Universidad
Carlos III de Madrid

Departamento de Informática

PROYECTO FIN DE CARRERA

IMPLEMENTACIÓN DE UNA ARQUITECTURA DE CONTROL HÍBRIDA PARA ROBOTS AUTÓNOMOS

Autor: José Luis Díaz Cebrián

Tutor: Agapito Ledezma Espino

Leganés, julio de 2011

AGRADECIMIENTOS

A mis padres y a mi hermana por su apoyo durante todos estos años. Por estar ahí en los momentos difíciles, comprenderme y no presionarme cuando eso era lo último que necesitaba.

A mi tutor, Agapito Ledezma, por ofrecerme un proyecto tan interesante y completo para poner fin a mi formación académica, así como por su ayuda y comprensión.

A Isaac Ruiz, por su experiencia y su ayuda con el robot. Espero que nos volvamos a ver pronto.

A todos los compañeros de carrera con los que he trabajado codo con codo en alguna práctica o trabajo. Especialmente a Carmen y Alfredo.

Y, por supuesto, gracias a mis amigos por todo lo vivido durante estos años. Por aguantarme, apoyarme y estar en lo bueno y en lo malo. Ángel Luis Torres, Gemma Zarza, Juan María Egido, Marta Poveda y Rubén Fernández. Muchísimas gracias.

RESUMEN

La robótica es un campo de la informática que se encuentra muy de moda actualmente. Aunque los medios de comunicación nos muestran unos avances realmente impresionantes, la verdad es que es un terreno en el que no se ha profundizado demasiado, y en el que es complicado solucionar problemas que aparentemente son muy básicos.

Este proyecto aborda el tema de la navegación básica de robots móviles autónomos, en concreto de un robot Pioneer 3-DX. Se trata de diseñar e implementar una arquitectura haciendo uso del paradigma híbrido de la robótica, que une los paradigmas deliberativo y reactivo, para asentar las bases de un marco de trabajo que permita dotar de comportamientos inteligentes a estos robots móviles.

El diseño de la arquitectura busca la creación de este marco, que sea lo más modular, extensible e independiente de las aplicaciones cliente, para poder incorporar nuevas funcionalidades y convertirse en un proyecto de investigación que se alargue y utilice en el futuro, no limitándose a resolver únicamente el problema planteado inicialmente.

ABSTRACT

Nowadays, one of the most famous and interesting fields in technological sciences is robotics. Mass media show incredible advances on this field. However, that doesn't adjust to reality. This is a field which has not been deepened, and it's very difficult to solve problems that they look relatively easy.

This project deals with the problem of basic navigation in autonomous mobile robots, specifically a Pioneer 3-DX robot. The objective of the project is designing and developing a robot architecture, by using the hybrid paradigm in robotics. This architecture shall combine deliberative and reactive paradigms in order to establish a framework which allows providing mobile robots with intelligent behaviours.

The hybrid architecture designed must create this framework, being as modular, extensible and client-application independent as possible, in order to provide new functionalities and become a researching project into the future, so it doesn't just confine to solve the initial problem of basic navigation.

ÍNDICE GENERAL

ÍNDICE DE TABLAS

ÍNDICE DE FIGURAS

Capítulo 1

Introducción y objetivos

1.1. Introducción

La robótica es uno de los campos en informática más jóvenes y que más interesan en la actualidad. Cada cierto tiempo se publican nuevas noticias relacionadas con estas tecnologías que nos acercan los avances realizados en este campo. Sin embargo, la realidad está bastante lejos de ser como nos llega por los medios de comunicación.

Actualmente la robótica es una rama que se encuentra en una fase muy temprana de investigación. La visión general que tiene la sociedad sobre la robótica está asociada a robots humanoides capaces de mostrar comportamientos muy complejos y casi humanos, lo que lleva a pensar que en poco tiempo llegarían a sustituir a trabajadores humanos en muchos campos de trabajo. Esta visión no es del todo real. Actualmente, pocos robots son capaces de mostrar comportamientos realmente inteligentes. Incluso el hecho de que un robot pueda moverse de forma completamente autónoma por un entorno, sin necesidad de ser teleoperado por un ser humano, y que lo haga de forma eficaz y eficiente, es un problema complejo que todavía se sigue estudiando.

Los robots móviles autónomos son uno de los sectores principales en el campo de la robótica actual. Es una de las líneas más comunes de investigación, en la que se estudian los comportamientos inteligentes de estos pequeños o grandes robots, y se busca que estos robots móviles alcancen una autonomía total que les permita poder navegar por un entorno de la forma más inteligente posible.

Este problema incluye muchos matices nada triviales de solventar, tales como la localización y posicionamiento dentro del entorno. Además, los entornos habitualmente son dinámicos, de forma que no sirve demasiado que un robot móvil autónomo sea capaz de moverse por un entorno completamente controlado, del cuál conoce todo. Todos estos problemas hacen de este sector un punto de partida fundamental en la investigación robótica, ya que sin un movimiento completamente autónomo, un robot no puede llevar a cabo gran cantidad de tareas de más alto nivel.

Así pues, muchos estudios e investigaciones tienen como objetivo que los robots móviles autónomos puedan desplazarse por entornos complejos y dinámicos sin ninguna ayuda de las personas. Para poder alcanzar estos objetivos, los robots necesitan actuar de forma deliberada, creando planes y estrategias para cumplir con las trayectorias necesarias que les permitan alcanzar sus metas. Pero esta forma de actuar no basta, puesto que el robot puede enfrentarse a situaciones inesperadas, de forma que también tiene que ser capaz de reaccionar a las condiciones que se den en su entorno en cada momento.

Este comportamiento que se espera de los robots móviles autónomos aúna los dos paradigmas clásicos de la robótica, lo que hace que la adquisición de comportamientos inteligentes por parte de estos robots sea aún más complicada: el robot debe tener en cuenta el pasado, el presente y el futuro de su ejecución. Por todas estas cosas, la navegación básica de un robot móvil autónomo no es un problema trivial, ni algo que esté completamente estudiado y solucionado en el campo de la robótica.

Este proyecto se enmarca, por tanto, dentro del ámbito de la navegación de robots móviles autónomos. Este proyecto pretende estudiar y dar solución al problema de la navegación básica de un robot móvil autónomo, y pretende hacerlo dotando al robot de comportamientos que incluyan la deliberación de sus movimientos y la reactividad frente a las condiciones actuales del entorno en cada momento.

A lo largo de este documento se detalla todo el trabajo realizado en este proyecto, consistente en la implementación de una arquitectura de control automático para un robot móvil autónomo, con el objetivo de dotar al robot de comportamientos inteligentes de navegación básica que le permitan desenvolverse por un entorno más o menos complejo y con cierto nivel de dinamismo, y con la intención de que lo haga de la mejor manera posible.

1.2. Objetivos

En el apartado anterior ya se ha dejado constancia de en qué consiste el proyecto, y por tanto, los objetivos que se consiguen con su desarrollo. Tal y como indica el propio título del proyecto, el objetivo principal es el de diseñar e implementar

una arquitectura híbrida para un robot Pioneer 3-DX. Es decir, se busca crear un marco de desarrollo para elaborar controladores automáticos para un robot, que le permita mostrar comportamientos inteligentes, y que estos controladores trabajen tanto a nivel deliberativo como reactivo.

Esta arquitectura híbrida debe asentar unas bases para que posteriormente se puedan incluir nuevos comportamientos y funcionalidades. De forma que a raíz del objetivo principal del proyecto, aparecen otros objetivos secundarios que son también muy importantes, y que están presentes en cada fase del proyecto, desde la identificación de requisitos hasta la implementación, pasando especialmente por el diseño de la arquitectura. Estos requisitos secundarios son la sencillez, extensibilidad, modularidad y robustez de la arquitectura diseñada e implementada. Todo ello para facilitar trabajos futuros sobre las bases de la arquitectura implementada, y que este trabajo no sea arduo para los desarrolladores que hagan uso de ella y les permita incluir nuevos módulos y funcionalidades al robot.

Además, otro objetivo fundamental del proyecto es que la arquitectura sea lo más independiente posible de la plataforma (hardware o software) que se utilice para ejecutarla, ya sea ésta un simulador o un robot moviéndose en un entorno real. También se busca documentar los diversos resultados obtenidos en las distintas plataformas probadas y documentar las diferencias que existen entre utilizar unas u otras.

Debido a la naturaleza del problema y a los objetivos que se desean lograr con la arquitectura, se trata de un trabajo complejo, de forma que se ha dejado cierta libertad a la hora de implementar comportamientos inteligentes en el robot. Por ello, se ha limitado a buscar unos comportamientos inteligentes de navegación básica, tales como mantener una trayectoria, esquivar obstáculos o seguir contornos. La modularidad de la arquitectura permite en un futuro añadir nuevos comportamientos inteligentes al robot.

Así pues, se puede elaborar el siguiente esquema, que muestra de forma resumida y clara los objetivos generales y específicos que dan vida a este proyecto:

- **Objetivo general:** Diseñar e implementar una arquitectura híbrida para un robot Pioneer 3-DX.

  - Establecer las bases de la arquitectura y crear un marco para el desarrollo de comportamientos inteligentes.
  - Dotar de las siguientes características a la arquitectura diseñada:
    - Sencillez.
    - Extensibilidad.
    - Modularidad.
    - Robustez.
- **Objetivo general:** Conseguir que la arquitectura sea lo más independiente posible de la plataforma cliente que haga uso de ella.
  - Funcionar tanto en simulador como en entorno real.
  - Documentar las diferencias encontradas entre las diferentes plataformas.
- **Objetivo general:** Dotar al robot de comportamientos inteligentes.
  - Implementar comportamientos de navegación básica que permitan al robot moverse por un entorno.
  - Dotar de extensibilidad a los comportamientos para incorporar nuevos comportamientos para el robot.

1.3. Fases del Desarrollo

El proyecto ha constado de cinco fases. La primera de ellas ha sido de documentación. En esta fase se ha recopilado información relacionada con el marco del proyecto: arquitecturas robóticas, herramientas software y formas de programar comportamientos robóticos. A continuación se ha llevado a cabo una fase de análisis del problema, realizando un estudio del sistema que finalmente se decidió desarrollar y la identificación de requisitos del mismo. Posteriormente, se entró en una de las fases más importantes de este proyecto: el diseño de la arquitectura, tanto a nivel arquitectónico como detallado. La cuarta fase consistió en la implementación de la arquitectura, y finalmente el proyecto finalizó con una fase de pruebas del sistema.

La figura 1 muestra la planificación seguida durante el desarrollo del proyecto, realizada al comienzo del mismo mediante una herramienta Gantt Project.

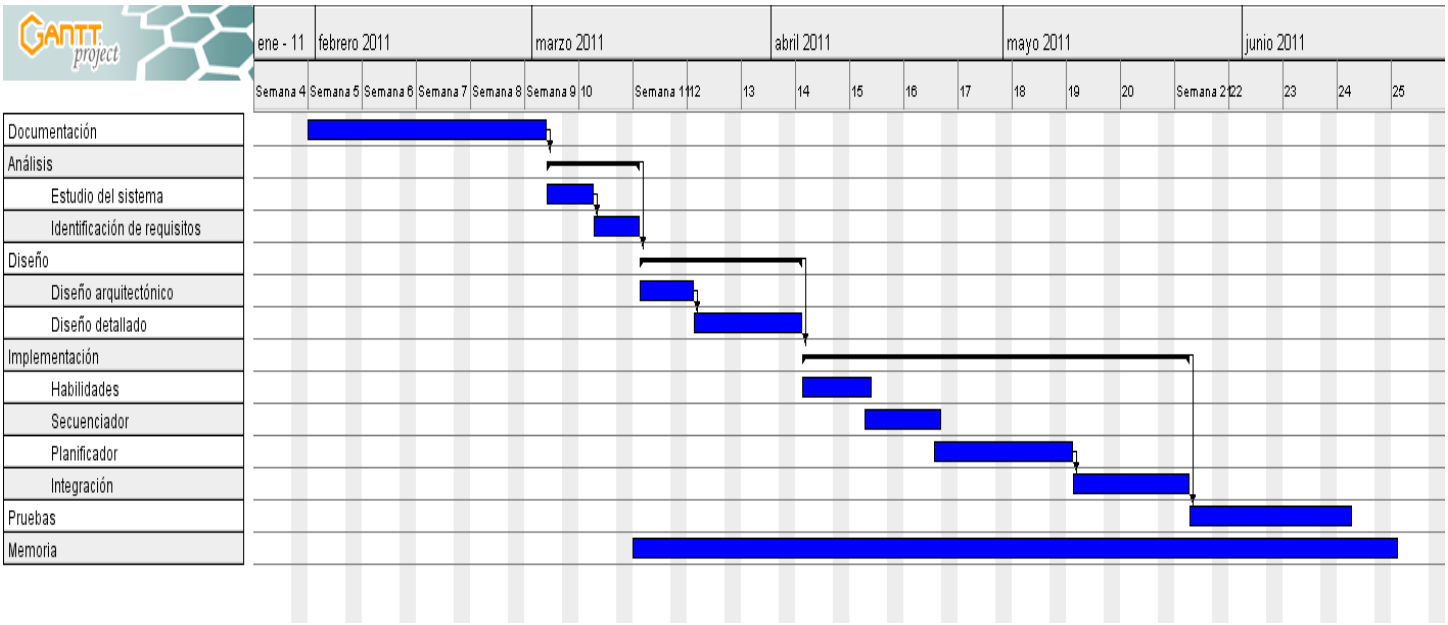


Figura 1: Diagrama de planificación Gantt del proyecto

La planificación Gantt refleja perfectamente que se ha seguido un ciclo de vida en cascada durante el desarrollo del proyecto. Tras la finalización de algunas fases ha sido necesario revisar o mejorar aspectos de fases anteriores, aunque esto no ha sido muy común. Ha existido una pequeña desviación en el tiempo frente a la planificación en la fase de diseño detallado, que se ha alargado una semana más allá de lo previsto. También durante la implementación de la arquitectura ha sido necesario revisar el diseño, pero esto no ha supuesto un mayor coste en cuanto al tiempo planificado.

Por otro lado, la planificación de las tareas de implementación y pruebas ha diferido durante el desarrollo real. Se han incluido pruebas desde los primeros compases de la implementación, debido a que una de las mayores dificultades del proyecto era comprobar su funcionamiento sobre un robot real y realizar un pequeño estudio de los problemas y diferencias encontradas frente a su funcionamiento en simulador. Por ello, se han realizado pruebas de conexión y funcionamiento básico del controlador una vez finalizada su implementación, lo que ha producido una desviación sobre la planificación a la hora de implementar el resto de capas y realizar las pruebas finales. Esta desviación ha sido de 3 semanas.

1.4. Medios Empleados

La mayor parte del desarrollo del proyecto se ha llevado a cabo en el laboratorio de informática 2.2.B06 del campus de Leganés de la Universidad Carlos III de Madrid. En este laboratorio se disponía de los principales medios empleados durante el proyecto.

