 }
```

Algoritmo E.22: Ejemplo de uso de la función Message::MatchMessage()

En la API actual[8] no se dispone del método Driver::GetCommand(), que ya no es necesario para la recepción de comandos o mensajes —en principio, de forma síncrona. En su lugar, basta implementar las acciones asociadas a cada una de los posibles mensajes que pueden recibirse del servidor de *Player*, dentro del método Driver::ProcessMessage(), determinando qué mensaje se tiene que tratar, tal y como muestra el algoritmo E.22. Por otro lado, se dispone del método Driver::ProcessMessages() —terminado en *s*. Éste se encarga de procesar todos los mensajes pendientes en la cola de mensajes recibidos desde el servidor de *Player*. Dispone de una implementación genérica en la clase base Driver, que puede sobrecargarse si se desea.

b) Almacenar el comando con el formato específico del dispositivo.

c) Escribir o enviar el comando al dispositivo externo.

Los métodos Driver::Publish() y Driver::ProcessMessage() manejan todas las técnicas de buffers intermedios y bloqueos requeridos para sincronizar los hilos del *driver* y el servidor de *Player*.

El método Main() sobrecargado queda liberardo de manejar las peticiones de configuración enviadas al *drivers* —o cualquier otro tipo de comando—, ya que de ello se encargará el método Driver::ProcessMessage(), invocado asíncronamente una vez para cada una de estas peticiones —i.e. comandos o mensajes. No obstante, el método Main() aún podrá encargarse de servir a todos los mensajes pendientes, llamando al método Driver::ProcessMessages(); si el servidor de *Player* —el productor de los mensajes— envía peticiones a mayor frecuencia que la frecuencia con que se cumplimentan las peticiones en el *driver* —el consumidor—, se puede producir una situación de bucle infinito —atendiendo a los mensajes recibidos y, por tanto, pendientes, que no se agotarán.

En las APIs anteriores a la actual[8] el método Main() sí tenía que manejar las peticiones de configuración, siguiendo estos pasos básicos:

1. Comprobar si hay nuevas peticiones con Driver::GetConfig().
2. Hacer lo necesario para cumplimentar las tareas asociadas a la petición.

3. Enviar una respuesta —de reconocimiento y realización de la petición— usando `Driver::PutReply()`.

Esto ya no es necesario. Por ello, desde el punto de vista de las comunicaciones con el servidor de *Player*, en el método `Main()` lo único que puede ser necesario —en determinadas ocasiones— es el envío de mensajes al servidor de *Player*, usando directamente el método `Driver::Publish()` —o llamando a otro método, como `PutData()`, que en última instancia llamará a `Driver::Publish()`.

Es importante que el *driver* responda, de una forma u otra, a todas las peticiones. Si no se consigue responder a las peticiones los clientes se bloquearán al comunicarse con el *driver*.

El método `Main()` sobrecargado también debe ser capaz de terminar correctamente —i.e. la función debe terminar limpiando los recursos usados cuando el usuario finaliza el servidor. Esto se consigue llamando a `pthread_testcancel()` dentro del bucle principal —al inicio del mismo. Esta función comprueba si el hilo del *driver* debe terminarse; en tal caso, se finaliza *Main()* inmediatamente. Si se requiere alguna tarea de limpieza adicional, los *drivers* también pueden sobrecargar el método `Driver::MainQuit()`, que se llamará en la terminación del hilo.

Existen ciertos tipos de *drivers* que requieren especial atención en su implementación o adaptación a *Player*. Se trata de los siguientes casos —ambos relacionados con las interfaces de *Player*.

1. *Multidriver* → Se trata de un *driver* que da soporte a más de una interfaz de *Player* a la vez. Esto modificará principalmente la implementación del constructor de la clase del mismo, como se verá en la sección E.2.14.1.

2. *Opaquedriver* → Se trata de un *driver* que da soporte a la interfaz *opaque* de *Player* (ver sección E.2.7). En la sección E.2.14.2 se explica cómo proceder para dar soporte a esta interfaz con un *driver*, mientras que en la sección E.2.7 se comenta la utilidad —o necesidad— de esta interfaz y cómo usar el *proxy* de la misma en una aplicación que use *Player* (ver sección E.2.7.1).

E.2.14.1. Implementar un *Multidriver*

Un *multidriver* —o un *driver* que da soporte a varias interfaces de *Player*— requiere ciertas aclaraciones a la hora de su implementación o adaptación. Éstas se han determinado a partir del ejemplo disponible en el directorio `examples/plugins/multidriver/` de la distribución de *Player* (ver sección E.2.2). Para empezar, para cada interfaz a la que se dé soporte se deben crear los siguientes atributos en la clase del *driver*:

1. *Interfaces* → Se deben crear tantos identificadores o direcciones a dispositivos —i.e. atributos del tipo `player_devaddr_t`— como el número de interfaces a las que se vaya a dar soporte. En el algoritmo E.23 se muestra un ejemplo en el que se da soporte a dos interfaces —en este punto no importa de qué tipo de interfaces se trate.

```
1 player_devaddr_t gps_addr;
2 player_devaddr_t posicion3d_addr;
```

Algoritmo E.23: Identifcadores para las interfaces

2. *Estructuras de datos* → Normalmente las interfaces tendrán alguna estructura de datos asociada —al menos una para el envío de datos al servidor de *Player*. De acuerdo con el algoritmo E.23, en el algoritmo E.24 se muestran las estructuras de datos para el envío de datos desde el *driver* al servidor de *Player*, para las interfaces que se usarán.

```
player_gps_data_t datos_gps;
player_position3d_data_t datos_position3d;
```

Algoritmo E.24: Estructuras de datos para las interfaces

Con estos atributos definidos, los pasos para la implementación de un *multidriver* son:

1. *Constructor* → Cuando sólo se da soporte a una interfaz se indica al constructor de la clase base `Driver` a qué interfaz de *Player* va a dar soporte el *driver* (ver sección E.2.14) con una macro —e.g. `PLAYER_POSITION2D_CODE` para la interfaz *position2d*. Sin embargo, cuando se da soporte a múltiples interfaces la subscripción a cada una de las interfaces debe realizarse programáticamente una por una, como muestra el algoritmo E.25 —al constructor de la clase base `Driver` sólo se le pasan dos parámetros y no se le indica ninguna interfaz a la que se dé soporte. Con el método `AddInterface()` se indicará que se da soporte a una interfaz dada. Sin embargo, para ello en el fichero de configuración la declaración del *driver* de incluir las interfaces a las que se da soporte en el campo `"provides"`. Con el método `cf->ReadDeviceAddr()` se obtienen las interfaces a las que da soporte el *driver*. Para ello se indica a cual nos referimos, para instanciar correctametne el identificador que se le asociará —e.g. a `gps_addr` se le asocia la interfaz *gps*. El significado de cada uno de los parámetros de interés es:

   Parámetro 4 Tipo de interfaz, indicando el código de la misma —e.g. `PLAYER_GPS_CODE` para la interfaz *gps*.

   Parámetro 5 El índice —*index*— de la interfaz. Con -1 se indica que se acepta cualquier índice.

   Parámetro 6 La clave —*key*— o identificador que se le asigna a la interfaz —e.g. `"gps"` para la interfaz *gps* en el ejemplo del algoritmo E.25.

```
MultiDriver::MultiDriver(ConfigFile* cf, int section)
            :Driver(cf, section){
  // Crear interfaz gps.
  if(cf->ReadDeviceAddr(&(gps_addr), section, "provides",
                        PLAYER_GPS_CODE, -1, "gps") != 0){
    SetError(-1);
    return;
  }
  if(this->AddInterface(gps_addr)){
    SetError(-1);
    return;
  }
  // Crear interfaz position3d.
  if(cf->ReadDeviceAddr(&(posicion3d_addr), section, "provides",
                        PLAYER_POSITION3D_CODE, -1, "posicion3d") != 0){
    SetError(-1);
    return;
  }
  if(this->AddInterface(posicion3d_addr)){
    SetError(-1);
    return;
  }
  // Otras tareas del constructor
}
```

Algoritmo E.25: Soporte para múltiples interfaces

En el algoritmo E.26 se muestra un ejemplo de fichero de configuración de un *multidriver* apropiado para el ejemplo del algoritmo E.25. Cada interfaz a la que se da soporte se indica en la lista del campo `provides`, usando el formato indicado en el algoritmo E.27, que se conoce como la especificación de la dirección del dispositivo; los elementos de este formato son:

a) *key* → Clave o identificador de la interfaz. Este campo es opcional y aporta semántica adicional al identificar una interfaz —e.g. indicando si se trata de una dispositivo específico, como un giróscopo.

b) *host* → Máquina en que se ejecuta el *driver*, i.e. un nombre de dominio —i.e. una URL— o dirección IP. Es opcional y por defecto será *localhost*.

c) *robot* → Puerto en el que escuchará el *driver*. Es opcional y por defecto será el 6665.

d) *interface* → Tipo de interfaz de *Player*; este campo es obligatorio y la interfaz debe ser una de la lista del cuadro E.1 (ver sección **??**).

e) *index* → Índice de la interfaz, útil cuando se da soporte a una misma interfaz más de una vez —e.g. en un *multidriver*.

```
driver
(
  name "multidriver"
  plugin "libmultidriver"
  provides ["gps:::gps:0" "position3d:::position3d:0"]
)
```

Algoritmo E.26: Fichero de configuración de un *multidriver*

```
key:host:robot:interface:index
```

Algoritmo E.27: Definición de la dirección de un dispositivo

2. *Ejecución* → Distinguiremos dos tareas bien diferenciadas —como en la sección E.2.14—, de acuerdo con el flujo de comunicación:

a) *Para datos entrantes (desde el* driver *al servidor de* Player*)* → En el miembro `Main()` se podrán rellenar los campos de la estructura de datos asociada a cada interfaz y se podrán enviar los datos al servidor de *Player* por separado. Se deben enviar por separado indicando a qué interfaz corresponden, como se muestra en el algortimo E.28, usando el método `Publish()`.