El 90% del trabajo realizado se ha elaborado con un ordenador portátil Acer proporcionado por el departamento de informática de la Universidad, habiéndose utilizado también un PC de sobremesa para programar ciertas partes de la arquitectura, y sobretodo para la elaboración de la mayor parte de la memoria. Además de este hardware, se ha hecho uso de un robot Pioneer 3-DX, también proporcionado por la Universidad, y cuyas características pueden consultarse en el capítulo 3.1.

En cuanto a las herramientas software utilizadas, que también se detallan en el capítulo mencionado en el párrafo anterior, cabe destacar el simulador Webots para la recreación de mundos virtuales y robots móviles en 3D. Este software es de pago, de forma que el departamento también ha proporcionado una licencia para su uso en el proyecto. También se han utilizado dos licencias de prueba gratuitas de la herramienta Altova UModel para el diseño de la arquitectura. Otro software utilizado ha sido Eclipse (un entorno de desarrollo Java) y diversas herramientas de oficina como editores de texto.

1.5. Estructura de la Memoria

Tras el presente capítulo de introducción y objetivos, este documento se ha dividido en otros seis capítulos que documentan todas las fases del desarrollo del proyecto y aportan la información necesaria para entender el trabajo realizado y facilitar la continuidad de este proyecto en un futuro.

El capítulo 2, Estado del Arte, expone la situación histórica y actual del ámbito en el que se enmarca el proyecto. En este capítulo se realiza un estudio de los diversos paradigmas de la robótica que han aparecido hasta nuestros días, así como de las arquitecturas (reactivas, deliberativas y especialmente híbridas) más importantes que se han desarrollado. El capítulo se completa con la exposición de diferentes plataformas

software utilizadas en el campo de la robótica y la descripción de algunos de los proyectos más conocidos que siguen actualmente en desarrollo.

A continuación, en el capítulo 3, se detalla de forma exhaustiva todo el trabajo realizado durante el proyecto. El capítulo comienza con una descripción de las herramientas utilizadas, tanto hardware como software. Luego se muestran los requisitos identificados para el sistema, y se muestra y explica en detalle el diseño arquitectónico y detallado realizado. A continuación, se expone el funcionamiento de la arquitectura y habilidades implementadas en el robot, para terminar con unas tablas a modo resumen que recogen las librerías de los principales elementos incluidos en el sistema.

El capítulo 4, Experimentación y Resultados, detalla las pruebas del sistema que se han llevado a cabo para comprobar el correcto funcionamiento de la arquitectura implementada, así como para documentar y exponer los resultados obtenidos. También, en este capítulo se explican las principales diferencias observadas entre la ejecución en simulador y entorno real, cuya documentación es uno de los objetivos de este proyecto.

Después, en el capítulo 5, Conclusiones y Líneas Futuras, se exponen una serie de conclusiones que se pueden sacar de todo el desarrollo del proyecto y se recogen los posibles trabajos futuros que pueden llevarse a cabo a partir de los resultados de este proyecto, algo en lo que se ha hecho bastante hincapié durante todas las fases del desarrollo.

Para terminar, el capítulo 6, Referencias, recoge toda la bibliografía utilizada en el proyecto, y en los anexos se incluye el Manual de Usuario, muy útil para aquellos usuarios de la arquitectura que vayan a trabajar en nuevos aspectos sobre el trabajo ya realizado en este proyecto. Entre los anexos incluidos también se encuentra el Presupuesto del proyecto y las tablas del diseño arquitectónico.

1.6. Glosario de Términos y Acrónimos

En este último apartado se recoge un pequeño glosario de los términos y acrónimos utilizados a lo largo del presente documento, con el objetivo de facilitar la lectura y comprensión de las partes más técnicas y/o especializadas.

Términos

- **Actuador**: Dispositivo del robot capaz de recibir órdenes y transformarlas en acciones para modificar el estado del entorno del robot. En este proyecto se hace referencia a un único actuador del robot: el motor.
- **Arquitectura**: Organización fundamental de un sistema (en este caso, de control automático), que incluye sus componentes, las relaciones entre sí y el ambiente, y los principios que gobiernan su diseño y evolución
- **Bumper**: Tipo de sensor parecido a un parachoques, utilizado para detectar contacto con los obstáculos y objetos que forman parte del entorno del robot. Trabaja con una señal todo-nada que únicamente se activa en caso de existir contacto con algún objeto.
- **Comportamiento**: Respuesta del robot frente a una tarea asignada y la situación actual del entorno donde debe llevar a cabo dicha tarea. Se llama comportamiento inteligente a la actuación autónoma del robot consistente en tomar información del entorno, interpretarla y realizar acciones que lleven a la consecución de la tarea.
- **Controlador (aplicación)**: Se llama controlador a la aplicación cliente que se encarga de gestionar el control automático de un robot móvil. Este controlador puede o no hacer uso de una arquitectura de control, y es el encargado de comunicarse con el robot y la arquitectura si la hubiera.
- **Controlador (arquitectura)**: Capa o subsistema inferior de la arquitectura diseñada e implementada en este proyecto, que trabaja a nivel reactivo para llevar a cabo cada una de las tareas asignadas al robot y gestionar la consecución de habilidades.
- **Deliberación**: Forma de actuar de un robot en la que toda acción corresponde a una planificación previa. Se basa en el conocimiento pasado sobre el entorno y la acción futura sobre el mismo.

- **Habilidad**: Comportamiento específico del robot que le permite llevar a cabo una tarea. Es un conjunto de instrucciones y operaciones que permiten al robot mostrar un comportamiento inteligente concreto.
- **Laser**: Tipo de sensor basado en un haz de luz para lo localización de objetos y obstáculos situados en un plano del entorno del robot.
- **Open-source**: Término con el que se conoce al software distribuido y desarrollado libremente. El código abierto tiene un punto de vista más orientado a los beneficios prácticos de compartir el código que a las cuestiones morales y/o filosóficas las cuales destacan en el llamado software libre.
- **Planificador**: Capa o subsistema superior de la arquitectura diseñada e implementada en este proyecto, que trabaja a nivel deliberativo para elaborar planes de acciones que permitan al robot alcanzar una meta global.
- **Reactividad**: Forma de actuar de un robot en la que toda acción corresponde a un estímulo. Se basa en el conocimiento del entorno en el momento presente.
- **Robot**: Sistema electromecánico que, por su apariencia y movimientos, ofrece la sensación de tener un propósito propio. Este término es utilizado para describir tanto a la máquina física como al concepto virtual. Otra definición, según el diccionario de la RAE es la de máquina o ingenio electrónico programable, capaz de manipular objetos y realizar operaciones antes reservadas solo a las personas
- **Robótica**: Ciencia y tecnología de los robots. Se ocupa del diseño, manufactura y aplicaciones de los robots. La robótica combina diversas disciplinas como son: la mecánica, la electrónica, la informática, la inteligencia artificial y la ingeniería de control. También se define como el diseño, fabricación y utilización de máquinas automáticas programables con el fin de realizar tareas repetitivas como el ensamblaje de automóviles, aparatos, etc. y otras actividades.
- **Secuenciador**: Capa o subsistema intermedio de la arquitectura diseñada e implementada en este proyecto, que trabaja tanto a nivel deliberativo como reactivo para crear secuencias de tareas manejables por el robot para conseguir el objetivo del plan.
- **Sensor**: Dispositivo del robot capaz de recibir información del entorno y transformarla en señales eléctricas para su posterior interpretación. En este

proyecto se hace referencia a distintos tipos de sensores, como los sónar o bumpers.

- **Simulador**: Aplicación informática que permite la reproducción de un sistema. Más específicamente, se trata de una plataforma software utilizada para recrear un mundo virtual y las características físicas y programables de un robot móvil, así como la ejecución de estos robots en el entorno virtual.
- **Sonar**: Tipo de sensor basado en ultrasonidos para la localización acústica de objetos y obstáculos en el entorno del robot. Este sensor emite señales acústicas y calcula la distancia a la que se encuentran los objetos midiendo el tiempo que tarda en recibir el eco de la señal.

Acrónimos

- **2D**: Dos dimensiones
- **3D**: Tres dimensiones
- **3T**: Three-Tier (Tres Capas, arquitectura híbrida)
- **ARIA**: Advanced Robotics Interface for Applications (Interfaz avanzada para aplicaciones robóticas, software del robot Pioneer 3-DX)
- **AuRA**: Autonomous Robot Architecture (Arquitectura híbrida)
- **BERRA**: Behaviour-based Robot Research Architecture (Arquitectura híbrida)
- **C++**: Lenguaje de programación orientado a objetos, extensión de C.
- **DSL**: Domain-Specific Language (Lenguaje específico del dominio)
- **ESL**: Execution Support Language (Lenguaje de apoyo a la ejecución de planes en un robot)
- **GPS**: Global Positioning System (Sistema de posicionamiento global)
- **IEEE**: Instituto de Ingenieros Eléctricos y Electrónicos
- **IP**: Internet Protocol (Protocolo de Internet, protocolo de nivel de interred)
- **Java**: Lenguaje de programación orientado a objetos
- **JPL**: Jet Propulsion Laboratory (Laboratorio de Propulsión, Los Angeles – California; Arquitectura deliberativa)
- **LAAS**: Laboratoire d'analyse et d'Architecture des Systèmes (Laboratorio de análisis de arquitectura de sistemas, Toulouse – Francia)

- **LED**: Light-Emitting Diode (Diodo emisor de luz)
- **LISP**: List Processing (Proceso de listas, lenguaje de programación funcional)
- **LPS**: Espacio de percepción local (Elemento de arquitectura híbrida)
- **MS**: Microsoft
- **NASREM**: NASA Standard REference Model (Arquitectura deliberativa)
- **OS**: Sistema operativo
- **PC**: Ordenador personal
- **RAE**: Real Academia de la Lengua Española
- **RAP**: Reactive Action Package (Paquete de acción reactiva)
- **RCS**: Real-Time Control System (Sistema de control en tiempo real)
- **SBM**: Maniobra basada en sensores (Elemento de arquitectura híbrida)
- **SDK**: Software Development Kit (Kit de desarrollo software)
- **SFX**: Sensor Fussion Effects (Arquitectura híbrida)
- **SMPA**: Sense-Model-Plan-Act (Sentir-Modelar-Planificar-Actuar)
- **SSS**: Servo, Subsumption, Symbolic (Arquitectura híbrida)
- **TCA**: Task Control Arquitectura (Arquitectura híbrida)
- **TCP**: Transmission Control Protocol (Protocolo de control de transmisión, protocolo de nivel de transporte de red)
- **URBI**: Plataforma de programación para el desarrollo de aplicaciones robóticas
- **Wi-Fi**: Wide Fidelity (Amplia fidelidad, conexión sin cables)

Capítulo 2

Estado del Arte

Para comprender mejor el trabajo realizado en este proyecto es necesario entender bien el campo en el que se ha trabajado y la situación actual de la robótica. Por ello, en este capítulo se hace una breve introducción a la robótica y a los paradigmas con los que se trabaja. Asimismo, se realizó un estudio de las diferentes arquitecturas que se han aplicado a los robots a lo largo de la historia y se compararon para entender mejor la motivación y objetivos del proyecto.

Como se verá durante el capítulo, se ha hecho especial hincapié en el estudio de las arquitecturas híbridas, ya que se ha trabajado en el robot utilizando este paradigma.

2.1. Introducción a la Robótica

Según el diccionario de la Real Academia de la Lengua Española, un robot es una máquina o ingenio electrónico programable, capaz de manipular objetos y realizar operaciones antes reservadas solo a las personas. La palabra “robot” tiene un origen literario. Otra definición diferente a la de la RAE podría ser la de “máquina de propósito general con capacidades similares (o mejores) a las de los seres humanos”. Incluso hay autores que hablan de “esclavos moralmente aceptables” para referirse a los robots.

Por otro lado, la robótica se define como la ciencia o rama de la ciencia que se ocupa del estudio, desarrollo y aplicaciones de los robots. Otra definición de robótica es el diseño, fabricación y utilización de máquinas automáticas programables con el fin de realizar tareas repetitivas como el ensamblaje de automóviles, aparatos, etc. y otras actividades. Básicamente, la robótica se ocupa de todo lo concerniente a los robots, lo cual incluye el control de motores, mecanismos automáticos neumáticos, sensores, sistemas de cómputos, etc.

Por tanto, se puede decir que un robot inteligente es un sistema de control automático. Pero un sistema de control automático ideal debería ser un sistema universal con capacidad de actuación y percepción. Debería ser autotransportable y amigable, es decir, que interaccione con seres humanos u otros sistemas. Además, el sistema debería ser escalable y distribuido. Pero lo más parecido a esto es un ser humano. Es por ello que actualmente, en el ámbito de la robótica, se busca crear robots completamente autónomos con comportamientos cercanos a los de los seres humanos.

En la figura 2 se pueden observar los elementos básicos de un robot: un sistema de control, un sistema electromecánico, los sensores con los que el robot obtiene la información del mundo (propioceptivos y exteroceptivos) y los actuadores, que le permiten cambiar su propio estado dentro de su modelo del mundo, o cambiar este mismo modelo.

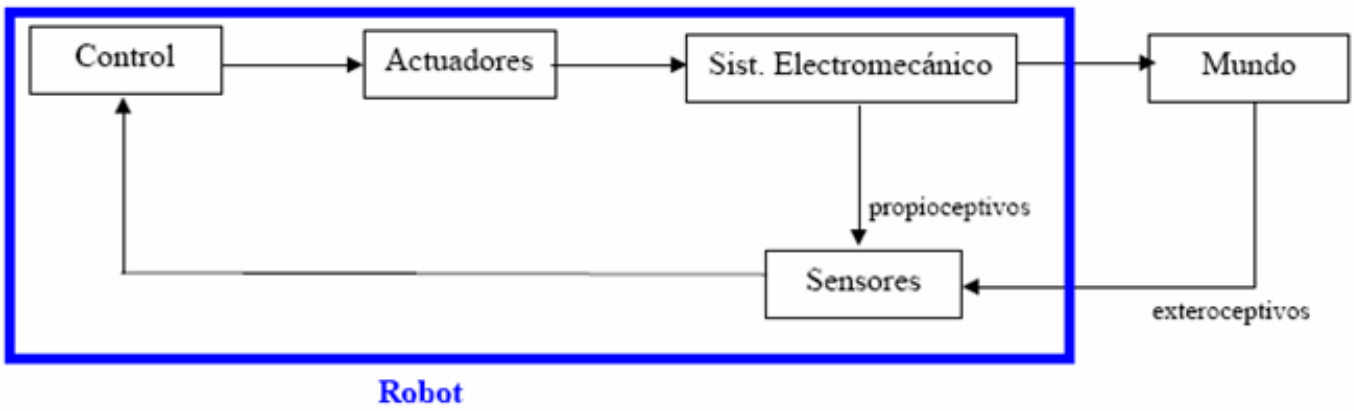


Figura 2: Elementos básicos de un robot

2.2. Paradigmas

Aunque cada sistema suele tener una arquitectura particular, generalmente estas arquitecturas se basan en dos paradigmas: el paradigma reactivo y el paradigma deliberativo. Una combinación de ambos paradigmas, conocida como paradigma híbrido, es la que actualmente tiene más aceptación. A continuación se verán los detalles de estos paradigmas.

2.2.1. Paradigma deliberativo

2.2.1.a. Descripción

Fue popular en los años ochenta. El objetivo de este paradigma es que los robots puedan moverse sin chocarse. Toda acción responde a una planificación previa. Para que el robot sea capaz de decidir, toma información sobre el entorno (sobre un modelo), que generalmente es un mapa. Este modelado del entorno puede no existir si se trabaja con modelos conocidos *a priori*, como por ejemplo los mapas, que son introducidos directamente al robot.

La figura 3 muestra una secuencia de operación con el paradigma deliberativo:

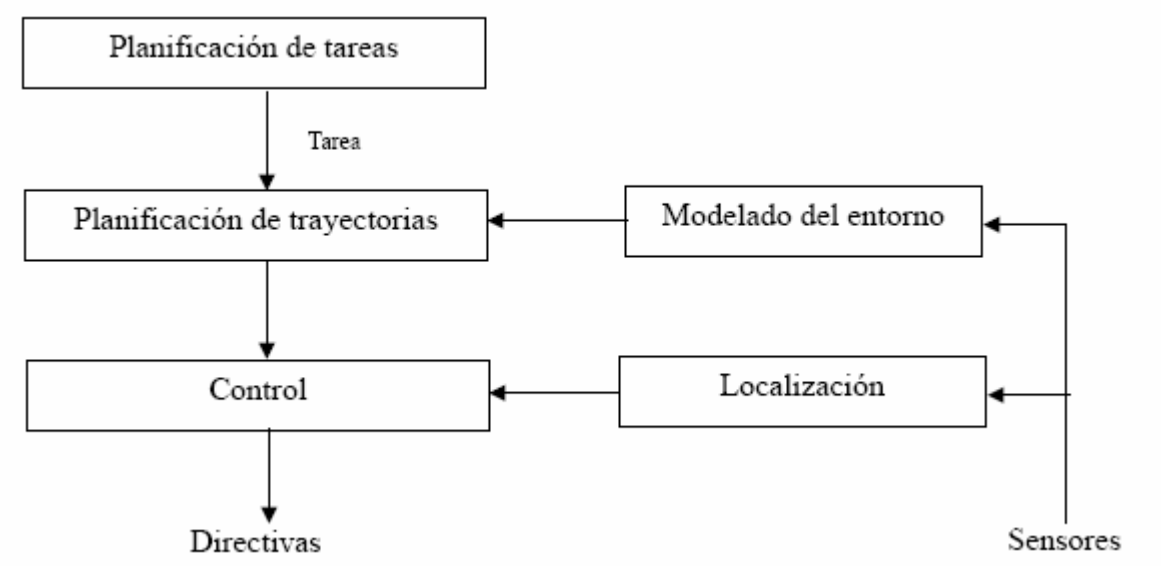


Figura 3: Secuencia de operación del paradigma deliberativo

Es necesario definir una representación interna del entorno del robot para realizar las acciones deliberativas. Además, se necesitan unos algoritmos que traten los datos percibidos y utilicen la representación del entorno para ejecutar las acciones a las que se ha llegado mediante el proceso de decisión. Pero existe un problema: toda esta información es demasiado extensa, y no puede llevarse a cabo de forma eficiente en los sistemas inteligentes. Por otro lado, el entorno del robot es dinámico, lo que puede reflejarse en cambios en el entorno desde que se perciben los datos y se llevan a cabo las acciones, de forma que a la hora de ejecutar las acciones, la información recibida ya no sea válida.

A este problema de la mala adaptación a entornos dinámicos, se le unen otros problemas propios del paradigma. Por un lado, tanto el modelo como la localización del robot nunca son precisos al 100%. Además, cabe destacar la lentitud de los algoritmos deliberativos y la posibilidad de que sea necesario replanificar con mucha frecuencia.

2.2.1.b. Arquitecturas

Antes de pasar a enumerar algunas arquitecturas deliberativas usadas para llevar a cabo el control automático de un robot, es interesante definir qué es una arquitectura en este ámbito.

La arquitectura de un sistema de control se define como la organización fundamental de un sistema, que incluye sus componentes, las relaciones entre sí y el ambiente, y los principios que gobiernan su diseño y evolución [IEEE, 2000]. Una arquitectura se puede observar desde otros dos puntos de vista, más prácticos. Por un lado, desde un punto de vista estático, una arquitectura es un concepto abstracto utilizado para realizar la descripción de un sistema. Desde un punto de vista dinámico, este concepto describe su funcionamiento.

Las arquitecturas que se muestran a lo largo del capítulo, están recogidas en un análisis de arquitecturas de control distribuido [Poza and Posadas, 2009]. Dentro de las arquitecturas deliberativas, existen dos modelos: el secuencial y el paralelo. Un ejemplo de arquitectura deliberativa que sigue un modelo secuencial es JPL (Jet Propulsion Laboratory) [Gat, 1990]. Esta arquitectura se implementó sobre un vehículo de exploración planetaria buscando como objetivo que dicho vehículo poseyera cierto nivel de autonomía parcial. La arquitectura se compone de cuatro módulos principales (figura 4): el módulo de percepción, el planificador de caminos, el planificador y monitor de ejecución y el sistema ejecutor. El fundamento de esta arquitectura es un paradigma secuencial llamado SMPA (Sense-Model-Plan-Act), mediante el cual se ejecutan maniobras predefinidas cuando el robot detecta algún tipo de situación peligrosa o no deseada.

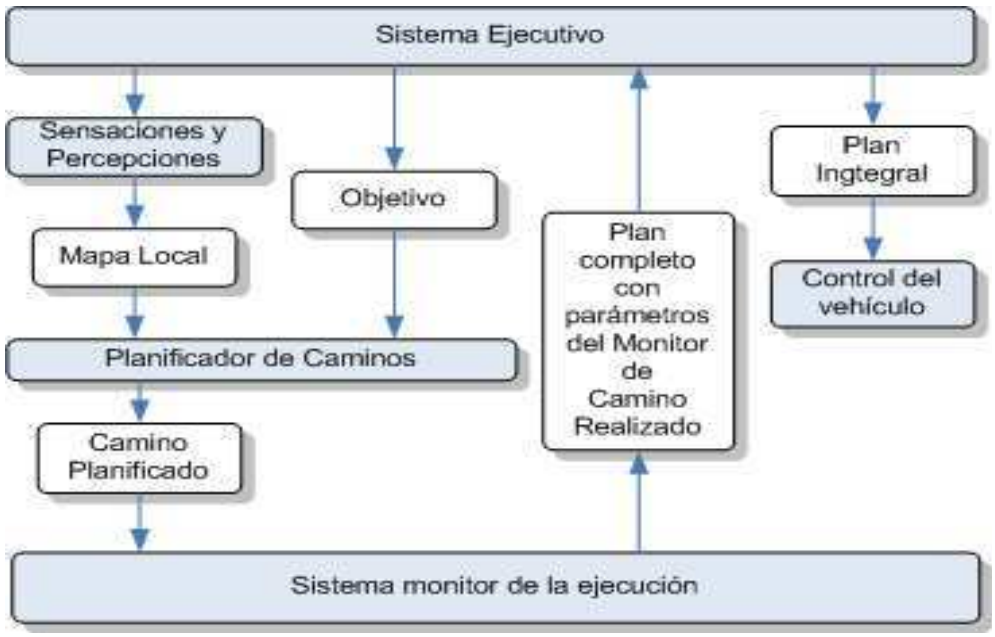


Figura 4: Arquitectura JPL Fuente: [Poza and Posadas, 2009]

Entre las arquitecturas deliberativas que siguen un modelo paralelo, un ejemplo es la arquitectura NASREM (NASA Standard REference Model), que se muestra en la figura 5. El modelo surge como una arquitectura de un sistema RCS (Real-Time Control System) [Albus, 1991]. La arquitectura está formada por diversos elementos inteligentes básicos (actuadores, sensores, procesamiento sensorial, modelo del mundo, generador de comportamientos y estimador de valores) que son integrados en módulos de computación a la hora de ser implementada en un sistema real. Este sistema mantiene de forma interna un modelo del mundo real, es decir, la representación de todo aquello que rodea al robot. Esto lo hace mediante la información procesada y proporcionada por los sensores. Por otro lado, al llevar a cabo acciones de control utilizando los actuadores del robot, se generan comportamientos que permiten lograr los objetivos definidos sobre el modelo del mundo percibido por los sensores. Los elementos nombrados anteriormente son integrados en un modelo jerárquico, tal y como se observa en la figura 5. Diversos autores han evolucionado la arquitectura de diferentes formas. Una interesante evolución es la que aborda jerárquicamente la generación de mapas cuando el robot navega por el entorno, en función de las necesidades de éste [Albus and Barbera, 2005].

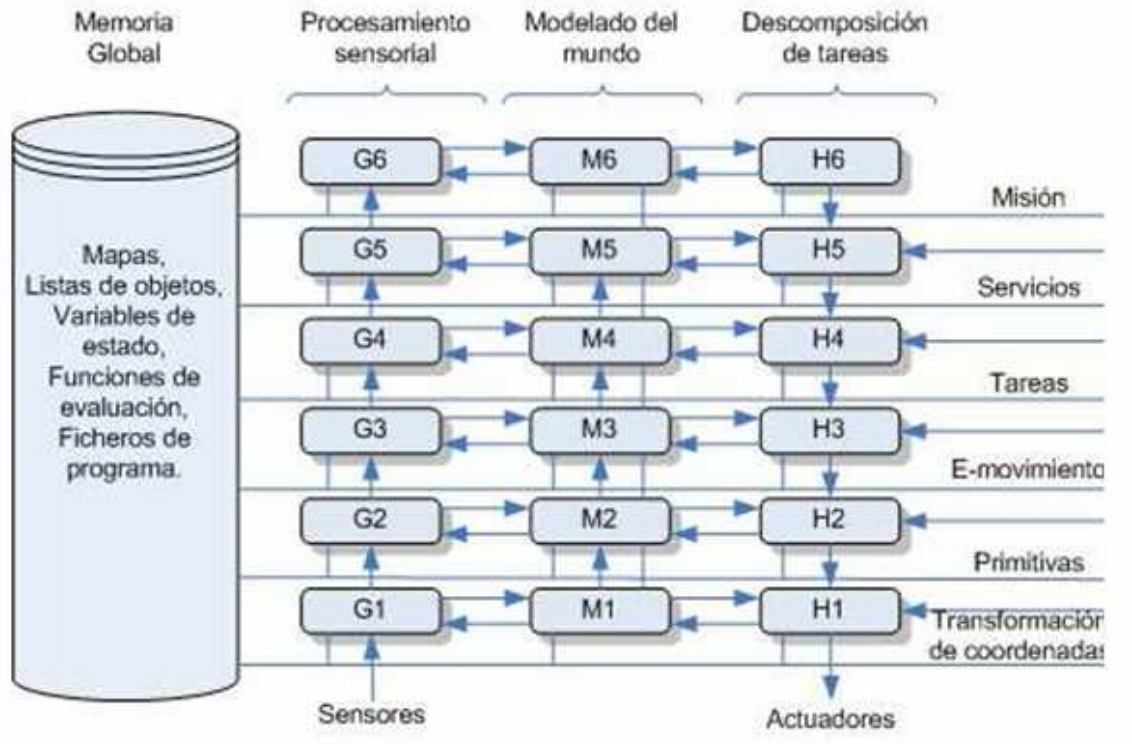


Figura 5: *Arquitectura NASREM* *Fuente: [Poza and Posadas, 2009]*

2.2.2. Paradigma reactivo

2.2.2.a. Descripción

Mientras que la planificación es una actividad previa a la ejecución, es decir, está ligada al futuro; la reactividad es una actividad ligada a la ejecución y que necesita estímulos externos (presente).

En este paradigma, el mejor modelo del entorno es el propio entorno. Se pretende emular habilidades básicas presentes en los animales, incluyendo aquellos poco evolucionados. Son, por tanto, sistemas basados en comportamientos, es decir, no tienen metas o planes, sino que emulan comportamientos (tales como buscar sitios oscuros si hay luz, seguir paredes, huir, etc). Las acciones son provocadas directamente por estímulos sensoriales (ver figura 6).

El objetivo que se busca en la navegación reactiva, es el de no depender de una capa deliberativa para lograr un comportamiento que lo parezca, por lo que el fundamento básico del paradigma reactivo implica una conexión más o menos directa entre los sensores y los actuadores. Esta conexión directa se basa en la dualidad Sentir – Actuar.

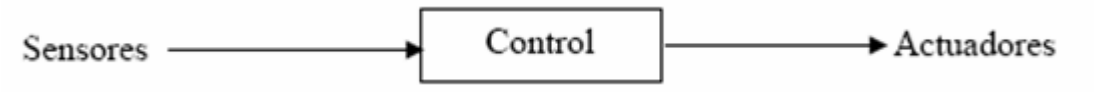


Figura 6: Secuencia de operación básica del paradigma reactivo

Un patrón de comportamiento recoge la salida de un sensor y genera unas señales en los actuadores. Las acciones que provoca un comportamiento son independientes de las que puedan generar el resto de comportamientos que se puedan estar dando en el mismo sistema. La ejecución concurrente de varios patrones de comportamiento hace que algunos de ellos sean eclipsados por otros. El resultado es un comportamiento, que no es básico, y que se corresponde a una combinación de comportamientos denominado comportamiento emergente.

Los comportamientos definen diferentes tipos de objetivos. En las arquitecturas planificadas el objetivo era alcanzar una meta. En las arquitecturas reactivas el objetivo es comportarse de una forma determinada, es una meta continua que nunca acaba. Para conseguir este objetivo mediante comportamientos emergentes, se pueden utilizar diversos métodos de fusión. Esta fusión de comportamientos da lugar a habilidades. Además, pueden incluirse pequeñas memorias temporales en las arquitecturas reactivas. Todo esto en conjunto nos da un esquema para el paradigma reactivo parecido al que muestra la figura 7.