```
// Rellenar estructura de datos gps
Publish(gps_addr, NULL,
        PLAYER_MSGTYPE_DATA, PLAYER_GPS_DATA_STATE,
        reinterpret_cast<void*>(&datos_gps),
        sizeof(player_gps_data_t), NULL);
// Rellenar estructura de datos position3d
Publish(position3d_addr, NULL,
        PLAYER_MSGTYPE_DATA, PLAYER_POSITION3D_DATA_STATE,
        reinterpret_cast<void*>(&datos_position3d),
        sizeof(player_position3d_data_t), NULL);
```

Algoritmo E.28: Envío de datos a múltiples interfaces

b) *Para datos salientes (desde el servidor de* Player *al* driver*)* →

E.2.14.2. Implementar un *Opaquedriver*

Un *driver opaque* debe tener asociada una estructura de datos —o un tipo de datos—, que será la que contendrá los datos a enviar. Éstos se enviarán en un vector de bytes —`uint8_t`— denominado `data`, que es uno de los campos de la estructura de datos `player_opaque_data` (ver sección E.2.7).

Dentro del directorio `examples/plugins/opaquedriver/` de la distribución de *Player* (ver sección E.2.2) disponemos de un ejemplo de implementación de un *driver opaque*. Según las necesidades del dispositivo para el que se crea el *driver*, los datos a comunicar al servidor de *Player* determinarán la estructura de datos a emplear. Ésta podrá ser todo lo compleja que queramos; en el algoritmo E.29 se muestra la del *driver opaque* de ejemplo, la cual cubre campos de todos los tipos de datos básicos. A esta estructura de datos se le denomina estructura de datos compartida, porque se usará tanto en el *driver* como en el *proxy* desde una aplicación de *Player* (ver sección E.2.7.1). La estructura de datos compartida —como tal— se pondrá en un fichero a parte. En concreto se tratará de una cabecera —un fichero `.h` en C++—, que se podrá incluir en el fichero de implementación o adaptación del *driver* y en la aplicación de *Player*, donde se usará el *proxy OpaqueProxy* (ver sección E.2.7.1).

```
1  typedef struct {
2    uint8_t uint8;
3    int8_t int8;
4    uint16_t uint16;
5    int16_t int16;
6    uint32_t uint32;
7    int32_t int32;
8    double doub;
9  } test_t;
```

Algoritmo E.29: Estructura de datos compartida

En la clase del *driver* tendremos que incluir como atributos la estructura de datos compartida y la estructura de datos de la interfaz *opaque* (ver cuadro E.10 de la sección E.2.7), tal y como muestra el algoritmo E.30. En este caso se trata de `test_t *mTestStruct` y `player_opaque_data_t mData`, respectivamente.

```
1  class OpaqueDriver : public Driver {
2    // ...
3  private:
4    // Estructura de datos a enviar.
5    test_t *mTestStruct;
6    // Datos opacos/genericos por donde se envia la estructura de datos
7    player_opaque_data_t mData;
8  };
```

Algoritmo E.30: Atributos para enviar los datos opacos

Tras la declaraciones antes comentadas, los pasos y zonas a implementar son las siguientes:

1. *Constructor* → En el constructor se debe crear la asociación de la estructura de datos compartida al campo `data` de la estructura de datos `player_opaque_data` de la interfaz *opaque*. Esto se muestra en el algoritmo E.31, donde inicialmente se almacena el tamaño de la estructura de datos compartida —`test_t`— en el campo `data_count` y se aplica un *cast* —con `reinterpret_cast<test_t*>`[10]— para que se tome el puntero —la dirección de memoria— del campo `data` en la estructura de datos `mTestStruct`.

[10] El modelado de tipos —*typecasting* o simplemente *cast*— es el proceso de convertir o promover un objeto de un tipo a otro. Aunque el compilador es capaz de efectar el *cast* implícito, se recomienda

Como ahora ambas variables apuntan a la misma dirección de memoria, los cambios en una se ven reflejados en la otra. Al rellenar los campos de mTestStruct luego se tendrán en el campo data serializados —por ser un vector de bytes, i.e. uint8_t. Adicionalmente también se pueden inicializar —opcionalmente— los campos de la estructura de datos.

```
OpaqueDriver::OpaqueDriver(ConfigFile* cf, int section)
              : Driver(cf, section, false,
                       PLAYER_MSGQUEUE_DEFAULT_MAXLEN,
                       PLAYER_OPAQUE_CODE){
  mData.data_count = sizeof(test_t);
  mTestStruct = reinterpret_cast<test_t*>(mData.data);
  // Inicializar los campos de mTestStruct (opcional)
  mTestStruct->uint8 = 0;
  mTestStruct->int8 = 0;
  mTestStruct->uint16 = 0;
  mTestStruct->int16 = 0;
  mTestStruct->uint32 = 0;
  mTestStruct->int32 = 0;
  mTestStruct->doub = 0;
}
```

Algoritmo E.31: Asociar la estructura de datos compartida

Se observa que se da soporte a la interfaz *opaque*, al pasar al constructor de la clase padre Driver la macro PLAYER_OPAQUE_CODE. No obstante, esto coincide con el proceso de adaptación de un *driver* a cualquier otro tipo de interfaz de *Player*.

2. *Ejecución* → Distinguiremos dos tareas bien diferenciadas —como en la sección E.2.14—, de acuerdo con el flujo de comunicación:

   a) *Para datos entrantes (desde el* driver *al servidor de* Player*)* → En el miembro Main() se podrán rellenar los campos de la estructura de datos mTestStruct y luego enviarlos al servidor de *Player* usando el método Publish(), tal y como muestra el algoritmo E.32. Se debe calcular el tamaño de los datos a enviar y enviar los datos con la estructura de datos player_opaque_data —que requiere un *cast* a void* con reinterpret_cast<void*>. De esta forma, en el lado del servidor, con el tamaño de los datos —data_count— se podrán volver a convertir los datos —data— a la estructura de datos compartida (ver sección E.2.7.1) —e.g. test_t en los ejemplos mostrados.

```
  // Modificar estructura de datos
  mTestStruct->uint8 += 1;
  mTestStruct->int8 += 1;
  mTestStruct->uint16 += 5;
  mTestStruct->int16 += 5;
  mTestStruct->uint32 += 10;
```

indicar el tipo *cast* —i.e. el tipo de conversión— de forma explícita. En C++ se incorpara una nueva sintaxis, frente a la clásica de C. Esta nueva sintaxis define cuatro palabras clave:

const_cast Sirve para poner o quitar atributos const o volatile. Su sintaxis es const_cast<T>(arg), donde T y arg deben ser del mismo tipo, pero con diferente modificador const o volatile.

static_cast Realiza la conversión de forma estática y su uso se recomienda siempre que la conversión esté clara —en lugar de dejar los *cast* implícitos. Su sintaxis es static_cast<T>(arg).

dynamic_cast Realiza la conversión de forma dinámica —en ciertos casos en tiempo de ejecución. Se usa en conversiones de punteros y referencias a clases. Su sintaxis es dynamic_cast<T>(arg).

reinterpret_cast La conversión se realiza por *fuerza bruta*, i.e. se obliga al compilador a aceptar un tipo de objeto por otro, por muy ilógica que sea la transformación. Son conversiones peligrosas, pero en ciertos casos necesarias —e.g. realizar conversiones temporales, para realizar determinadas transformaciones y volver a interpretarlos en su sentido original. Su sintaxis es reinterpret_cast<T>(arg).

```
7    mTestStruct->int32 += 10;
8    mTestStruct->doub = sin(mTestStruct->uint8/10.0);
9
10   // Enviar datos
11   uint size = sizeof(mData) - sizeof(mData.data) + mData.data_count;
12   Publish(device_addr, NULL,
13           PLAYER_MSGTYPE_DATA, PLAYER_OPAQUE_DATA_STATE,
14           reinterpret_cast<void*>(&mData), size, NULL);
```

Algoritmo E.32: Envío de datos opacos al servidor de *Player*

b) Para datos salientes (desde el servidor de Player *al* driver*)* →

E.2.15. Instanciación de un *driver*

Para instanciar un *driver* el servidor de *Player* necesita conocer dos cosas:

1. El nombre del *driver*, tal y como aparecerá en el fichero de configuración.
2. La función de construcción[11] del *driver*, usada para crear una nueva instancia del *driver* en el servidor. Normalmente, esta función tendrá la apariencia que se muestra en el algoritmo E.33.

```
1  Driver* ExampleDriver_Init(ConfigFile* cf, int section){
2    return ((Driver*)(new ExampleDriver(cf, section)));
3  }
```

Algoritmo E.33: Función de construcción del *driver*

La función del algoritmo E.33 podrá llamarse múltiples veces: una para cada ocurrencia del nombre del *driver* en el fichero de configuración.

Cada *driver* debe registrarse él mismo en el servidor de *Player* usando el método `DriverTable::AddDriver()`, proporcionando el nombre del *driver* y la función de construcción. Por tanto, los *drivers* deben definir una función para registrarse como la del algoritmo E.34.

```
1  void ExampleDriver_Register(DriverTable* table){
2    table->AddDriver("exampledriver", ExampleDriver_Init);
3  }
```

Algoritmo E.34: Función para registrar el *driver*

La función de registro del *driver* E.34 debe llamarse exactamente una vez, al iniciarse el programa. Para los *plugin drivers* esto se hace definiendo una función de inicialización para el objeto o librería compartida, como se muestra en el algoritmo E.35.