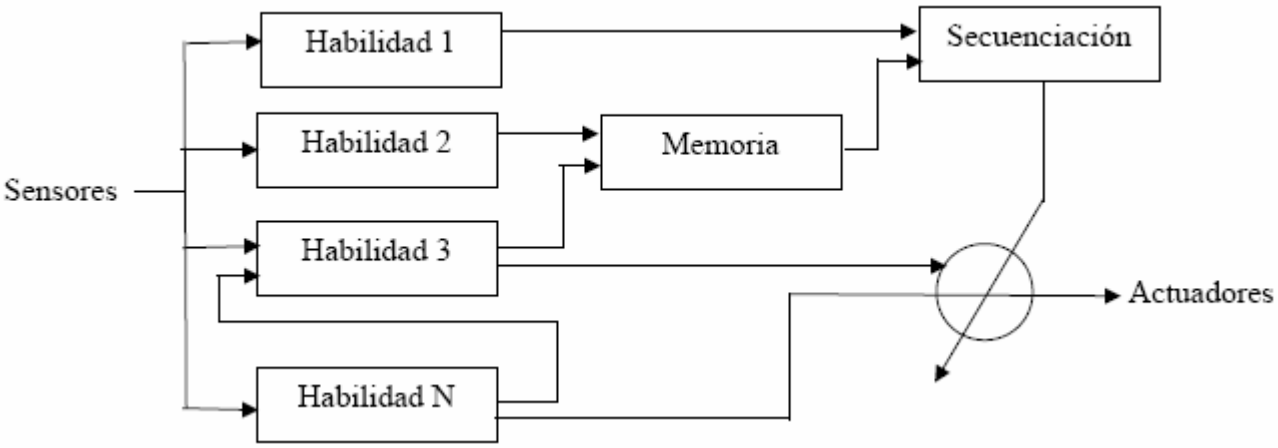


Figura 7: Secuencia de operación compleja del paradigma reactivo

Al contrario que en el paradigma deliberativo, no existe una representación interna del entorno del robot, de forma que los sistemas reactivos no necesitan llevar a cabo una localización o planificación. Así, estos sistemas no necesitan procesamientos complejos o estructuras de datos que representen el modelo del mundo. Tampoco dependen de algoritmos, por lo que estos sistemas también se caracterizan por poseer cierto nivel de predicción temporal, lo que permite cumplir restricciones de tiempo. Pero esto no es siempre así. Hay sistemas que se encuentran en la frontera entre este paradigma y el deliberativo, debido a la inclusión de algún tipo de memoria sencilla en el sistema. Con este nuevo elemento, el entorno sí está siendo representado, aunque de forma muy primitiva.

Para concluir con el paradigma reactivo, es importante mencionar los problemas más comunes en este tipo de arquitecturas. Algunos de ellos son derivados de trabajar con información local, como que es difícil alcanzar objetivos globales y no suele

optimizar. También es común que el robot caiga en errores locales provocados por bucles infinitos.

2.2.2.b. Arquitecturas

Entre las arquitecturas que utilizan un paradigma reactivo, las más conocidas son la arquitectura de subsunción (o subsumption) de Brooks y los vehículos de Braintenberg, que no son exactamente una arquitectura, sino que utilizan un modelo de campos potenciales para actuar de forma reactiva.

En los vehículos de Braintenberg, existe conexión directa (o basada en combinaciones sencillas) entre los sensores y los actuadores del robot [Braitenberg, 1984]. En la figura 8 se puede observar un sencillo ejemplo de esto. Los vehículos de Braintenberg se utilizan como representación básica de los principios de la navegación de robots utilizando el paradigma reactivo. Desde un punto de vista arquitectónico, estos vehículos suponen una cota inferior; representan la información y las comunicaciones básicas necesarias para implementar una arquitectura reactiva. Pero se pueden llevar a cabo misiones u objetivos más complejos con el simple hecho de incorporar más procesamiento entre los sensores y los actuadores. Una de las características de estos vehículos es que muy difícil (prácticamente imposible) predecir cómo van a comportarse, pese a que las conexiones entre sensores y actuadores son muy básicas y deterministas. Esto no quiere decir que no se puedan introducir comportamientos a los vehículos, ya que es posible conseguirlo variando las conexiones.

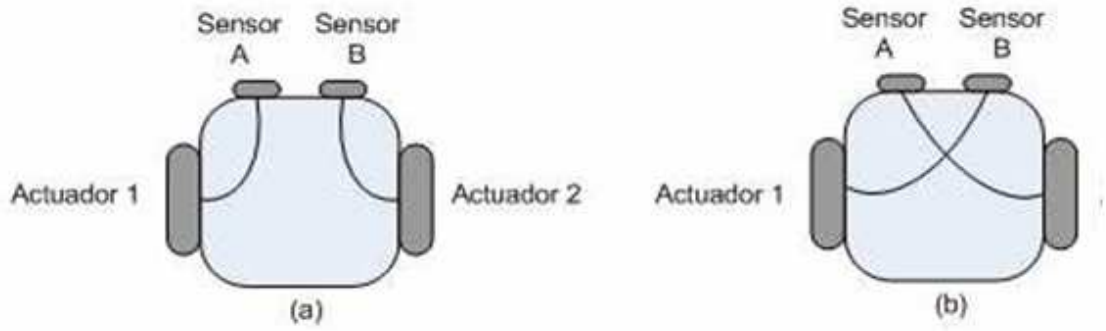


Figura 8: Vehículos de Braintenberg Fuente: [Poza and Posadas, 2009]

Sobre la arquitectura de subsunción, en primer lugar, se puede definir "subsumir" como incluir algo como un componente en una síntesis o clasificación más

abarcadora. También se puede considerar como algo parte de un conjunto más amplio, o como caso particular sometido a un principio o norma general. [Poza and Posadas, 2009]. La arquitectura de subsunción es no simbólica y no verbal, y es usada como alternativa a las arquitecturas más tradicionales (ver figura 9.a). Pero también puede hacerse uso de sus módulos básicos como ampliación para otro tipo de arquitecturas más complejas. La arquitectura de subsunción [Brooks, 1986] es una aproximación de procesamiento paralelo (ver figura 9.b), que hasta entonces no se había utilizado en el campo de la inteligencia artificial clásica, frente a aproximaciones secuenciales. Los sensores del robot son las entradas de información de la arquitectura, para todas y cada una de sus capas. Estas capas procesan la información obtenida por los sensores para ofrecer posteriormente los resultados a los actuadores, que tendrán un efecto sobre el entorno del robot. Una gran ventaja de esta arquitectura es que es posible realizar una implementación de cada capa de diferentes maneras. Algunos autores utilizan máquinas de estado finito, mientras que otros prefieren redes neuronales, por dar algunos ejemplos.

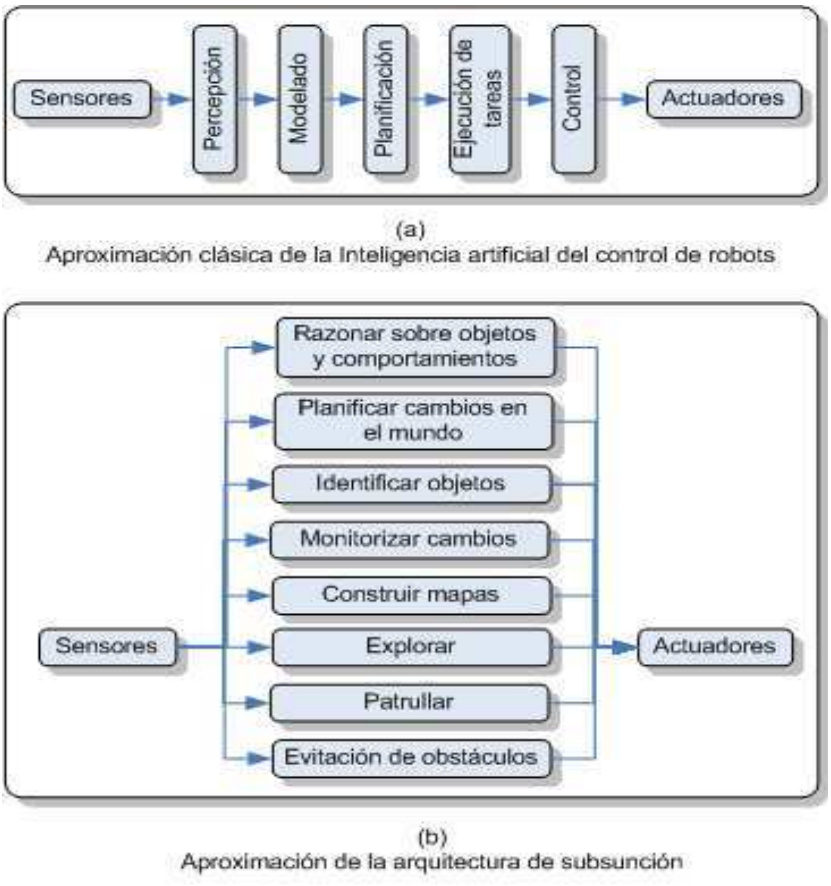


Figura 9: Arquitectura Subsumption Fuente: [Poza and Posadas, 2009]

2.2.3. Paradigma híbrido

El paradigma híbrido aúna las características de ambos paradigmas (deliberativo y reactivo). Su esquema básico, mostrado en la figura 10, es poseer una parte deliberativa, con un planificador, pero actuar a bajo nivel de forma reactiva, con un sistema ejecutor basado en comportamientos.

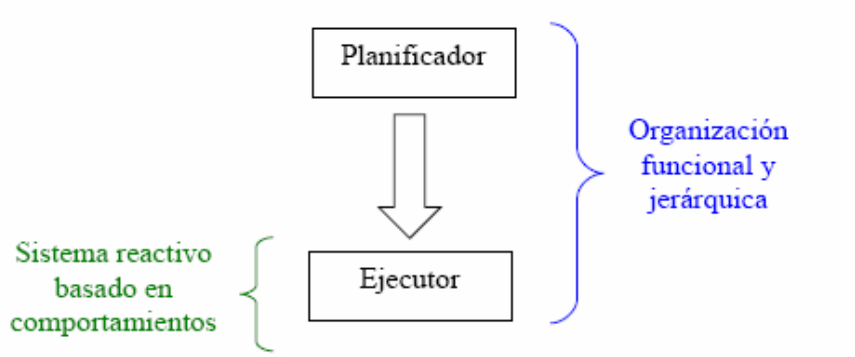


Figura 10: Esquema básico del paradigma híbrido

Con este paradigma se consigue la velocidad de actuación que proporciona el paradigma reactivo, así como la garantía de cumplir las restricciones de tiempo, pero a su vez también se consigue la potencia de navegación propia del paradigma deliberativo. Existen diferentes factores que determinan la separación entre los dos niveles, pero hay dos fundamentales: cómo se utiliza la información de entrada al sistema (si proviene directamente de los sensores, si es tratada previamente por algún módulo del sistema, etc.) y qué información se envía a los actuadores o es utilizada por otros niveles de las arquitecturas.

Así pues, se puede considerar que el paradigma híbrido se sustenta en dos capas diferenciadas: una capa reactiva que aporta al robot los mecanismos básicos de supervivencia en el entorno, y una capa deliberativa que le permite llevar a cabo tareas más complejas. La capa reactiva obtiene información del entorno de los sensores y manda esta percepción a la capa deliberativa, que a su vez puede interaccionar con la capa reactiva con respecto a las acciones que debe ejecutar el robot.

Pero esta solución es muy rígida. Para flexibilizar la arquitectura híbrida hay que redefinir el concepto de “plan”, que es sustituido por unas pautas más flexibles. Se toma como ejemplo un plan que indica las acciones a llevar a cabo por un robot humanoide

para ir de un despacho a otro. Este plan podría ser algo así: "*levántate, sigue ciertas coordenadas hacia la puerta, abre la puerta, gira, sigue ciertas coordenadas hacia el otro despacho…*" La idea es convertir ese "plan" en unas instrucciones más flexibles como "*sal del despacho, gira y sigue el pasillo hasta el otro despacho*".

Este nuevo "plan" es una secuencia de comportamientos que debe seguir la parte reactiva del sistema. Esta secuencia puede ser lineal o tener múltiples alternativas en función de que se den o no ciertas condiciones. De esta forma, un sistema basado en el paradigma híbrido podría tener la siguiente arquitectura:

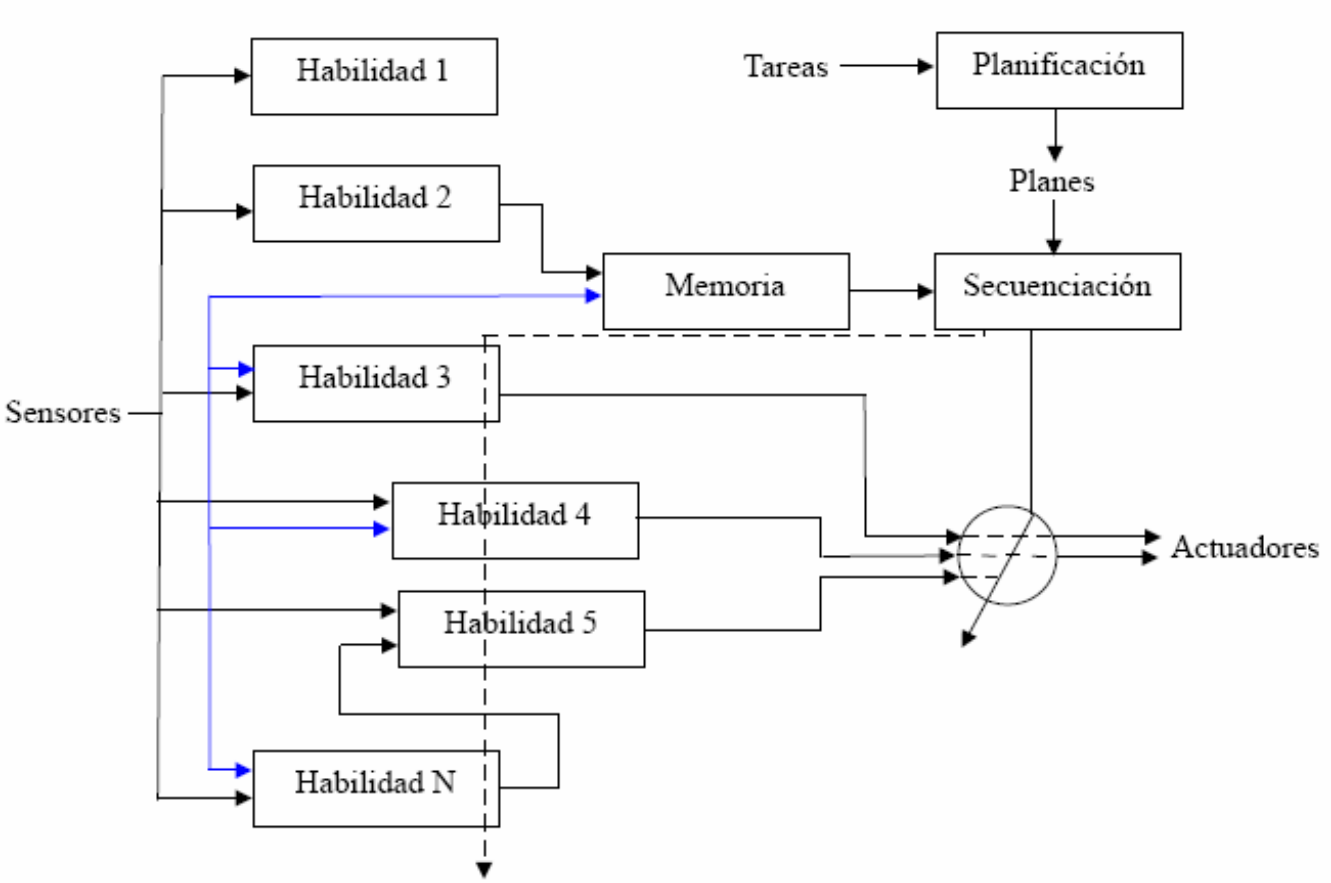


Figura 11: Secuencia de operación compleja del paradigma híbrido (a)

En la figura 11 se pueden apreciar algunas de las formas en las que las habilidades se comunican o son tratadas por los diversos componentes presentes en una arquitectura híbrida. Cabe destacar que es sólo un ejemplo, y que ni todos los elementos son obligatorios, ni son los únicos que pueden aparecer. El planificador puede ser sustituido por una librería de planes, o incluso puede ser el mismo secuenciador quien

controle la planificación del sistema, tal y como se muestra en la figura 12.