```
1  extern "C" {
2    int player_driver_init(DriverTable* table){
3      ExampleDriver_Register(table);
4      return 0;
5    }
6  }
```

Algoritmo E.35: Inicialización del objeto compartido

Esta función se llamará cuando se cargue el objeto compartido, antes de que se instancie ningún *driver*. El bloque `extern "C"` se usa para evitar las modificaciones en el nombre de la función de inicialización, que haría C++.

[11] La función de construcción del driver desde el punto de vista del servidor de *Player*, se conoce, en inglés, como la *factory function*, de forma similar al patrón de diseño *Factory* o Factoría [Gamma et al., 2003].

E.2.16. Construir una librería compartida

El *driver* de ejemplo incluye un **Makefile** para construir librerías compartidas[12]. Para realizar la construcción manual usaremos los comandos del algoritmo E.36.

```
g++ -Wall -g3 -c exampledriver.cc
g++ -shared -nostartfiles -o exampledriver.so exampledriver.o
```

Algoritmo E.36: Construcción manual de una librería compartida

Aunque el método anterior es correcto y funcionará, es recomendable usar el comando **pkg-config** para obtener los *flags* de compilación. Por tanto, usaremos la invocación alternativa del algoritmo E.37.

```
g++ -Wall -g3 `pkg-config --cflags player` -c exampledriver.cc
g++ -shared -nostartfiles -o exampledriver.so exampledriver.o
```

Algoritmo E.37: Construcción de una librería compartida con **pkg-config**

En cualquier caso, disponiendo del fichero **Makefile** que se muestra en el algoritmo E.38, bastará con ejecutar el comando `make` desde la línea de comandos —tras situarse en el directorio donde estén el fichero **Makefile** y el código fuente del *driver*.

```
DRIVER=exampledriver

all: lib$(DRIVER).so

%.o: %.cc
	$(CXX) -Wall -fpic -g3 `pkg-config --cflags playercore` -c $<

lib$(DRIVER).so: $(DRIVER).o
	$(CXX) -shared -nostartfiles -o $@ $<

clean:
	rm -f *.o *.so
```

Algoritmo E.38: Fichero **Makefile** para construir una librería compartida

Sustituyendo el valor de la variable `DRIVER` se podrá construir la librería u objeto compartido de otro *driver*. En el algoritmo E.38 se ha instanciado esta variable con `exampledriver`, que coincidiría con los ejemplos mostrados en los algoritmos E.36 y E.37.

[12] Por librerías compartidas se entienden las denominadas *shared libraries* o *shared objects*, en inglés. Son librerías u objetos compartidos propios del Sistema Operativo, en este caso *Linux*. En principio se les denominará *librerías*, aunque la terminología de *objetos* también es correcta.

E.3. XML

Extensible Markup Language (XML) is a simple, very flexible text format derived from SGML (ISO 8879). Originally designed to meet the challenges of large-scale electronic publishing, XML is also playing an increasingly important role in the exchange of a wide variety of data on the Web and elsewhere.

— Liam R. E. Quin
2001-2008 (World Wide Web Consortium (W3C), XML Activity Lead)

Definición E.1 (Elemento (XML)). *Un elemento es una construcción del lenguaje XML que contiene la información de los datos representados. Se caracteriza por almacenar estos datos de forma auto-documentada, v. g. si una ristra es el título de un libro, ésta estará almacenada dentro de un elemento denominado <**titulo**>.*

Definición E.2 (Atributo (XML)). *Una etiqueta es simplemente un nombre genérico para un elemento (véase la Definición E.1). Existen dos tipos de etiquetas:*

De Apertura *Etiqueta con la forma <**elemento**>, que se pone al inicio de la especificación de un elemento.*

De Cierre *Etiqueta con la forma </**elemento**>, que se pone al final de la especificación de un elemento.*

Toda la información perteneciente al elemento estará contenido entre las etiquetas de apertura y cierre del elemento.

Definición E.3 (Etiqueta (XML)). *Un atributo es una construcción del lenguaje XML que aparece dentro de la etiqueta de apertura de un elmento.*
Todo atributo debe tener obligatoriamente un valor, de uno de los diversos tipos de datos soportados por XML.

E.4. Validación

La validación XML consiste en comprobar que la sintaxis de un fichero XML es la correcta, de acuerdo con las directrices de un fichero de especificación de dicha sintaxis.

E.5. Documento de Esquema de XML

Los esquemas XML (XSD) son un lenguaje basado en XML que permite especificar la sintaxis de lenguajes basados en XML. Esto permite la validación de los mismos. Se remite al lector al W3C para consultar más detalles sobre este tipo de lenguaje.

E.6. XPath

XPath es un lenguaje de consulta para acceder a información almacenada en ficheros XML. Se puede consultar su sintaxis en el W3C.

E.7. Software

Como la mayoría de los datos que se tendrán para indicar las misiones y componentes del AUV se codificarán con el lenguaje XML, conforme a unas especificaciones o restricciones definidas en ficheros XSD, será necesario implementar o disponer de librerías que permitan trabajar con éstos ficheros.

En este sentido, las tareas habituales con los ficheros XML (de forma genérica) son:

1. Leer el fichero XML para obtener su árbol DOM. Si es posible realizar la construir del árbol es porque el fichero XML está *bien formado*, es decir, el uso de la sintaxis de etiquetas XML es correcto.

2. Validar los ficheros XML respecto a la definición de datos de los mismos, bien con DTDs o con ficheros XSD. En nuestro caso se ha optado por ficheros XSD para especificar, también en XML, como debe ser la multiplicidad de los datos presentes en los ficheros XML con los datos de interés. La validación será correcta si el fichero XML es válido de acuerdo a lo indicado en el fichero XSD, respecto al que se declara seguir sus restricciones o sintaxis.

3. Tareas adicionales con el árbol DOM, como puede ser recorrerlo y obtener los valores de los *elementos* y *atributos*, obtenidos a partir de las declaraciones en XML.

4. Operaciones para guardar o crear ficheros XML desde código, mediante programación.

5. Uso de herramientas adicionales para el uso de los ficheros XML como bases de datos. Es el caso de las *consultas*[13] usando XPath.

Se trata de operaciones bastante estandarizadas y comunes en otros ámbitos. Por ello existen librerías que implementan parte del conjunto de las mismas, conforme a los estándares declarados por el W3C[14].

Dada la complejidad del diseño e implementación de las operaciones necesarias, resulta más interesante el uso de una librería, para lo cual se usará la API que ésta declare. Esto aportará otros beneficios adicionales según la librería que se use.

E.7.1. Librería *libxml2*: *The XML C parser and toolkit of Gnome*

Como su propio nombre indica, se trata de una librería que proporciona un *parse* de XML, escrito en lenguaje C, y otras herramientas. En principio se desarrolló para el entorno gráfico de *Gnome*, pero en realidad es posible usarlo para otros ámbitos y en varias plataformas, entre las que se incluyen *Linux* y *Windows*. Esto proporciona compatibilidad multiplataforma a nuestro software, desde el punto de visto del tratamiento de los ficheros de declaraciones en XML, a la hora de hacer las tareas indicadas previamente.

[13] En español denominamos *consulta* a lo que en inglés se conoce como *query*.
[14]

La librería se conoce con el nombre *libxml2* [Veillard, 2006]. Proporciona un gran número de utilidades, cubriendo las tareas que vamos a necesitar en nuestro sistema. Sólo la validación con ficheros XSD está aún en una fase de desarrollo, si bien se dispone de completo soporte para la validación con ficheros DTD. Esto puede ser un ligero inconveniente, pero en realidad la implementación actual puede ser suficiente para el uso que nosotros vamos a dar a las validaciones.

Hay que añadir que aunque *libxml2* está implementada en lenguaje C, existe una serie de interfaces para otros lenguajes de más alto nivel. Estas interfaces están disponibles en forma de paquetes de las distribuciones *Linux*. Así, para C++ existe la posibilidad de usar una interfaz más cómoda con la librería, que permitiría una integración aún mejor con el sistema del AUV. La compatibilidad multiplataforma no se perdería, si bien se encontraría a aún nivel más bajo que la interfaz para C++.

E.7.1.1. Ejemplo de uso de *libxml2*

La propia librería proporciona ejemplos de uso, con la finalidad adicional de testear el correcto funcionamiento de la misma tras la instalación. Al usar la librería habrá que conocer la API sólo hasta cierto nivel de detalle. Como vamos a trabajar con el árbol DOM, será necesario no sólo conocer las funciones disponibles sino la estructura de dicho árbol y otros tipos de datos declarados en la librería. Además, se tienen que incluir los ficheros necesarios de la librería para el correcto funcionamiento durante la compilación de los algoritmos que desarrollemos; al compilar también resultará necesario indicar opciones para que la librería se incluya perfectamente en la fase de linkado.

E.7.1.2. Utilidades de línea de comandos de *libxml2*: *xmllint*

Existen utilidades de línea de comando, como **xmllint**, que permite acelerar el proceso de desarrollo. Esto facilita la validación de ficheros XML de forma inmediata, sin tener que utilizar librerías de programación.

E.7.1.3. Librería *libxml++*. *Wrapper* C++ para la librería *libxml2*

La librería *libxml++* [Johnson et al., 2006] es un *wrapper*[15] en lenguaje C++ para la librería *libxml2*.

E.8. *log4j*. Configuración del registro del sistema

> *Have you ever witnessed a system failure and spent hours trying to reproduce it?*
> *Infrequently occurring bugs are treacherous and cost tremendously in terms of time, money and morale.*
> *With enough contextual information, most bugs take only minutes to fix. Identifying the bug is the hard part.*
>
> — Ceki Gülcü
> *2002* (Autor de **The Complete *log4j* Manual**)

[15] En español se suele traducir el término *wrapper* como *envoltorio*. Se trata de una API que encapsula a otra API; en este caso, las clases de C++ sirven de envoltorio ya que encapsulan la API en lenguaje C de la librería *libxml2*, proporcionando una interfaz más sencilla y cómoda.

Se remite al lector a la consulta del manual de **log4j** [Gülkü, 2002] para conocer los detalles de este sistema de resgistro o *log*, así como el lenguaje de especificación oficial utilizado para configurarlo.

Hay que señalar que existen dos posibles formatos de especificación de la configuración de estos ficheros de configuración. Para la elaboración de este proyecto se ha hecho de uso de la variante en XML. Ésta se usa para definir el plan de *log* (PdL) que se incluye como parte de la misión.

E.9. XDR. Representación de Datos Independiente

El estándar *eXternal Data Representation* (XDR) se ha usado tal cual se especifica en las RFC. Se ha realizado una implementación en la que se ofrece una interfaz basada en *streams* de C++. Con esto se simplifica enormemente el uso de este formato de serialización. Se remite al lector a las RFC para los detalles de la representación de los datos que usa XDR.

Hay que indicar que XDR es una representación binaria, a diferencia de otras alternativas como *Simple Object Access Protocol* (SOAP), la cual es textual y usa XML.

E.10. Implementación de Tareas de Comunicación

En la aplicación hay un fuerte componente de comunicación, aunque en comparativa con el resto de subsistemas su carga es similar. En cualquier caso, existe una serie de tareas de comunicación en las que la implementación se tendrá que realizar con Sockets.

A continuación se listan los elementos o subsistemas susceptibles de implementación con Sockets para la cumplimentación de sus tareas de comunicación; se indica cómo realizar cada tarea, bien con Sockets directamente, o bien con otras alternativas diferentes.

1. *Control Remoto del AUV* → Desde la aplicación de control y supervisión de la misión que ejecutará el AUV se podrá entrar en modo *Control Remoto*, en el que realmente se entrará en una sesión de un *Intérprete de Comandos*, donde existirá un intérprete encargado de ejecutar los comandos que el usuario indique. Se trata de dos elementos, por un lado el intérprete propiamente dicho y por otro la interfaz de comunicación, como pudiera ser un cliente *telnet* o bien *ssh*, en cuyo caso se tendría encriptación y por tanto seguridad en el acceso por control remoto. En conclusión, podría realizar una implementación similar a la de un cliente y servidor *telnet* o *ssh*, o bien realizar la implementación de un subconjunto de sus funcionalidad. También podría adoptarse una aplicación ya existente con estas funcionalidades, como *OpenSSH*, en el caso de clientes y servidores *ssh*. En este sentido el cliente se encontraría en la aplicación de control de la misión, mientras que el AUV debe disponer del servidor para atender las peticiones de los clientes.

2. *Transferencias de Ficheros* → Se consideran tanto las transferencias de ficheros desde la interfaz de control, lo cual incluye *Planes de Misiones*, *Datos Batimétricos*, etc., como las transferencias efectuadas por el AUV, como pueden ser los *Informes de Estado del AUV*, de acuerdo con lo indicado en la misión. Esto implica una tipología de comunicación cliente/servidor, con clientes tanto en el software de

control de la misión como en el propio AUV. Para esta tarea podría hacer uso del protocolo FTP[16] o la implementación de un subconjunto de sus funcionalidades.

En el caso de la comunicación entre los distintos procesos que intervengan en la aplicación existe la posibilidad de usar los *UNIX Sockets*, en lugar de los *Berkeley Sockets*, orientados a Internet y el protocolo IP. No obstante es considera que el uso de los *Berkeley Sockets* será más apropiado porque se podría permitir que los procesos en cuestión se ejecutarán de forma remota.