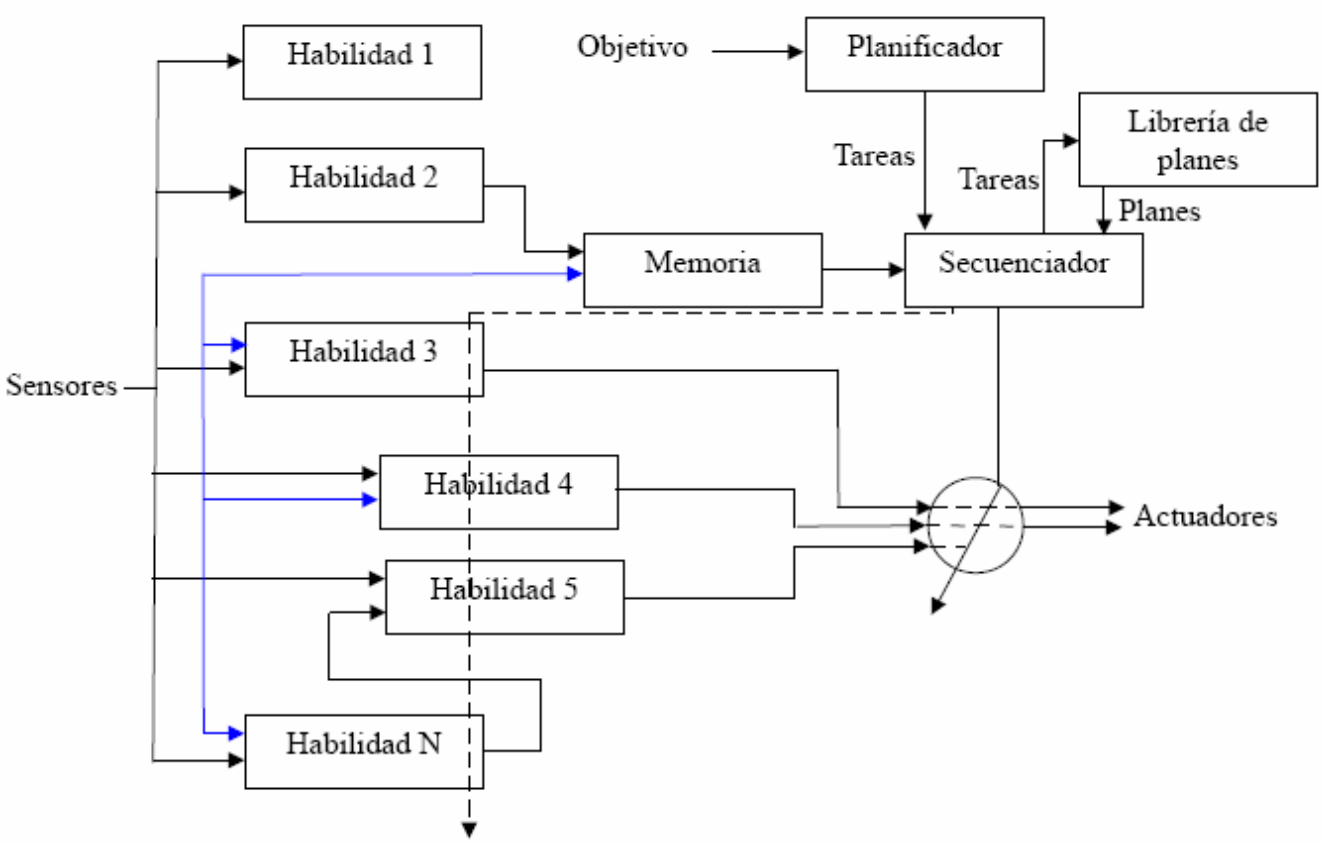


Figura 12: Secuencia de operación compleja del paradigma híbrido (b)

2.3. Arquitecturas Híbridas

Por ser el tema central de este proyecto las arquitecturas híbridas, este apartado se centra en el estudio de este tipo de arquitecturas. Existen multitud de arquitecturas híbridas, y por sus características, éstas se suelen dividir en organizativas, basadas en jerarquías de estados, orientadas a modelos y organizadas en niveles.

En este apartado se exponen brevemente algunos ejemplos de cada uno de estos tipos de arquitecturas híbridas, destacando sus características más importantes, para luego realizar una comparativa de todas ellas. Este estudio es importante porque en base a él posteriormente se eligió una de las arquitecturas para basarse en ella a la hora de crear la arquitectura híbrida para el robot, objetivo principal del proyecto. Se tendrán en cuenta aquellas características de las arquitecturas que permitan conseguir otros objetivos secundarios del proyecto, como la modularidad y extensibilidad de ésta para

futuros trabajos que tengan como base la arquitectura implementada.

2.3.1. Organizativas

Estas arquitecturas toman conceptos empresariales para realizar una división de responsabilidades. Se exponen las arquitecturas AuRA [Arkin, 1990], la SFX [Murphy, 2000] o HILARE [Alami et al., 2000]. Este tipo de arquitecturas, llamadas también gerenciales, poseen una capa deliberativa que maneja el conocimiento global del mundo y organizan las responsabilidades utilizando una jerarquía de éstas, y una capa reactiva que maneja los comportamientos básicos del robot, introduciendo también memoria en el sistema y un estado externo. Las arquitecturas gerenciales suelen estar presentes en el ámbito de la información espacial (frente a la información temporal).

2.3.1.a. AuRA

AuRA (Autonomous Robot Architecture) es una arquitectura utilizada en robots móviles que se fundamenta en la teoría de esquemas. Posee una gran inspiración en sistemas biológicos, basando sus comportamientos en enfoques psicológicos y neurofisiológicos. Como todas las arquitecturas deliberativas, distingue claramente dos capas de actuación: deliberativa y reactiva (ver figura 13).

El control reactivo se basa en comportamientos primitivos o esquemas, y mediante la composición de éstos surgen otros comportamientos más complejos. Esta complejidad depende en gran medida de la complejidad con la que los distintos esquemas son combinados. Entre ambos niveles de la arquitectura (reactivo y deliberativo) existe un componente que caracteriza a AuRA; el subsistema homeostático. Este componente controla tareas de supervivencia para el robot cuando éste se encuentra ante circunstancias peligrosas, pero también se encarga de mantener el equilibrio entre ambas capas de la arquitectura. De esta forma, permite al robot la alteración dinámica de su comportamiento en el nivel inferior. Por último, la capa deliberativa está formada por un sistema cartográfico y otro de planificación.

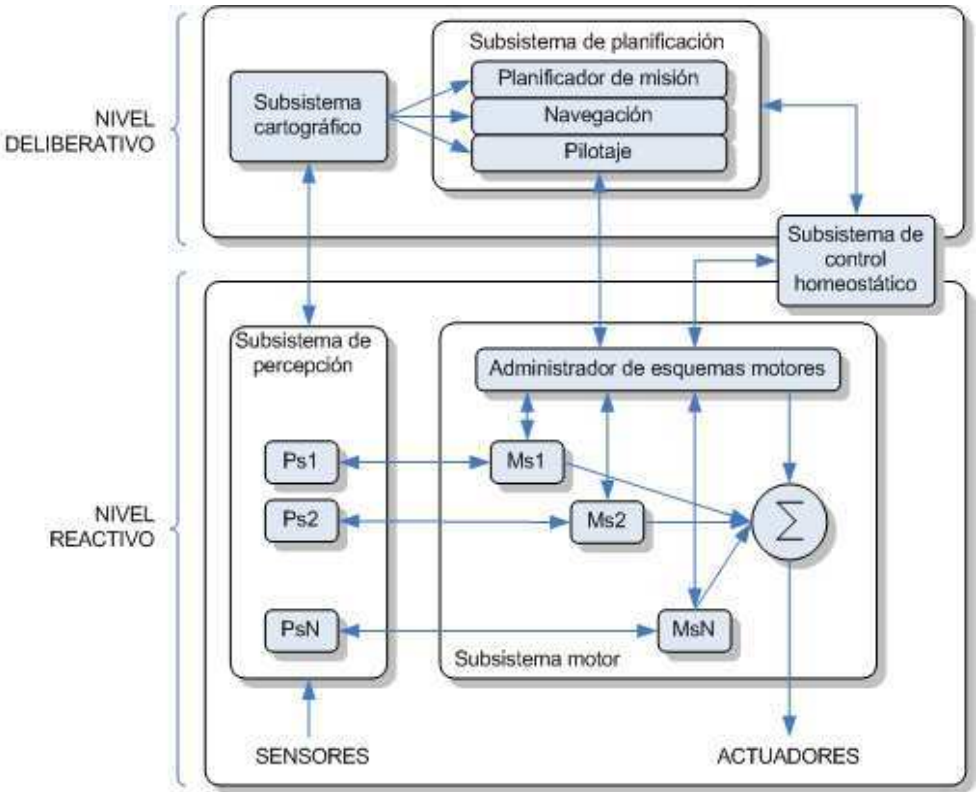


Figura 13: Arquitectura AuRA *Fuente: [Poza and Posadas, 2009]*

Como se puede ver, AuRA posee un diseño basado en módulos muy diferenciados. Esto permite que se puedan reemplazar por otros módulos de una forma fácil y segura, lo que otorga una gran flexibilidad de implementación de la arquitectura.

2.3.1.b. SFX

La arquitectura SFX (Sensor Fusion Effects) surge como una extensión de AuRA en la que se incluye una nueva capa de robustez. Esta capa incorpora procesos de fusión sensorial y es tolerante a fallos. La información percibida por los sensores se proporciona a ambas capas (reactiva y deliberativa) por medio de pizarras, una estructura de datos muy común en sistemas de inteligencia artificial, que permite formar estructuras de conocimiento global. La figura 14 muestra gráficamente la estructura de SFX.

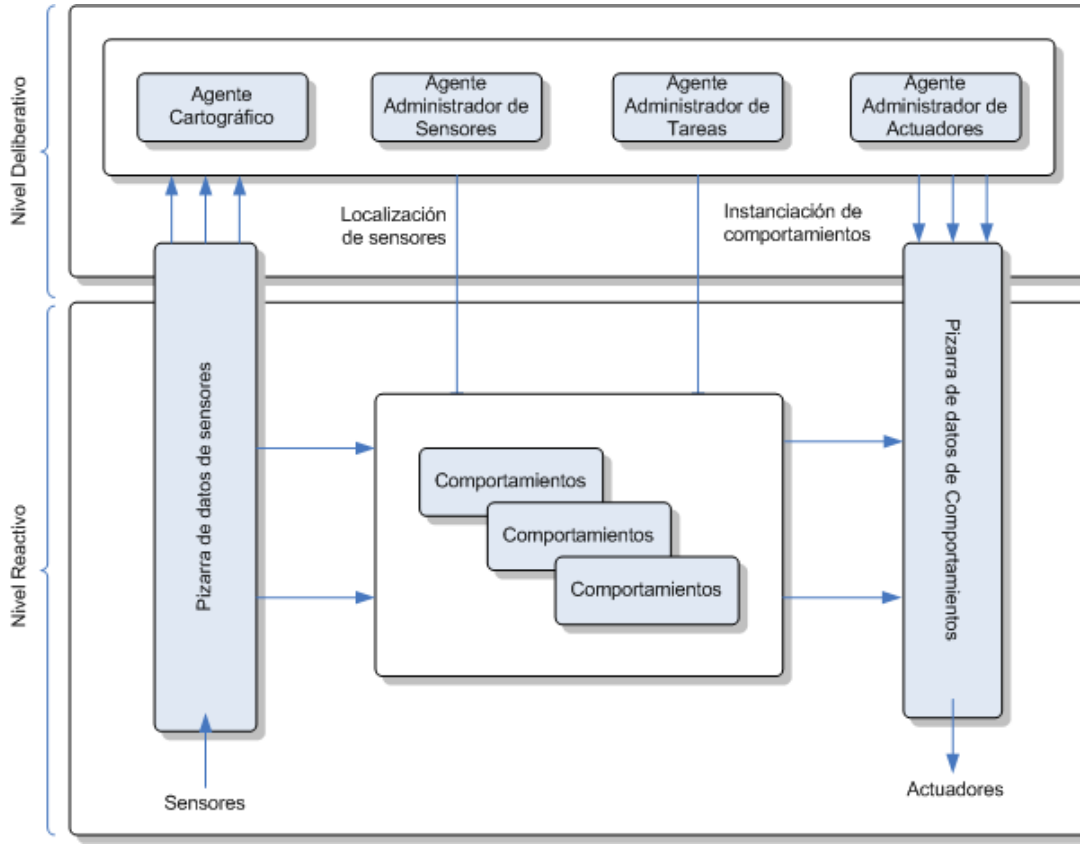


Figura 14: Arquitectura SFX Fuente: [Poza and Posadas, 2009]

El nivel deliberativo de la arquitectura SFX está basado en agentes, que son componentes software independientes especializados en funciones concretas, con capacidad para interactuar entre ellos. El agente principal de este nivel es el planificador de misiones, que se encarga de la interacción con el usuario y la especificación de la misión al resto de agentes. El objetivo de estos otros agentes es dar con los comportamientos que aseguren el éxito de la misión, cumpliendo con todas aquellas restricciones que se hayan definido para ella. Por otro lado, el nivel reactivo está formado por dos subcapas, correspondientes a los comportamientos estratégicos y a los tácticos. Mientras que AuRA hace uso de esquemas (campos potenciales) para lograr la fusión de comportamientos, SFX realiza métodos de filtrado que hace que los comportamientos fundamentales inhiban a otros comportamientos, tal y como ocurría en la arquitectura de subsunción de Brooks, vista como exponente del paradigma reactivo. Sin embargo, en el caso de la arquitectura SFX son los comportamientos tácticos (capa inferior) los que inhiben a los comportamientos estratégicos (capa superior).

2.3.1.c. HILARE

La arquitectura HILARE, desarrollada en el LAAS (*Laboratoire d'analyse et d'Architecture des Systèmes*, Toulouse), está compuesta de tres niveles jerárquizados, cada uno de los cuales posee diferentes restricciones temporales y representa los datos de forma distinta (ver figura 15).

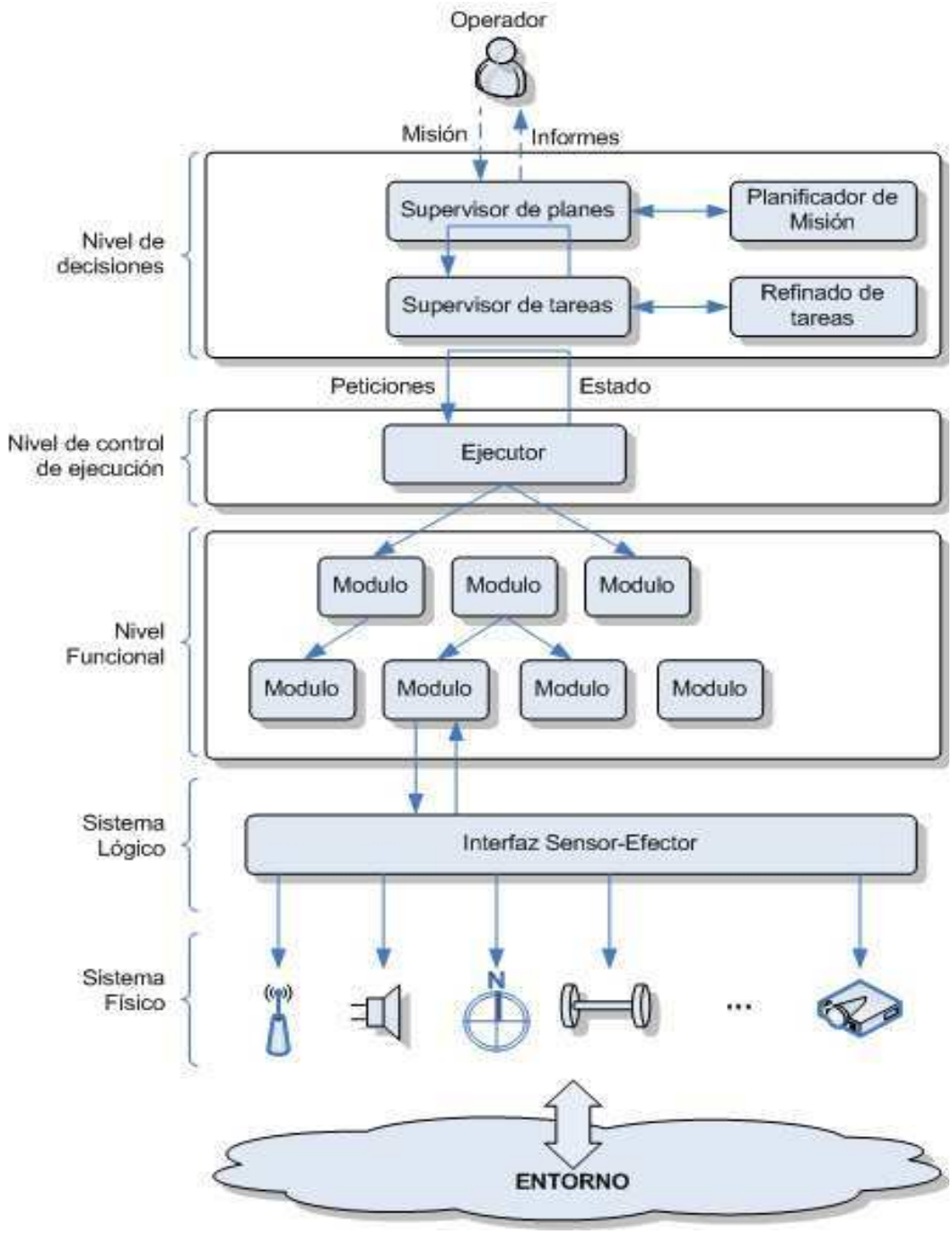


Figura 15: Arquitectura HILARE Fuente: [Poza and Posadas, 2009]

El nivel funcional se corresponde con el nivel reactivo del paradigma híbrido; este nivel controla la percepción de robot y su capacidad de acción, ambas encapsuladas en módulos controlables y comunicados entre sí. A continuación está el nivel ejecutivo, encargado de controlar y coordinar de manera dinámica la ejecución de las diferentes funciones que se encuentran distribuidas entre los módulos, de acuerdo con las tareas que son especificadas en el siguiente y último nivel: el de decisión. El nivel de decisión incorpora aquellas capacidades que permiten producir las tareas del plan y su supervisión. Este nivel de decisión, a su vez, puede estar formado por varias capas, según lo requieran las capacidades, ya que este nivel es un planificador y supervisor con la característica de que permite integrar deliberación y reactividad.

2.3.2. Basadas en jerarquías de estados

Las arquitecturas basadas en jerarquías de estado están organizadas en el ámbito temporal, al contrario que las anteriores. Como ejemplos de este tipo de arquitecturas se presentan la 3T [Bonasso, 1997] y ATLANTIS [Gat, 1992]. Este tipo de arquitecturas suelen estar formadas por tres capas que diferencian el instante temporal en el que trabajan: pasado, presente y futuro. Y es, mediante esta distinción del instante temporal, como se distinguen las capas reactivas y deliberativas. La capa deliberativa suele trabajar con el conocimiento pasado y futuro (en este caso, suposiciones y predicciones futuras). Esta capa está organizada en función de un estado interno temporal. El estado pasado es cosa del secuenciador (la monitorización proporciona información del pasado), mientras que el estado futuro es cosa del planificador (la planificación genera información futura). En estas arquitecturas es posible la planificación en tiempo real si se tiene en cuenta la velocidad de ejecución como modo de organización de las tareas. Por otro lado, los comportamientos emergen gracias a la generación y monitorización de secuencias de comportamientos, la fusión de éstos en habilidades y la subsunción.

2.3.2.a. Arquitectura 3T

Esta arquitectura se diseñó para funcionar en diferentes robots independientemente del hardware que poseyeran. Hace uso de varios niveles de abstracción para que el robot pueda llevar a cabo varias tareas (ver figura 16). Una

característica fundamental de esta arquitectura es que está diseñada para funcionar específicamente en entornos dinámicos.

El nivel superior lo compone el planificador, encargado de seleccionar objetivos que luego se descomponen en el nivel intermedio (secuenciador) en una serie de acciones de bajo nivel. En el nivel inferior, el controlador descompone de nuevo las acciones en habilidades. Esta estructura en tres capas permite observar claramente las características del paradigma híbrido, en el que el planificador hace las veces de capa deliberativa y el controlador de capa reactiva. El secuenciador quedaría entre las dos capas, encargado de activar o desactivar diferentes conjuntos de habilidades para llevar a cabo el plan elaborado.

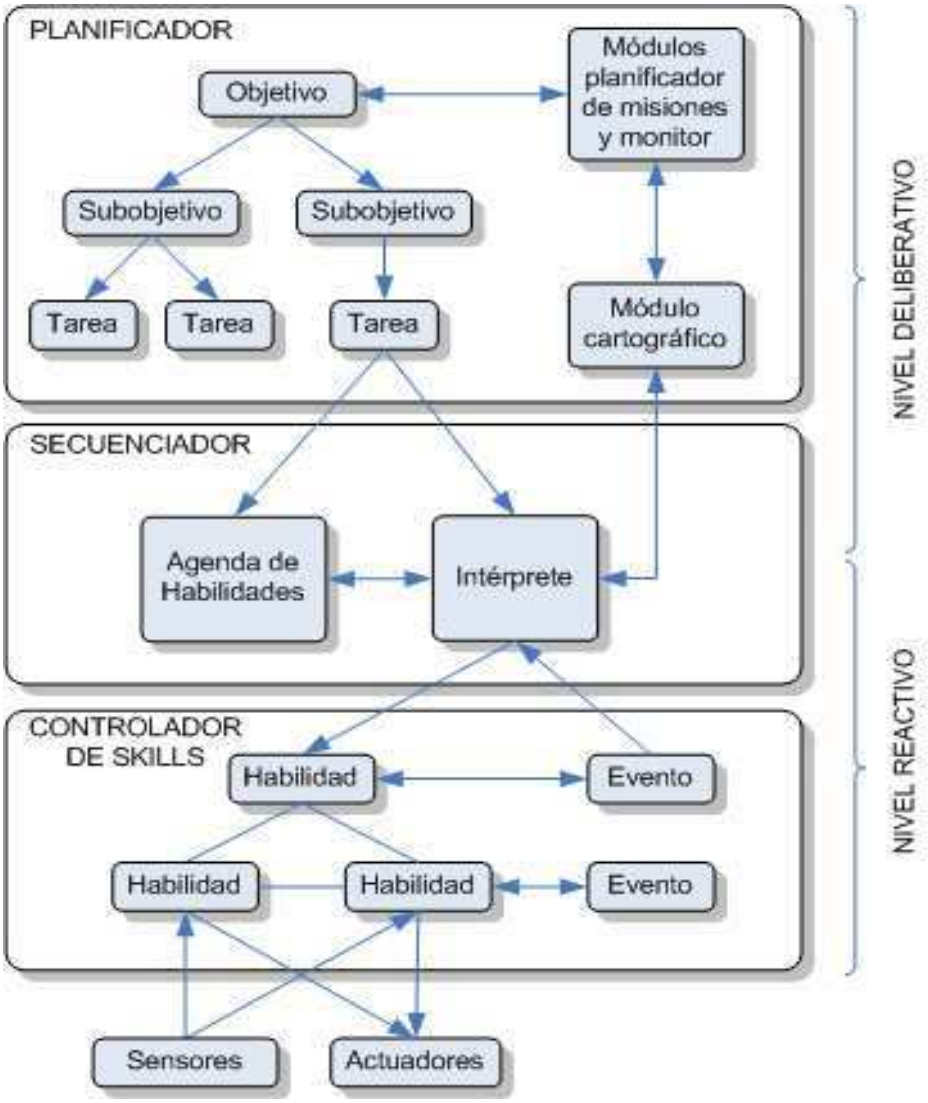


Figura 16: Arquitectura 3T Fuente: [Poza and Posadas, 2009]

2.3.2.b. ATLANTIS

La arquitectura ATLANTIS es muy parecida a la 3T. La característica diferenciadora fundamental de esta arquitectura frente a otras es que no emplea el modelo clásico de estados y acciones. Este modelo se basa en que la evolución de un sistema o entorno a lo largo del tiempo se puede describir como un conjunto de estados discretos. Un estado pasa a otro distintos mediante una acción, que puede ser una estructura computada o bien una acción física del entorno. Por su parte, la arquitectura ATLANTIS se basa en un modelo de acciones continuo, no discreto. Este modelo utiliza operadores con tiempo de ejecución muy pequeño en comparación con el tiempo que necesitan los procesos más costosos que se ejecutan en el sistema. En este modelo los procesos se llaman actividades y los operadores, decisiones.

Los niveles superiores de la arquitectura se componen de procesos computacionales que producen actividades en niveles inferiores, y en el nivel más bajo se encuentran las actividades primitivas. Éstas son procesos completamente reactivos. Es complicado analizar un modelo de acciones basado en actividades como un modelo de clásico de estados y acciones, y eso se debe a la inexistencia de correspondencia entre decisiones y cambios en el entorno. ATLANTIS está formada por tres componentes, al igual que la arquitectura 3T: un controlador, un secuenciador y un componente deliberativo (ver figura 17).

El controlador se encarga de gobernar las actividades primitivas (reactivas), caracterizadas por carecer de acciones computacionales. Por otro lado, el secuenciador controla la secuencia de actividades primitivas y computaciones deliberativas. El hecho de secuenciar es complicado, dado que no es trivial hacer un reparto efectivo de las acciones.

El principio básico del secuenciador es el denominado “fallo consciente”, es decir, el sistema puede darse cuenta de que se ha producido un fallo en un algoritmo. El secuenciador inicia y finaliza las actividades primitivas. Para ello, activa y desactiva conjuntos de módulos del controlador, enviando además parámetros mediante algún canal de comunicación con éste. Por último, otra función del secuenciador es monitorizar el estado y progreso de una actividad primitiva.

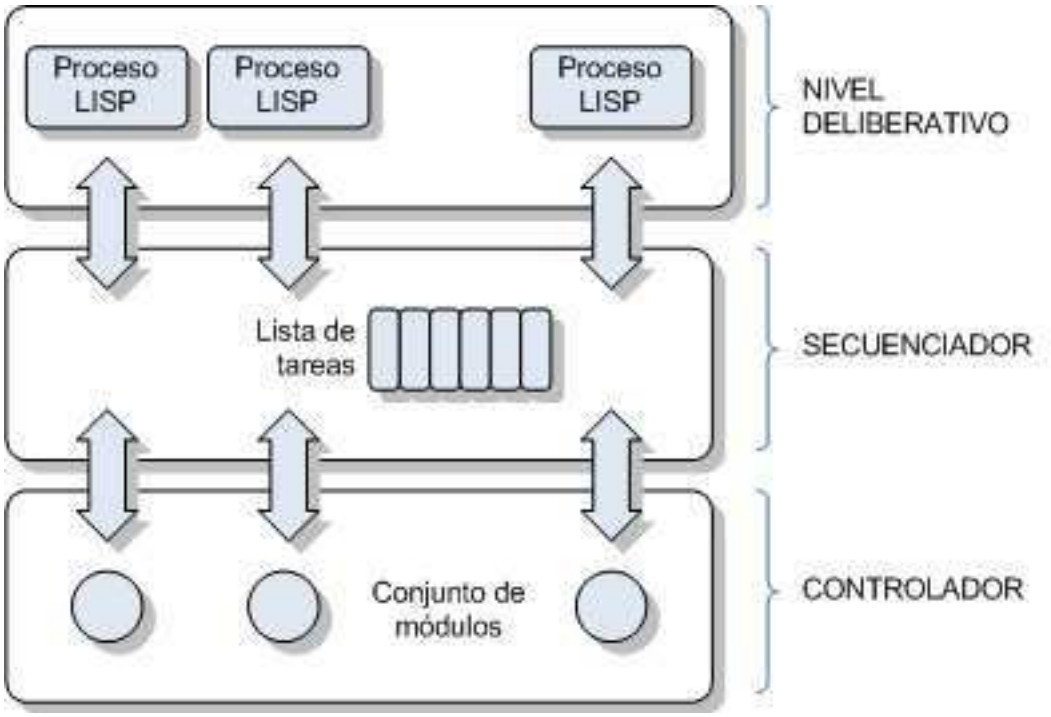


Figura 17: Arquitectura ATLANTIS Fuente: [Poza and Posadas, 2009]

Ambos componentes, secuenciador y controlador, deben proveer un interfaz que permita la conexión entre los sensores y actuadores del robot con un sistema de ejecución de algoritmos. La arquitectura ATLANTIS también proporciona esta conexión entre el modelo clásico de estados y acciones con el modelo propuesto por esta arquitectura de actividades continuas.

2.3.3. Orientadas a modelos

Son arquitecturas orientadas a modelos aquellas que utilizan dichos modelos como sensores virtuales. De este tipo se exponen Saphira [Konolige, 1998] y TCA [Simmons, 1994]. En ellas, el nivel deliberativo lo componen todo aquello relacionado con comportamientos que permiten lograr un objetivo. Por su parte, el nivel reactivo representa los comportamientos como unidades de control que trabajan en el presente, pero que a su vez hacen uso del conocimiento global mediante sensores virtuales. Estas arquitecturas orientadas a modelos tienen su ámbito tanto en el uso de la información temporal como espacial.