E.10.1. API de C para los Berkeley Sockets

La forma directa de programar aplicaciones cliente/servidor con Sockets desde las plataformas o Sistemas Operativos al estilo *UNIX*, como son la gran mayoría de distribucion *Linux*, es con la API de Berkeley Sockets, en lenguaje C normalmente, pues es el usado para la implementación de estos Sistemas Operativos. Este tipo de Sockets son los conocidos como Berkeley Sockets[17] y su API en C es la que se tendría que usar a la hora de programar.

El principal inconveniente de la API es lo tediosa que resulta la implementación de determinadas aplicaciones cliente/servidor, en las que se repite tareas continuamente y el código no es tan fácil de mantener como en el caso de disponer de un diseño e implementación con *POO* (Programación Orientada a Objetos) en C++, para lo que sirve de apoyo el uso de librerías de Sockets ya desarrolladas con estos paradigmas, por lo que la integración con una aplicación que sigue la misma filosofía sería mucho mejor.

E.10.2. API de C para WinSock

En el caso del Sistema Operativo *Windows*, para programar aplicaciones cliente/servidor, con Sockets, en realidad se debe hacer uso de la API de WinSock. Esto hace que ciertos aspectos en la programación varíen, de forma bastante notable en lo referente a cabeceras de inclusión y declaración iniciales de los Sockets (en este caso WinSock). A la hora de desarrollar aplicaciones multiplataforma esto supone un impedimento, si bien en C/C++ es posible que el código fuente sea compatible con los Berkeley Sockets y WinSock, haciendo lo oportuno según el caso de forma separado con macros de precompilación según el Sistema Operativo *host* en el momento de compilar los fuentes para generar los ejecutables de la aplicación desarrollada. Esta labor sigue siendo hasta cierto punto compleja y en el caso de las librerías que se comentan en el apartado **??** ya está realizada y testeada.

E.10.3. Frameworks de Sockets en C/C++

Como alternativa al uso de la interfaces o APIs básicas para el manejo de Sockets, bien en *Linux* con la API de C para los Berkeley Sockets, o bien en *Windows* con WinSock, existe la posibilidad del uso de algunas librerías ya existentes. Según el lenguaje de programación se tendrán unas librerías u otras e incluso en ciertos lenguajes de alto nivel

[16] Protocolo de Transferencia de Ficheros, del inglés *File Transfer Protocol* (FTP), conforme a lo especificado en su RFC.

[17] En el enlace `http://beej.us/guide/bgnet/` se dispone de un manual para la programacion con los Berkeley Socket, que complementa a los manuales del propio Sistema Operativo *Linux*, consultables con el comando *man*

pueden existir pequeñas interfaces a modo de funciones que realizan ciertas tareas con un alto de nivel de abstracción y encapsulamiento. No obstante, aquí nos centraremos en la búsqueda de *frameworks* o librerías en lenguaje C o C++, fundamentalmente éste último.

En este sentido se dispone de varias librerías, de las que a continuación se comentan dos de ellas:

1. *cURL* y *libcURL* → Se trata, en primer lugar, de una herramienta de la línea de comandos para la transferencia de ficheros con la sintaxis URL. Da soporte a múltiples protocolos del nivel de aplicación, como son: FTP, FTPS, TFTP, HTTP, HTTPS, TELNET, DICT, FILE y LDAP. Además, da soporte para certificados SSL y otro tipo de funcionalidades como: HTTP POST, HTTP PUT, subir[18] ficheros por FTP, *proxies*, *cookies*, autentificación con usuario y contraseña (*Basic*, *Digest*, *NTLM*, *Negotiate*, *Kerberos*, etc.), reinicialización[19] de transferencias de ficheros, túneles proxy[20], etc. En segundo lugar, *libcURL* constituye la API de programación. Se trata de una librería multiplataforma gratuita de fácil manejo para transferencias URL del lado del cliente. Dando el soporte antes indicado.

2. Alhem *C++ Sockets* → Se trata de un librería para facilitar el desarrollo de aplicaciones cliente/servidor con Sockets. La principal novedad es que está desarrollada en C++, siguiendo un diseño bastante correcto de la jerarquía de Sockets y los protocolos soportados, entre los que se incluye HTTP básicamente (no hay soporte para tantos protocolos como en *libcURL*). Esta librería se comenta con más detalle en el apartado **??**.

3. *The ADAPTATIVE Communication Environment (ACE™)* → Se trata de una librería de programación de comunicación Orientada a Objetos, desarrollada en C++.[21]

La principal ventaja de este tipo de librerías radica en que facilitan la programación con Sockets y el desarrollo de aplicaciones cliente/servidor. Además, no sólo proporcionan una interfaz para los Sockets básicos del Sistema Operativo, sino que dan soporte para protocolos de capas con mayor nivel de abstracción que la de transporte[22]. La labor de desarrollo de uno de estos protocolos es bastante compleja, por lo que estas librerías

[18] En inglés *upload*.

[19] En inglés *resume*, lo que implica que la transferencia continuará desde el punto por donde iba en el momento en que se interrumpió.

[20] En inglés *proxy tunneling*

[21] *ACE* se publicita, en inglés, como *An OO Network Programming Toolkit in C++*

[22] De acuerdo con el modelo de referencia de Interconexión de Sistemas Abiertos (OSI, *Open System Interconection*), se distinguen las siguientes capas o niveles:

1. *Nivel de Aplicación* → En este nivel se incluyen protocolos como HTTP, FTP, etc. que son a los que se hace referencia cuando se indica que las librerías disponibles dan soporte para los mismos.
2. *Nivel de Presentación* → En este nivel se encuentra protocolos de seguridad como SSL, al que también se da soporte en las librerías, y que es de uso habitual en HTTPS.
3. *Nivel de Sesión*
4. *Nivel de Transporte* → Las librerías dan soportes a los protocolos TCP y UDP a este nivel, tratándose de un protocolo orientado a conexión y no orientado a conexión, respectivamente.
5. *Nivel de Red*
6. *Nivel Físico*

suponen una buena opción en el caso de que sea necesario algún protocolo a nivel de aplicación, como pudiera ser el FTP para el intercambio de ficheros. No obstante, también podrán incluirse aplicaciones como servidores y clientes FTP que realicen las tareas requeridas de forma independiente, pero controlada por la aplicación principal.

E.10.3.1. *Alhem* C++ Sockets

De acuerdo con el autor[23], la librería de C++ Sockets [Hedström, 2006] es:

> Esta es una librería de clases en C++ bajo licencia *GPL*[24] que *mapea* la API de Berkeley Sockets de C, y funciona tanto en algunos sistemas *Unix* como en *Win32*. Las características incluidas, aunque no limitadas son: soporte SSL, soporte IPv6, Sockets TCP y UDP, TCP encriptado, protocolo HTTP, el manejo de errores es altamente configurable, etc. Las pruebas de *testeo* han sido efectuadas en: *Linux* y *Windows 2000*, y algunas partes en *Solaris* y *Mac OS X*.

Se trata de un framework o una librería que permite programar con Sockets desde C++. Se dispone de un amplio conjunto de clases bastante probadas y que hacen mucho más sencilla la programación de aplicaciones cliente/servidor usando Sockets.

Las ventajas que proporciona el *framework Alhem* C++ Sockets a la aplicación que se desarrollará, son fundamentalmente las siguientes:

1. Proporciona un diseño e implementación bien elaborado para la parte relativa al soporte básico de las comunicaciones en el Sistema Operativo, dando soporte multiplataforma (*Linux* y *Windows*, fundamentalmente). Como está realizado en C++, al igual que la aplicación que se desarrollará, la integración será bastante homogénea, incluso en la documentación (como es el caso de la documentación del código con *doxygen*[25]).

Los principales inconvenientes a su uso son:

1. Curva de aprendizaje para conocer la API de *Alhem* C++ Sockets. Este factor es de bajo riesgo, ya que se trata de una librería bien documentada y con múltiples ejemplos. Sólo en el caso de aplicaciones específicas surgirían problemas en el uso, al no disponer de un ejemplo o patrón de uso de las clases del *framework*.

2. Existencia de errores, si bien la disponibilidad del autor es muy buena y ante posibles *bugs* el tiempo transcurrido hasta la corrección o la ayuda para solventarlo suele ser corto de acuerdo con la experiencia de otros usuarios[26].

[23] Anders Hedström, el autor de C++ Sockets y otros proyectos dispone de su web en `http://www.alhem.net`; el *framework* (o librería) de C++ Sockets, los tutoriales y ejemplos están disponibles desde `http://www.alhem.net/Sockets/index.html`

[24] Dispone de una licencia alternativa para el uso de la librería C++ Sockets en aplicaciones de fines comerciales, consultable en `http://www.alhem.net/Sockets/license2.html`.

[25] *Doxygen* es un generador de documentación para varios lenguajes, entre los que se incluye C++, disponible en `http://www.doxygen.org`. La salida de la documentación admite varios formatos, incluido LaTeX.

[26] La experiencia de otros usuarios y los problemas de *Alhem* C++ Sockets se aprecian sustancialmente mejor desde el foro, ubicado en:

`http://bhost.alhem.net/cgi-bin/bbs/viewer?book=538&sheet=3`

3. Imposibilidad de realizar ciertas tareas con las clases proporcionadas en el *framework*, en cuyo caso habría que recurrir a la programación directamente en lenguaje C con la API de los Berkeley Sockets. Se incluye en este punto los problemas de rendimiento derivados de la implementación del *framework*.

4. El software se encuentra bajo una licencia de software libre, en concreto GPL[27], que no permite el uso del *framework* en aplicaciones con fines comerciales. No obstante se dispone de una licencia alternativa para cubrir este caso. En cualquiera de ellos, supone un inconveniente que podría materializarse en aspectos económicos o de derechos de autor.

En conclusión, si el uso de Sockets no es especialmente intensivo en la aplicación, no será necesario el uso de *Alhem* C++ Sockets. Desde el punto de vista de la calidad del software y en especial de la portabilidad, con este *framework* se consigue que esta parte del software tenga soporte multiplataforma (*Linux* y *Windows*, fundamentalmente). En el caso de realizar las implementaciones con Berkeley Sockets, en lenguaje C, realmente se crearán clases en C++ para cubrir las necesidades de la aplicación, por lo que no se perderá la posibilidad de un fácil mantenimiento, incluso para dar soporte multiplataforma.

E.10.3.2. The ADAPTATIVE Communication Environment (ACE™)

Dado que sólo se ha realizado la consulta de este entorno de programación de comunicaciones, se remite al lector al siguiente enlace: http://www.cs.wustl.edu/~schmidt/ACE-overview.html

E.11. Prolog

E.11.1. Interfaz en lenguaje C para trabajar con Prolog

El lenguaje Prolog ofrece una interfaz para C, la cual permite interactuar con Prolog desde una aplicación desarrollada en lenguaje C. Por extensión, en realidad, podremos usar la interfaz desde aplicaciones realizada en C/C++. En cuanto a la plataforma o implementación de Prolog, la mayoría proporciona esta funcionalidad, es decir, una API para programación con otro lenguaje, concretamente C. Sin embargo, nosotros nos centraremos en la implementación de distribución libre conocida con SWI-Prolog[28].

E.11.1.1. API

De acuerdo con el Manual de Referencia de SWI-Prolog [Wielemaker, 2004] disponemos de una API para interactuar con Prolog desde una aplicación codificada en lenguaje

[27] GPL (GNU General Public License) se puede consultar en:

http://www.gnu.org/licenses/gpl.html

[28] SWI-Prolog está disponible en http://www.swi-prolog.org. Existen otras distribuciones de Prolog como GNU Prolog (versión gratuita, disponible desde http://gnu-prolog.inria.fr, con manual [Díaz, 2002]), SICStus Prolog (versión no gratuita, disponible desde http://www.sics.se/isl/sicstuswww/site/index.html, con manual gratuito [Laboratory, 2006]), etc. Las indicaciones de los manuales, respecto al lenguaje Prolog, suelen ser compatibles, si bien es necesario usar la referencia para la interfaz C para Prolog de la distribución que se emplee, en nuestro caso SWI-Prolog.