2.3.3.a. Saphira

La arquitectura Saphira [Konolige, 1998] se basa en un sistema integrado de sensorización y control. Posee un componente central llamado LPS o espacio de percepción local, que se trata de una representación geométrica del entorno del robot (ver figura 18). Este componente está diseñado para poder adaptarse a diversos niveles de interpretación de los datos sensoriales, ya que diferentes tareas encomendadas al robot hacen uso de distintas representaciones.

El componente LPS se encarga de proporcionar la información sobre el entorno al robot, y es el elemento fundamental en las tareas de fusión de información, planificación e integración de la información del mapa. Los niveles de control y percepción hacen uso constantemente del LPS, que proporciona coherencia en la representación a toda la arquitectura. Mientras, en el nivel de control, el problema de control se basa en el uso de comportamientos básicos como evitar obstáculos o seguir un pasillo.

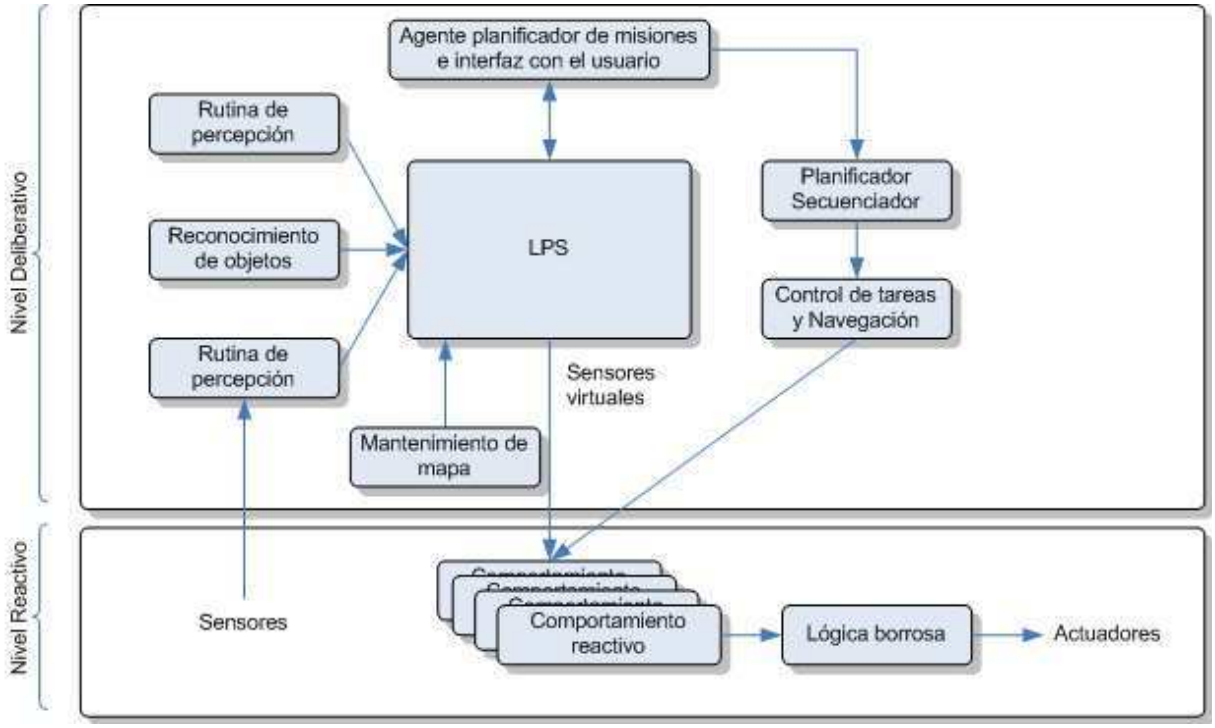


Figura 18: Arquitectura Saphira *Fuente: [Poza and Posadas, 2009]*

Una de las características que diferencian a Saphira de otras arquitecturas es el uso de técnicas de lógica difusa para definir los comportamientos y las combinaciones entre estos. Además, los diferentes componentes de Saphira son completamente independientes, es decir, no tienen la necesidad de llevar a cabo su ejecución en el nodo correspondiente al robot sobre el que se están ejecutando.

2.3.3.b. TCA

La arquitectura TCA (Task Control Architecture) se basa en el concepto de control estructurado. Este concepto parte de elementos básicos deliberativos que pueden ser extendidos con otros elementos reactivos, de forma que la arquitectura combina comportamientos de ambos tipos. El control de tareas (Task Control) es la coordinación de diferentes componentes de percepción, planificación y ejecución en el robot, que permite alcanzar unos objetivos. Estas tareas de control incluyen también cómo determinar qué objetivo concreto debe atenderse a cada momento, cómo construir los planes con cualquier nivel de detalle, así como la monitorización del progreso y la realización de las correspondientes transacciones.

La arquitectura TCA dota al robot con diversos módulos que ejecutan tareas específicas, y que se comunican entre sí mediante un módulo central de control, enviando mensajes a este componente. Todos los módulos de la arquitectura son construcciones de control que pueden llevar a cabo tanto comportamientos deliberativos como reactivos, tal y como se ha comentado anteriormente.

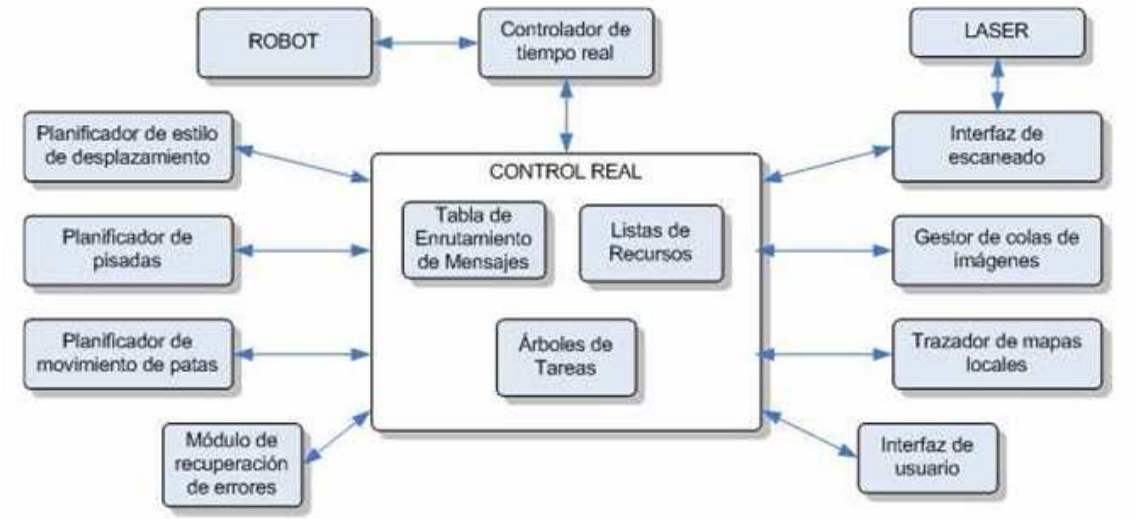


Figura 19: Arquitectura TCA *Fuente: [Poza and Posadas, 2009]*

Como se observa en la figura 19, el componente central de esta arquitectura está formado por una representación jerárquica de tareas y subtareas; se trata de un gestor de mensajes entre el resto de módulos. La estructura del sistema la generan los propios módulos con las prioridades dadas a los mensajes que se intercambian. Al ser la arquitectura jerárquica y dinámica, le otorga gran potencialidad.

2.3.4. Organizadas en niveles

Por último, es interesante presentar brevemente otros modelos de arquitecturas híbridas, que aunque menos comunes, también han sido utilizados a lo largo del tiempo. Las arquitecturas que se van a exponer a continuación no pueden clasificarse en ninguno de los tipos anteriores, pero todas poseen en común una organización en niveles diferenciados, en donde la organización no depende de características temporales o espaciales, como ocurría en los apartados anteriores.

2.3.4.a. GLAIR

La arquitectura GLAIR, definida en [Hexmoor, 1993] y [Lammens, 1993] se organiza, tal y como muestra la figura 20, en tres niveles diferenciados: nivel de conocimiento (KL), nivel Perceptuo-Motor (PML), y nivel Senso-Actuador (SAL). La arquitectura GLAIR, basada en comportamientos, aúna un sistema simbólico tradicional con un sistema físico. Se caracteriza por la existencia de estos tres niveles distintos, cada uno de los cuales posee su propia representación y su propio mecanismo de implementación, en especial, se caracteriza por el nivel de conocimiento. Estos tres distintos niveles distribuyen tanto la representación y la percepción, como el razonamiento y la generación de comportamientos. El punto más interesante de esta arquitectura es que los robots son empotrados en el entorno dinámico, y su ejecución se basa en la interacción y reacción continua, con la que exhiben un comportamiento inteligente.

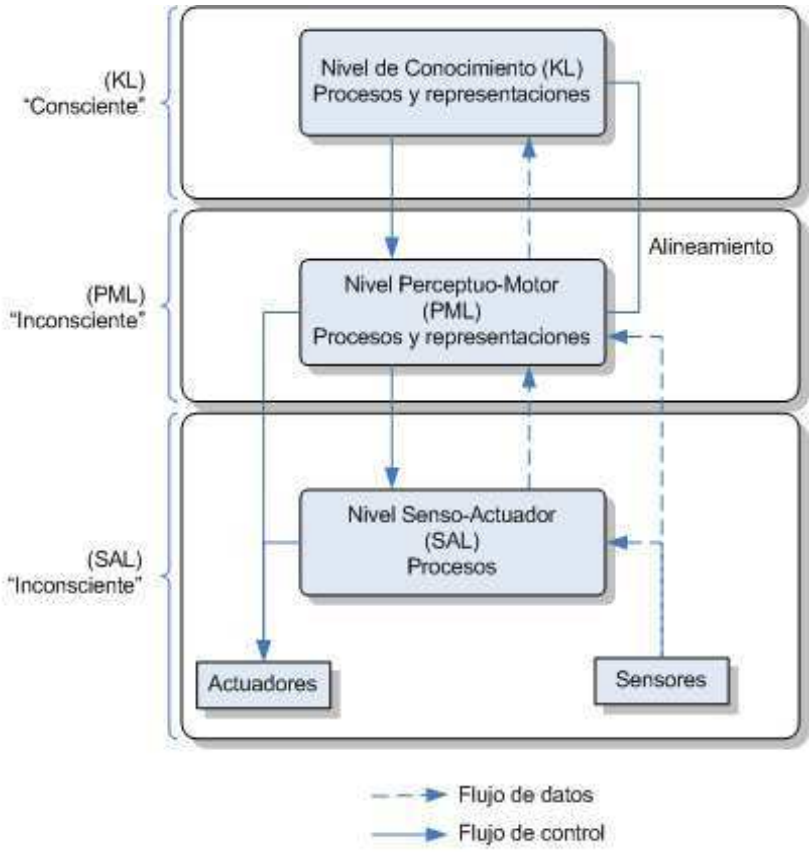


Figura 20: Arquitectura GLAIR Fuente: [Poza and Posadas, 2009]

Pasando a la descripción de cada nivel de la arquitectura, el primer nivel (conocimiento) está formado por un sistema planificador y hace uso de una representación selectiva de los diferentes elementos del entorno (objetos, eventos y estado) en función del curso que esté tomando la acción. Con esto se consigue modelar únicamente las entidades relevantes de la interacción del robot con su entorno. Por otro lado, el nivel perceptivo-motor hace uso de una representación más refinada de dichos elementos, proporcionando un gran detalle en ella que hace posible un control muy preciso de los actuadores. A su vez, los sensores también deben ofrecer un gran nivel de detalle. Para terminar, el nivel senso-actuador se encarga de las acciones motoras y sensoriales más básicas, donde se encuentran los "reflejos". Estos reflejos son bucles de bajo nivel que comunican sensores y actuadores, operan de forma independiente a los niveles superiores y se encargan de "expulsar" las acciones de estos niveles.

Lo que hace interesante a esta arquitectura es la división en tres niveles, de forma que se asocian comportamientos psicológicos a éstos. Esta característica hace que

se asimile perfectamente la conexión entre arquitecturas y aquellos sistemas reales que implementan.

2.3.4.b. BERRA

La arquitectura BERRA (Behaviour-based Robot Research Architecture) [Lindström, 2000] se desarrolló para una aplicación muy determinada: un robot de servicio que lleva a cabo multitud de misiones diferentes en un entorno de oficina. Se tuvo muy en cuenta que la arquitectura debía dar soporte a multitud de conceptos para que el robot pudiera llevar a cabo las tareas correctamente. Algunas de las características que tenía que tener en cuenta la arquitectura era un entorno conceptual reutilizable, la distinción de niveles de competencia, la simplicidad a la hora de integrar nuevos componentes, la flexibilidad, un rendimiento lo más eficiente posible en tiempo de ejecución, y que además fuera fácil de depurar.

Esta arquitectura también se divide en 3 capas (ver figura 21), como las 3T y ATLANTIS. La capa deliberativa es capaz de llevar a cabo los servicios requeridos, y la capa reactiva es capaz de reaccionar a situaciones inesperadas. A la hora de seleccionar las tareas, BERRA utiliza una estrategia de planificación y configuración [Arkin, 1998]. Esto implica que los módulos reactivos pueden configurarse y conectarse entre ellos, haciendo uso de una red flexible.

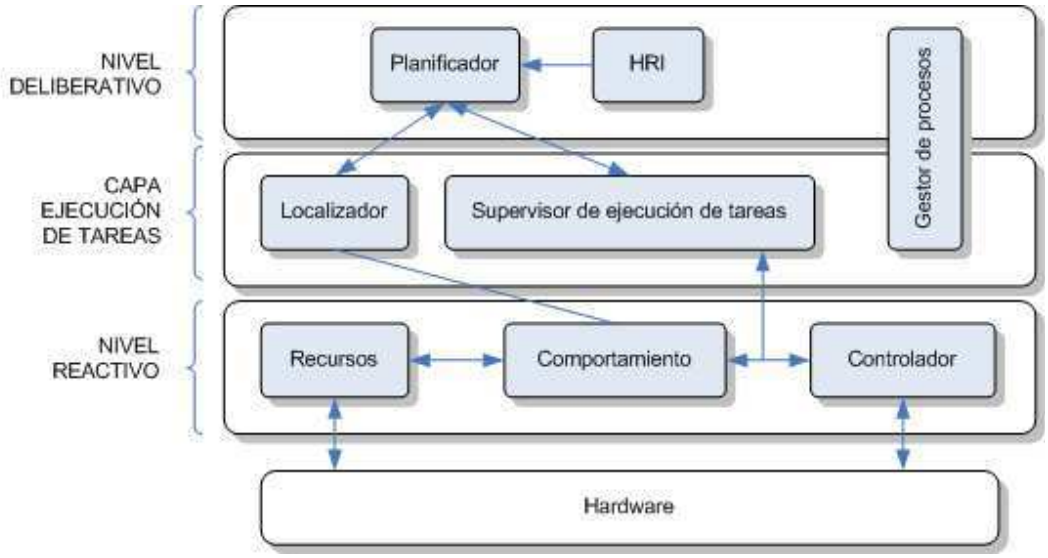


Figura 21: Arquitectura BERRA Fuente: [Poza and Posadas, 2009]

La capa superior (deliberativa) se encarga de recibir comandos. Estos comandos deben ser inteligibles por el sistema y ser enviados por un operador humano. En esta capa se recoge también las misiones de localización, conocimiento del entorno y planificación de trayectorias, que permiten al robot moverse entre distintos emplazamientos de su entorno. En la capa inferior (reactiva), basada en comportamientos, se proporciona la conexión entre sensores y actuadores. Los sensores pueden ser utilizados por distintos comportamientos simultáneamente, lo que exige tener un mecanismo de información compartida por los sensores.