C. Esta API nos permite realizar las tareas básicas de Prolog, desde C, pero se sigue un esquema bien definido. Por tanto, para interactuar con Prolog, al menos se deben seguir o considerar los siguientes pasos:

1. Crear un programa o base de datos Prolog, con los predicados que deseemos para la parte de nuestra aplicación que se codificará en Prolog.

2. Realizar la aplicación en lenguaje C/C++. Esto incluye la parte de la aplicación que no interactuará con Prolog y la que sí lo hará. La parte que interactua con Prolog se debe desarrollar siguiendo los siguientes pasos:

   a) Incluir la librería con la API de Prolog, que se trata de

   b) Realizar la inicialización de Prolog, que en principio requiere del uso de la función PL_initialise, que puede recibir parámetros al estilo de los parámetros que recibe el punto de entrada main.Opcionalmente o en caso de ser necesario, hay que inicializar o incluir en la inicialización el soporte para *readline*.

   c) Ahora construiremos los predicados y términos de Prolog que necesitemos, para lo cual se realizan las siguientes tareas:

      1) Con la función PL_predicate podremos crear un predicado indicando por parámetro su nombre, aridad[29] y el módulo o entorno en que existirá el predicado[30].
      2) Con la función PL_new_term_refs se crea un determinado número de referencias a términos de Prolog; el número se indica por parámetro. También puede usarse PL_new_term_ref para crear una sola referencia. Al crear varias referencias, el acceso a cada uno se consigue con punteros consecutivos, sumando de uno en uno a la referencia base (ver **??**).
      3) A cada término que se cree se le podrá asignar un valor, el cual puede tener un determinado tipo de datos, o bien no asignarle nada, en cuyo caso será una variable, es decir, un término no instanciado, que mediante el proceso de unificación podrá instanciarse tras la ejecución de algún predicado. Para esto se usa, según el tipo de datos, las funciones que tienen como prefijo PL_put_, como por ejemplo PL_put_atom_chars para crear un término que es un cadena de caracteres (ristra).
      4) Con la función PL_call_predicate se llama a un predicado, creado con PL_predicate, al que se puede pasar la referencia a un término base, que permite el paso de parámetros al predicado en la llamada. Esto produce la ejecución dentro del motor de Prolog.
      5) Tras ejecutar un predicado en Prolog, suele ser normal tomar algún resultado para usarlo en la aplicación desarrollada en C/C++. Para ello, de forma análoga a las funciones de instanciación de los términos según el tipo de datos, se puede tomar un término de Prolog como un tipo de datos, para tenerlo en C/C++. Estas funciones tienen el prefijo PL_get_, como por ejemplo PL_get_integer, que toma un término como un entero.

[29] La aridad es equivalente al número de parámetros de una función; este término es muy común en Prolog, como lenguaje de programación lógica, por hacerse uso de predicados y la semántica asociada.

[30] El entorno suele ser üser", que equivaldría al de una sesión normal en el intérprete de Prolog (el ejecutable prolog, en el caso de SWI-Prolog).

Obviamente, es posible realizar muchas mas tareas, pues la API dispone de muchas más funciones, que pueden consultarse en el Manual de Referencia de SWI-Prolog [Wielemaker, 2004].

E.11.1.2. Implementación. Esqueleto de aplicación C para trabajar con Prolog

Conforme a lo indicado en la sección ??, a continuación se ilustra con un esqueleto del código que se usa en la aplicación implementada en C/C++. Por claridad, dicho esqueleto se divide en partes, según la tarea que se realiza, siguiendo el guión de la lista de la sección ??.

Para empezar, el esqueleto básico, en el que simplemente se realiza la inclusión de la librería con la API, de SWI-Prolog, y la inicialización, sería el siguiente:

La fase de inicialización es precisamente la que aparece en el código dentro de la función principal main. Podría poner o crearse una función que se encargara de esta tarea de inicialización. El único problema, para no limitar su funcionalidad, afecta a los parámetros que se pasan a la función de inicialización PL_initialise.

Tras la inicialización se puede realizar la creación de predicados o términos de Prolog, sin importar el orden en que se haga, salvo en el caso de que el predicado requiere de términos. Para crear un predicado se usa la función PL_predicate, que devuelve un predicado como el tipo de datos predicate_t. Esta función simplemente requiere de la indicación del nombre del predicado, su aridad y el módulo en que se crea el predicado (ver , donde se tiene el ejemplo de creación del predicado llamado "term_to_atom", de aridad dos, en el módulo üser"; el predicado "term_to_atom.^{es} parte de la librería de predicados de Prolog, de modo que lo que se hace es crear un predicado para llamarlo).

Los términos se crean con la función PL_new_term_refs de forma general, indicando el número de términos deseados. Como se muestra en el código se tendrá variables del tipo de datos term_t para contener cada término. El término base se obtiene con la llamada a PL_new_term_refs y el resto se obtiene sumando de uno en uno a la base (que es un puntero o referencia al término) para los siguientes términos. En el código se muestra un ejemplo de creación de dos términos.

Por defecto, los términos creados serán como variables de Prolog, pero con las funciones PL_put_ se podrá instanciar cada término. En el código se muestra como se instancia el término t1 con una ristra.

Si deseamos hacer una llamada a un predicado con aridad no nula, usaremos la función PL_call_predicate, pasando por parámetro el predicado (de tipo de datos predicate_t) y la referencia a los términos (de tipo de datos term_t), que son los parámetros[31]. En el código se muestra la llamada para el predicado y términos antes vistos, en los códigos

Después de la llamada al predicado, con las funciones PL_get_ podremos ver el valor que ha tomado una variable de Prolog tras la llamada. Por ejemplo, podemos tomar el valor de una lista de Prolog o de un entero. En el caso de un entero el proceso es simple y basta con una llamada, ya que el tipo de datos existe en C/C++. Sin embargo, para tomar una lista no ocurre así, y se requiere de un pequeño algoritmo que se encargue de tomar los elementos de la lista. En el código se puede ver como se toma un entero con PL_get_integer.

[31] Cuando se tienen varios parámetros, creados con PL_new_term_refs, basta con poner la referencia base, es decir, la que devuelve ésta función, de modo que se tomarán todas las demás referencias a los términos, por ser consecutivas.

E.11.1.3. Compilación. Librerías necesarias y uso de la aplicación *plld*

A la hora de compilar los fuentes para obtener el fichero ejecutable de nuestra aplicación existen dos alternativas:

1. Usar el compilador gcc directamente y configurar correctamente las librerías a incluir.
2. Usar la utilidad *plld*, que configura de forma automáticamente las llamadas que deben hacerse con gcc. El problema de esta opción es que puede no ser apropiada para la compilación de un proyecto más complejo, donde haya otras librerías a incluir, por ejemplo. No obstante puede servir de pista para usar la otra técnica, si bien se puede hacer manualmente con los datos de las variables de entorno de Prolog, que en el caso de SWI-Prolog pueden obtenerse mediante la orden

Usando el comando *plld* para un fichero llamado operarunidades.c y un fichero de Prolog asociado, con el nombre operarunidades.pl, tendremos el resultado de

La opción -pl es necesaria para indicar el nombre del ejecutable de Prolog, que en el caso de SWI-Prolog es prolog. Luego se indica el nombre del ejecutable (operarunidades) que deseamos obtener y los ficheros en lenguaje C/C++ y Prolog, respectivamente (operarunidades.c y operarunidades.pl, respectivamente, en la salida).

Con la opción -v (ver la salida , donde se ve que se ejecutan comandos adicionales a los que usan gcc, pero que en principio no se requieren) se verá una salida extendida, que permite ver como se deben construir las llamadas a gcc, que serían las que tendríamos que ejecutar para obtener el mismo y deseado resultado, como se muestra en la salida

E.12. Sintaxis de ejemplos de la tipología de misiones

La sintaxis usada para mostrar ejemplos de las tipologías de misiones (véase el Capítulo 11) ha sido la que mejor se ha adaptado a las características de éstas. Se trata de un subconjunto de C/C++, pues es bastante conocido, fácil de entender y apropiado en este caso. Para la definición de valores vectoriales se usa la sintaxis de Matlab por ser la más concisa para ello. Además, se indican las unidades, siempre que procede, en lenguaje natural acompañando al valor.

Los elementos usados son:

Comentarios Se usa el comentario de línea de C++: //.

Sentencias Las sentencias se escriben en línas separadas, como en Matlab, por lo que no se usa el punto y coma, como en C/C++.

Declaraciones Se indica el tipo de dato y su nombre para realizar la declaración de una instancia, como se hace en C/C++, pero sin definir previamente el tipo, como en Matlab —v. g. **Area area**.

Comandos Los comandos tienen las sintaxis de las llamadas a funciones en C/C++, con el nombre del comando y la lista de parámetros entre paréntesis y seperados por coma, v. g. **Recorrer(area, modo, resolucion, profundidad)**.

Estructuras y campos En muchos casos se hace uso de estructuras y con el punto se accede a campos internos de las mismas. Los campos pueden ser a su vez estructuras, pudiéndose construir jerarquías de datos.

Valores dimensionales O valores con unidades, tiene el valor y el nombre de la unidad en lenguaje natural. Los valores adimensionales no se acompañana de ninguna unidad. Para valores especiales como el porcentaje se usa el símbolo típico, v. g. %.

Vectores Se indexan con corchetes y desde 1, v. g. **condicion[1]**. Los literales se muestran con la sintaxis de Matlab, i. e. con los valores entre corchetes y separados por comas, v. g. $[10, 20, 4]$.

Ristras Las ristras se escriben directamente, sin necesidad de usar comillas, v. g. **temperatura**.

Valores singulares O valores límites, como infinito o números no representables —i. e. **NaN** (Not a Number). Se indican directamente con el texto de los mismos, v. g. **infinito**, **NaN**; se escribirán facilitando la lectura en su contexto.

La sintaxis empleada es flexible a consecuencia de su finalidad. Para una comprensión detallada se remite al lector al Capítulo 13 para consultar el formalismo empleado en la definición de misiones (véase también el Capítulo 12 para información más detallada sobre la arquitectura de definición de misiones).

E.13. Esquemas XML de la especificación formal de la misión

> *XML Schemas express shared vocabularies and allow machines to carry out rules made by people.*
> *They provide a means for defining the structure, content and semantics of XML documents.*
>
> — C. M. Sperberg-McQueen and Henry S. Thompson
> *2000-2008* (Miembros del World Wide Web Consortium (W3C))

Par cada plan de la misión se utiliza un esquema XML que especifica la sintaxis del lenguaje del mismo. Estos ficheros son muy extensos y no se incluirán en este documento. Respecto a la sintaxis de XSD, puede consultarse lo comentado en la Sección E.3.

E.13.1. Dirección

De acuerdo con lo comentado en la Sección 13.4 las direcciones especificadas en los planes de la misión tiene la sintaxis definida por las expresiones regulares mostradas en el Algoritmo E.39 y la gramática BNF del Algoritmo E.40 —i. e. definición léxica y sintáctica, respectivamente.

```
1 # BYTE debe estar en el intervalo [0, 255]
2 BYTE: 0 | [1-9][0-9]*
3 # PUERTO debe estar en el intervalo [1, 65535]
4 PUERTO: [1-9][0-9]*
5 HOST: [a-zA-Z][a-zA-Z0-9_]+
```

Algoritmo E.39: Dirección. Expresiones regulares

```
ip: BYTE '.' BYTE '.' BYTE '.' BYTE ;
direccion: ip | HOST ;
direccionPuerto: direccion | direccion ':' PUERTO ;
direcciones: direcciones ',' direccionPuerto | direccionPuerto ;
```

Algoritmo E.40: Dirección. Gramática BNF

E.13.2. Parámetros de la misión. Direccionamiento de atributos con XPath

Una sentencia o *query* XPath permite referenciar elementos o atributos XML del árbol XML de la misión, tal y como se explica en la Sección 13.9, donde se comenta la especificación de los parámetros de la misión y su utilidad en la modificación dinámica de los diferentes compoentes de ésta. El uso de XPath es especialmente apropiado porque permite la especificación de consultas sobre XML de acuerdo a un estándar contrastado (véase la Sección E.6). Como se ha usado XML como lenguaje para la especificación de la misión (véase el Capítulo 13), esto aportará otras ventajas, como la posibilidad de acceso y modificación de atributos XML al editar la misión desde el **planificador** (véase la Sección 12.3 y Apéndice A, respectivamente).

```
/planDeMedicion[@id='7']/tarea[@id='2']/@nombre
```

Algoritmo E.41: Direccionamiento de un atributo con XPath

```
/mision[@id='3']/planDeMedicion[@id='7']/tarea[@id='2']/@nombre
```

Algoritmo E.42: Direccionamiento indicando la misión

En el Algoritmo E.41 se muestra un ejemplo de sentencia XPath para la modificación de una atributo de los planes de la misión (véase la Sección E.6 para consultar los detalles relativos a la sintaxis de XPath). En este caso, leída de izquierda a derecha, se interpreta de la siguiente forma:

1. **/planDeMedicion[@id='7']** Referencia el PdM cuyo atributo **id** vale 7. Los planes de la misión se buscarán en el árbol de directorios definido en el Cuadro 13.1. El primer elemento referenciado en la sentencia XPath será un plan de la misión, aunque adicionalmente puede indicarse una misión concreta primero —v. g. en el Algoritmo E.42 se muestra el mismo direccionamiento que en el Algoritmo E.41, pero indicando que se trata del PdM de la misión cuyo atributo **id** vale 3.

2. **/tarea[@id='2']** Referencia el elemento **tarea** cuyo atributo **id** vale 2 —dentro del PdM 7.

3. **/@nombre** Referencia el atributo **nombre** —dentro de la tarea 2.

En el PdS se puede usar el comando **CambiarVariableMision** dentro de la acción **ejecutarComando** para reconfigurar algún componente de la misión (véase la Sección 13.7). Esto se puede hacer directamente con una sentencia XPath o indicando el nombre de un parámetro de la misión. En el segundo caso se usará la arroba **@** como prefijo para facilitar la tarea de interpretación del PdS (véase también la Sección 13.9).

Apéndice F

Criterios de evaluación de la especificación de la misión

A la hora de evaluar o valorar la calidad de las diferentes arquitecturas de especificación de misiones tratadas en el estudio de la Sección 12.1, se ha determinado una lista de criterios que se han estimado relevantes. La lista consta de los siguientes criterios: **modularidad**, **flexibilidad**, **monitorización**, **control remoto**, **facilidad de definición**, **portabilidad**, **reconfiguración** y **aprendizaje**, explicados en la Definición F.1, Definición F.2, Definición F.3, Definición F.4, Definición F.6, Definición F.7, Definición F.8 y Definición F.9, respectivamente.

Definición F.1 (Modularidad). *Propiedad por la cual es posible añadir nuevos elementos sin tener que modificar el resto de elementos.*
En el caso de la especificación de misiones se concreta en la posibilidad de añadir o modificar especificaciones concretas sin que ello afecte al resto de componentes de la misión —i. e. planes, tareas u otro tipo de especificación.

Definición F.2 (Flexibilidad). *Capacidad para adaptarse a diferentes circunstancias sin que ello produzca "efectos secundarios" que inutilicen el resto del sistema.*
Una especificación de misiones será flexible si puede adaptarse a diferentes tipos de misiones fácilmente y sin que ello comprometa o invalide (véase la Definición 12.7) la misión.

Definición F.3 (Monitorización). *Propiedad que permite la observación y supervisión del estado de ejecución o realización de un determinado proceso. Esto permite determinar que está encaminado adecuada y eficazmente hacia el resultado final deseado.*
En una especificación de misiones monitorizable se dispone de una secuencialidad clara o de una segmentación de sus componentes que permite saber hasta qué punto de la misión se ha llegado —i. e. el grado de realización de toda la misión o de cada uno de sus componentes.

Definición F.4 (Control remoto). *Propiedad que permite que un sistema sea manipulado a distancia —i. e. controlado remotamente.*
Una especificación de misiones controlable remotamente debe estar definida de forma que

sea reconfigurable (véase la Definición F.8). El control remoto propiamente dicho lo proporcionará el sistema (véase la Sección 15.4), que permitirá la manipulación de la misión durante su ejecución.

Definición F.5 (Comando). *Propiedad que permite manipular un sistema mediante indicaciones que se denominan comandos (véase la Definición 15.1); cf. control remoto. Respecto a la especificación de misiones, ocurre lo mismo que con el control remoto (véase la Definición F.4).*

Definición F.6 (Facilidad de definición). *Cualidad por la cual una especificación o definición resulta cómoda e intuitiva para un usuario que domine la semántica de la misma sin tener que conocer detalles adicionales, v. g. lenguajes de programación, procedimientos de especificación.*
Una especificación de misiones fácil de definir se apoyará en una Interfaz Gráfica de Usuario (GUI) para modelar la misión, que manejará una representación textual de la misma de forma transparente al usuario.

Definición F.7 (Portabilidad). *Grado de dependencia de la plataforma en la que se ejecuta o trabaja con un software. La portabilidad es mayor cuanto menor es la dependencia con la plataforma.*
En el caso de la especificación de misiones, la portabilidad se centra en las herramientas disponibles para la especificación, validación e interpretación de la misión[1] *(véase la Sección 12.3). La especificación y validación la realizará el planificador (véase la Apéndice A), mientras que la interpretación la realizará el sistema (véase la Parte III). Estas herramientas software se denominan multiplaforma porque pueden usarse en diversas plataformas.*

Definición F.8 (Reconfiguracion). *Capacidad de modificar la configuración original, i. e. los parámetros que definen un sistema o especificación.*
Una especificación de misiones reconfigurable dispondrá de parámetros que modelan la misión y sus componentes, los cuales pueden modificarse durante la ejecución de la misión.

Definición F.9 (Aprendizaje). *Dentro del campo de la IA, el aprendizaje es un campo cuyo objetivo consiste en desarrollar técnicas que permitan* aprender *al software. Es la capacidad de generalizar comportamientos a partir de información no estructurada suministrada en forma de ejemplos.*[2]
En lo relativo a la especificación de misiones, el aprendizaje formará parte del sistema —integrado como un módulo. La misión debe ser reconfigurable (véase la Definición F.8) para que el sistema pueda modificarla automáticamente en base a un algortimo de aprendizaje que se alimentará de la información sensorial (véase el Sección 15.7) e intentará ajustar los parámetros de configuración de la misión o algunos de sus componentes, para que los efectos de éstos sobre el entorno sean los más apropiados.

[1] La misión *per se* es independiente de la plataforma, ya que es interpretada. Por este motivo, la portabilidad evaluable será la del intérprete que se necesita.
[2] Esta definición se corresponde con lo que se conoce como aprendizaje inductivo —i. e. por inducción.

Apéndice G

Herramientas Software. Librerías y *Frameworks*

G.1. Librerías de Programación en C++. *CoreLinux++* y otras

A la hora de desarrollar el análisis, diseño e implementación de una aplicación software, mediante el paradigma de la Orientación a Objetos y el uso de Patrones de Diseño, es conveniente considerar el uso de librerías ya desarrolladas que cumplen estas características. Estas librerías o *frameworks* servirían de base para el desarrollo de la aplicación, fundamentalmente en la codificación o implementación. Será en este punto donde proporcionen una serie de clases ya implementadas conforme a los paradigmas de Orientación a Objetos y Patrones de Diseño, lo que constituye una estandarización, la cual facilitará el desarrollo y la comprensión del código fuente por terceros. Además, en las fases de análisis y diseño se podrán aplicar técnicas con vistas al uso de los mencionados paradigmas.

Como en nuestro caso trabajaremos con C++ como lenguaje base para la implementación de la aplicación, la librería deberá estar desarrollada para C++. Este es el caso de *CoreLinux++* [Castellucci et al., 2006].

No obstante, aunque el uso de una librería facilita y reduce el tiempo de desarrollo, normalmente sólo se usará un subconjunto de las funcionalidades ofrecidas por la librería. Por ello, siempre es posible tomar parte de la misma o simplemente implementar las partes necesarias, con una filosofía similar a la de este tipo de librerías, pero con un caracter más pragmático.

G.1.1. Otras librerías

Otras librerías destacables para la integración de diferentes funcionalidades en C++, es **Boost** [Dawes et al., 2006]. Se ha usado especialmente las macros de preprocesador que ésta proporciona. Se invita al lector a visitar la web de esta librería para más información y estudio de las librerías que ofrece.

G.1.2. Librerías para gestión de procesos e hilos. *POSIX Threads*

Para la gestión de hilos en C++ se requiere el uso de una librería de los *POSIX Threads*. No obstante, en la implementación realizada del sistema sólo se usan algunos elementos puntuales, ya que se aprovecha la interfaz que ofrece CoolBOT.

G.2. *Frameworks* para arquitecturas híbridas

A la hora de desarrollar el sistema del AUV será interesante observar otros diseños. Existen ciertas arquitecturas desarrolladas a modo de *frameworks*, donde la facilidad de integración de nuevos componentes es la principal virtud. En estos sistemas se dispone de una representación bastante apropiado del modelo de un sistema robótico, donde se debe integrar gran variedad de sensores y actuadores de heterogéneas características. En este sentido, el tipo de arquitectura más apropiada es la híbrida.

En lo sucesivo se mostraron los casos más significativos de estos *frameworks*, destacando sus principales características.

Entre los *frameworks* más representativos, utilizados en este proyecto y que forman parte del sistema *Sick***AUV**, se encuentran *Player* y CoolBOT [Gerkey et al., 2006, Domínguez Brito, 2003].

Índice de figuras

Índice de cuadros

Índice de teoremas

Índice de algoritmos

Glosario

4GL 4^{th} Generation Language — Lenguaje de 4ª Generación 279

ACE ADAPTIVE Communication Environment — Entorno de Comunicación ADAPTATIVO. *Framework* para la programación de aplicaciones que se comunican a través de *sockets*, que emplea patrones de diseño 336
AI Aproximación Incremental. Paradigma de Desarrollo del software 17
API Application Programming Interface — Interfaz de Programación de Aplicaciones. Conjunto de funciones y procedimientos que ofrece una librería software 24
ASC Autonomous Surface Craft — Embarcación de Superficie Autónoma 43
ASCII American Standard Code for Information Interchange — Código Estadounidense Estándar para el Intercambio de Información 118
ASV Autonomous Surface Vehicle — Vehículo de Superficie Autónomo 43

BD Base de Datos — DataBase 332
BNF Backus Normal Form o Backus-Naur Form — Forma Normal de Backus ... 60
bottom-up lit. abajo-arriba. Modelo o estrategia de diseño donde las partes individuales se diseñan en detalle y luego se enlazan para formar componentes mayores y así sucesivamente hasta obtener el sistema completo 162

CF Compact Flash 30
CMS Content Management System — Sistema de Gestión de Contenido 25
CoolBOT A Component-Oriented Programming Framework for Robotics — Un Marco de Programación Orientada a Componentes para Robótica 299
CTAN the Comprehensive TEX Archive Network — la Red de Archivos TEX Global. Ofrece documentación y paquetes LATEX 26
CTD Conductivity Temperature Deep — Conductividad Temperatura Presión. Sensor integrado de conductividad, temperatura y presión; también permite el cómputo matemático de la salinidad a partir de éstas 53

DOF Degrees Of Freedom — Grados De Libertad 60
DOM Document Object Model — Modelo de Objectos para la representación de Documentos. Es un modelo computacional que permite acceder y modificar de forma dinámica el contenido, estructura y estilo de documentos como XML 333
DSL Domain Specific Language — Lenguaje de Dominio Específico 279
DTD Document Type Definition — Definición del Tipo de Documento 333

DVI DeVice Independent — Independiente del DisPositivo 26

EBNF Extended BNF — BNF Extendido 60
ecosonda Dispositivo similar al sonar aunque no suele operar de forma automática, suele ser un equipo portátil y su transductor no es fijo —i. e. puede operar en varias posiciones 80
EPS PostScript encapsulado 26

FLWOR Las expresiones FLWOR toman su nombre de los cinco tipos de sentencias de consulta de las que pueden estar compuestas: **FOR**, **LET**, **WHERE**, **ORDER BY** y **RETURN** 333
FSL Forward Looking Sonar — Sonar Frontal 76
FTP File Transfer Protocol — Protocolo de Transferencia de Ficheros 25

GCC GNU Compiler Collection — Colección de Compiladores de GNU 22
GIMP GNU Image Manipulation Program 26
GPL General-Purpose programming Language o Generic Programming Language — Lenguaje de programación de Propósito General 279
gradiente Dirección en el espacio en la que se aprecia una variación de una determinada propiedad o magnitud física; se expresa y representa con un vector 74
GUI Graphical User Interface — Interfaz Gráfica de Usuario 97

HAL Hardware Abstraction Layer — Capa de Abstracción Hardware 215
helical lit. helicoidal. Con forma de hélice o espiral. Tipo de recorrido o maniobra de exploración de un área 71

IA Inteligencia Artifical — Artificial Intelligence. Disciplina que se encarga de construir procesos que producen acciones o resultados que maximizan una medida de rendimiento determinada, basándose en la secuencia de entradas percibidas y en el conocimiento almacenado por la arquitectura física subyacente 346
IDE Integrated Development Environment — Entorno de Desarrollo Integrado ... 22
ITOCA Intelligent Task-Oriented Control Architecture — Arquitectura de Control Inteligente Orientada a Tareas 171

lenguaje de consulta Lenguaje usado para realizar consultas en bases de datos y sistemas de información, i. e. obtener datos de éstos 333
log lit. registro, diario. Se trata del proceso de registro de información de un sistema, que habitualmente es guardada en ficheros en disco y que contiene información de depuración o monitorización del estado y evolución del sistema 155
LOP Language Oriented Programming — Programación Orientada al Lenguaje 280

mooring lit. amarrado. Proceso de anclaje —i. e. amarrado— de un vehículo u objeto. Suele aplicarse a las boyas oceanográficas 43
mowing lit. segando. Tipo de recorrido de exploración de un área equivalente al de *zigzag*. Maniobra de realización de este tipo de recorrido 71
multibeam lit. multihaz. Característica o tipología de sonar que emplea múltiples haces y suele usarse para determinar la profundidad del fondo marino —batimetría . 80
multiplataforma Software ejecutable en varias plataformas (cf. portabilidad) .. 346

NOAA National Oceanic and Atmospheric Administration — Administración Oceánica y Atmosférica Nacional 4

OBTLC Object-Based Task Level Control architecture — arquitectura a Nivel de Tareas Basada en Objetos. Esta arquitectura dispone de tres niveles: *servo*, tareas y organización 163
odometría Estudio de la estimación de la posición de un determinado vehículo durante la navegación 80

parser Analizador sintáctico 60
Payload Carga útil. Equipamiento o tareas que forman parte de la finalidad última de la misión —que normalmente produce resultados, v. g. muestras de parámetros físico-químicos 60
PCMCIA Personal Computer Memory Card International Association 30
PdA Plan de Almacenamiento 107
PdC Plan de Comunicación 108
PDF Portable Document Format — Formato Portátil de Documento 26
PDL Page Description Language — Lenguaje de Descripción de Página 26
PdM Plan de Medición 108
PdN Plan de Navegación 108
PdS Plan de Supervisión 108
pipeline lit. tubería. Tubería o conducto para transportar algo —v. g. gas, electricidad, telecomunicaciones. 76
PostScript PDL que es usado en impresoras como formato de transporte de archivos gráficos 26
Prolog Programation et Logique — Programación y Lógica. Lenguaje de programación lógico e interpretado muy común en la IA 117

query lit. consulta. Consulta a una BD. Los elementos y atributos de un fichero XML pueden considerarse como una BD y se pueden consultar con XPath 333

R/V Research Vessel — Buque de Investigación 43
RAE Diccionario de la Real Academia Española 68
RAW lit. crudo. Datos o formato original, i. e. no procesado 76
RdP Red de Petri 287
RL Reinforcement Learning — Aprendizaje por Refuerzo 102
ROV Remote Operated Vehicle — Vehículo Operado Remotamente. Robot submarino no tripulado y conectado a un barco en superficie por medio de un umbilical .. 43

scanner Analizador léxico 60
Schematron Lenguaje basado en reglas de validación para realizar afirmaciones sobre la presencia o ausencia de patrones en XML 334
SDB Sensor Data Bus — Bus de Datos Sensoriales 172
SOAP Simple Object Access Protocol — Protocolo de Acceso Simple a Objetos. Protocolo que define cómo dos objetos en diferentes procesos pueden comunicarse por medio del intercambio de datos XML 335

sonar SOund Navigation And Ranging — Navegación Y Alcance por SOnido. Técnica de localización acústica; también alude al equipo empleado para generar y recibir el sonido 76
Sonar Multihaz Sonar que proporciona información de profundidad en exploraciones batimétricas —en lugar de imágenes, como un SSS. También existen las ecosondas multihaz 80
SONQL Semi-Online Neural-Q_learning. Algoritmo de RL propuesto por [Carreras Pérez, 2003] 102
SPA Sense Plan Act — Percepción Planificación Acción 162
SQL Structured Query Language — Lenguaje de Consulta Estructurado. Lenguaje declarativo de acceso a bases de datos relacionales que permite especificar cierto tipo de operaciones sobre las mismas, v. g. consultas 333
SSH Secure SHell — TErminal Seguro 25
SSS Side Scan Sonar — Sonar de Barrido Lateral 76
STDL SAUVIM TTDL — TDL de SAUVIM 99
STL Advanced Configuration and Power Interface — Interfaz Avanzada de Configuración y Energía 24
STL Standard Template Library 24
survey lit. exploración, inspección, reconocimiento. Tarea o tipología de misión específica consistente en la exploración de una determinada zona; se trata de una tipología de misión de exploración de área 99

TCM Task Control Management — Manejo de Tareas de Control 99
TD Temporal Difference — Diferencias Temporales. Metodología de algortimos que permite solucionar el problema del RL 102
TDL Task Description Language — Lenguaje de Descripción de Tareas 99
TDLC TDL Compiler — Compilador de TDL 99
tether lit. atar. Acción de unir un vehículo submarino con un umbilical 60
tipología lit. estudio de los tipos. Estudio y clasificación de los tipos que se practica en una determinada ciencia 65
top-down lit. arriba-abajo. Modelo o estrategia de diseño donde se formula un resumen del sistema sin especificar los detalles, para posteriormente ir refinando cada elemento con mayor detalle. Se hace uso de **cajas negras** para representar los elementos aún no detallados 162
transecto Línea o tramo que une un par de *waypoints* 72

UAV Unmanned Aerial Vehicle — Vehículo Aéreo No tripulado 43
UCS Universal Character Set — Conjunto de Caracteres Universal 26
UGV Unmanned Ground Vehicle — Vehículo Terrestre No tripulado 43
umbilical Cable empleado para mantener la conexión y operar remotamente vehículos submarinos —habitualmente un ROV 43
Unicode Universal code — Código universal. Sistema de codificación de caracteres, recogido como subconjunto de UCS 26
untethered Sin umbilical —lit. desatado 60
UTF-8 8-bit Unicode Transformation Format. Se trata de una norma de transmisión de longitud variable para caracteres codificados utilizando Unicode 26

UUV Unmanned Underwater Vehicle — Vehículo Submarino No tripulado 43
UUV Unmanned Underwater Vehicle 5
UUV Unmmaned Untethered Vehicle 60
UV Unmanned Vehicle — Vehículo No tripulado 9

W3C World Wide Web Consortium — Consorcio *World Wide Web* 333
waypoint lit. punto del camino. Punto de paso en el recorrido de una ruta 68
WYSIWYG What You See Is What You Get — Lo Que Ves Es Lo Que Obtienes. Se aplica a los procesadores de texto y otros editores de texto que permiten escribir un documento viendo directamente el resultado final 26

XDR eXternal Data Representation — Representación de Datos Independiente. Protocolo de representación de datos que permite la transferencia de éstos entre diferentes arquitecturas 335
XML eXtensible Markup Language — Lenguaje de Marcas eXtensible 332
XPath XML Path Language — Lenguaje que permite construir expresiones que recorren y procesan un documento XML 333
XQuery Lenguaje de consulta diseñado para consultar colecciones de datos XML. Utiliza expresiones XPath para acceder a determinadas partes de los documentos XML y también añade expresiones similares a las usadas en SQL, conocidas como FLWOR 333
XSD XML Schema Document — Documento de Esquema de XML 332

yoyo lit. yoyó. Tipo de recorrido de exploración de un área equivalente al de *zigzag*. Maniobra de realización de este tipo de recorrido 71

zigzag Voz originaria del alemán *zick-zack*, que da la idea de sinuosidad o serpenteo — normalemente en forma de **Z**— y se utiliza para indicar el comportamiento dinámico de un objeto móvil sobre una ruta tortuosa, llena de curvas 71

Índice alfabético