La capa intermedia, destinada a la ejecución de tareas, está compuesta por un componente llamado supervisor de la ejecución de tareas (TES). Este componente se encarga de la gestión de la capa inferior, recibiendo por parte del planificador un cambio de estado, y transmitiéndolo a los elementos reactivos. También forma parte de esta capa el localizador, que se encarga de almacenar información a largo plazo, pero no es un componente con relevancia para la capa inferior. Su función principal es, tal y como su propio nombre indica, la localización del robot dentro de su entorno.

2.3.4.c. Otras

Otras arquitecturas que resultan interesantes son Sharp, SSS y Payton's Architecture.

La arquitectura Sharp [Laugier, 1998] también es una arquitectura de tres capas. Como la arquitectura BERRA, surgió como solución a una aplicación concreta (navegación de un vehículo por una red de carretera), aunque puede extenderse a muchas más áreas. Sharp utiliza como base los niveles de las arquitecturas en tres capas vistas con anterioridad (capa deliberativa, capa secuencial y capa reactiva) y los transforma en un planificador, un programador de misiones y un controlador de movimientos. Todos estos componentes pueden observarse en la figura 22.

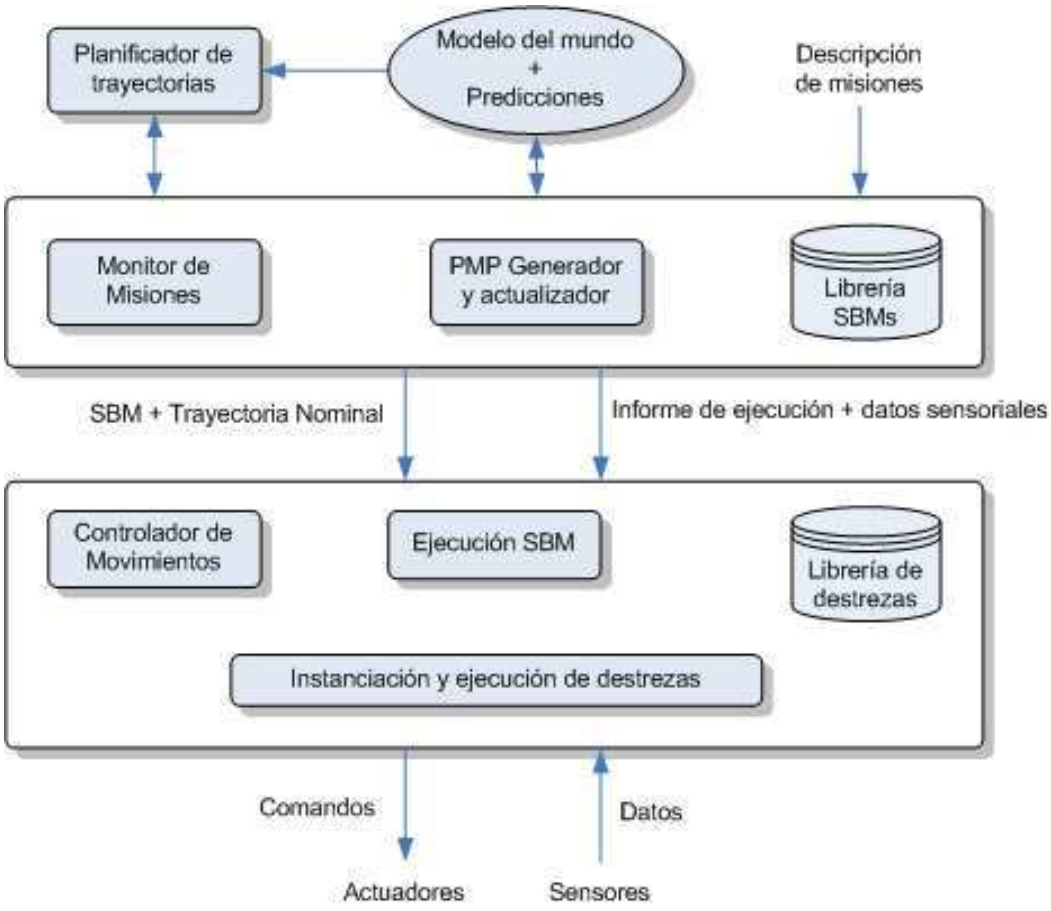


Figura 22: Arquitectura Sharp *Fuente: [Poza and Posadas, 2009]*

Esta arquitectura permitió mejorar la eficiencia y robustez de los sistemas autónomos de navegación por carretera, objetivo que se consiguió mediante la construcción "online" de los planes. Este concepto pasó a llamarse "Sensor-Based-Maneuver" (SBM), y puede entenderse como una "meta-destreza" de los vehículos que hacían uso de esta arquitectura. El concepto de SBM se basa en otro concepto clásico llamado "*script*" o guión.

Por otro lado, SSS (Servo, Subsumption, Symbolic) es también una arquitectura de tres capas (como se puede comprobar, esta estructura es de las más comunes dentro del paradigma híbrido) que combina una capa de servo-control, una basada en la subsunción y una tercera capa simbólica [Connell, 1992]. Esta arquitectura toma las mejores características de un sistema tradicional de servo-control para combinarlo con otros sistemas de inteligencia artificial, como los sistemas multi-agentes.

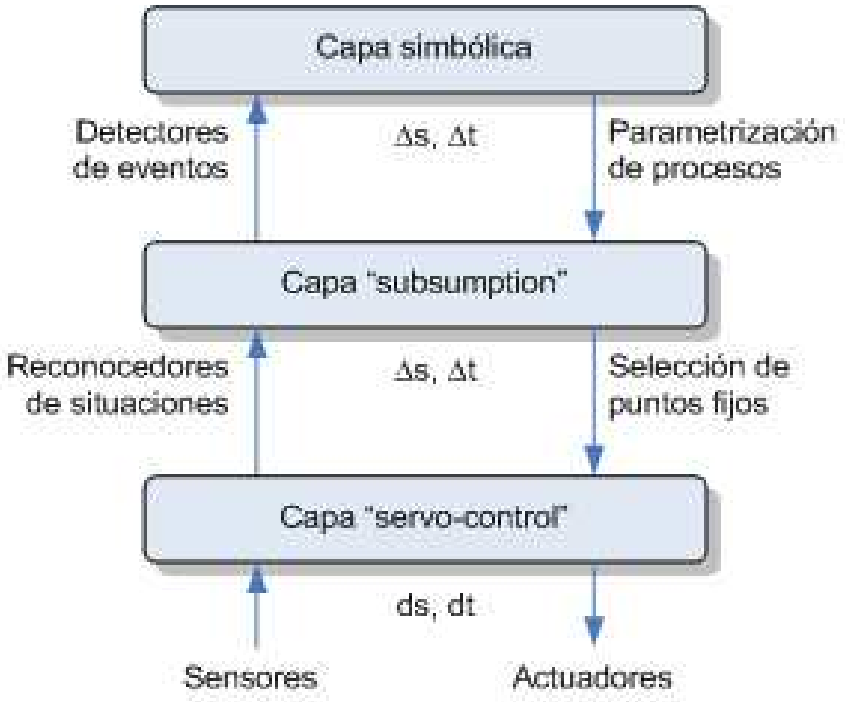


Figura 23: Arquitectura SSS Fuente: [Poza and Posadas, 2009]

Las tres capas de esta arquitectura (ver figura 23) cuantifican progresivamente el espacio y el tiempo. La primera de las capas (servo-control) trabaja en ambos dominios continuos. La capa de subsunción comprueba de forma constante los sensores, pero en este caso suelen ser situaciones concretas (no continuas) obtenidas por medio de la capa inferior. La capa simbólica discretiza aún más el tiempo, utilizando para ello eventos significativos. No existe en esta arquitectura una capa en la que se realice una discretización en tiempo previa a la discretización del espacio. Esto se debe a que los eventos temporales son discretizados a partir de cambios en las situaciones espaciales.

Para terminar con las arquitecturas híbridas, Payton's Architecture [Payton, 1986] se basa en una descomposición jerárquica. Cada capa de la arquitectura se caracteriza por un procesamiento diferente de los datos proporcionados por los sensores. Esta arquitectura está compuesta de capas que forman un sistema de percepción con cuatro módulos principales (ver figura 24).

En primer lugar, se tiene un planificador de misiones encargado de generar una secuencia de objetivos geográficos, que se pretenden alcanzar cumpliendo siempre con ciertas restricciones en el movimiento del robot. Este planificador se basa en un mapa del modelo del mundo, y es capaz de generar trayectorias mediante la conexión de los

objetivos geográficos definidos en la capa superior. Este primer planificador posee un tiempo de respuesta alto (del orden de minutos). Por otro lado, también se tiene un planificador local, que indica los detalles de los movimientos dentro de los caminos definidos anteriormente. Este segundo planificador posee un tiempo de respuesta bajo (del orden de segundos). Por último, un planificador reflexivo controla la ejecución de las tareas relacionadas con el movimiento del robot, y lo hace en tiempo real.

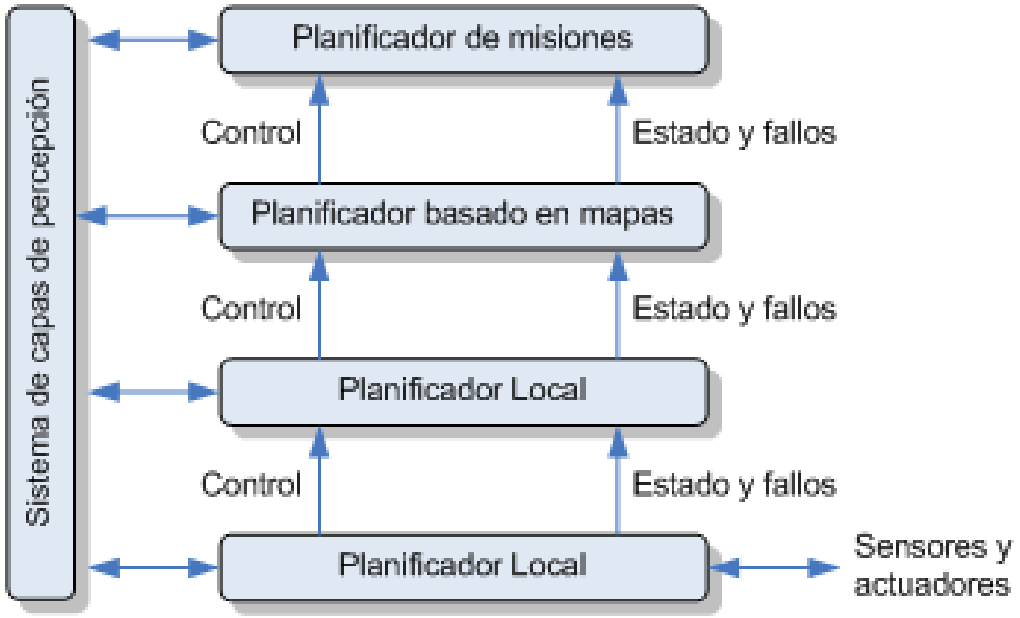


Figura 24: Payton's Architecture Fuente: [Poza and Posadas, 2009]

La implementación de la arquitectura Payton's se ha realizado mediante el uso de agentes expertos comunicados por una pizarra distribuida. Al usar este método, se puede controlar la actividad de un módulo de la arquitectura por una capa superior mediante la selección de los agentes expertos adecuados que activen dichos módulos. En cuanto a los comportamientos reflexivos, pueden asociarse con sensores virtuales para poder proveer información especializada. La comunicación prioritaria que puede llevarse a cabo mediante la pizarra distribuida es lo que permite que se activen los comportamientos reflexivos más apropiados en cada situación.

En el lado negativo, las principales limitaciones de la arquitectura son causa del mecanismo de comunicación entre las capas y por las limitaciones a la hora de combinar los comportamientos.

2.3.5. Comparativa

Una vez vistas diferentes arquitecturas que utilizan los diversos paradigmas de la robótica, es hora de compararlas. Esta comparativa y valoración fue de gran utilidad a la hora de realizar el análisis del sistema y seleccionar una de las arquitecturas, para cumplir los objetivos del proyecto de la mejor forma posible.

Entre las características de todas las arquitecturas que se han expuesto, con más o menos detalle, a lo largo de este apartado, se han seleccionado algunas de ellas, especialmente las que no se caracterizan por la especificidad de los componentes de la arquitectura. También se han seleccionado aquellas características que permiten alcanzar los diferentes objetivos principales del proyecto, como por ejemplo la modularidad de la arquitectura.

Según los autores del análisis de arquitecturas de control distribuido [Poza and Posadas, 2009], el nivel de modularidad de cada arquitectura se entiende como *"lo divisible que puede ser el control y por tanto lo adaptado que está a una situación concreta con pequeños cambios"*. No sólo se tiene en cuenta la modularidad de cada arquitectura, sino también la de la capa deliberativa, o sea, cómo se ha realizado el diseño de los diferentes módulos de esta capa, teniendo en cuenta las condiciones concretas sobre las que va a funcionar cada arquitectura. De esta forma, la modularidad es uno de los mejores indicadores de la capacidad de distribución de la arquitectura, y por tanto, de la escalabilidad de ésta, característica fundamental de los objetivos del presente proyecto.

En el estudio pormenorizado de las diferentes arquitecturas de control distribuido [Poza and Posadas, 2009], se obtuvo una tabla comparativa que se detalla a continuación:

Arquitectura	Tipo	Subtipo	Modularidad reactiva	Modularidad deliberativa	Modularidad general
Subsunción	Reactiva	Jerárquica	Alta	Alta	Muy alta
JPL	Deliberativa	Secuencial	Baja	Baja	Muy baja
NASREM	Deliberativa	Paralelo	Alta	Alta	Muy alta
AuRA	Híbrida	Organizativa	Alta	Baja	Media

Arquitectura	Tipo	Subtipo	Modularidad reactiva	Modularidad deliberativa	Modularidad general
SFX	Híbrida	Organizativa	Alta	Baja	Media
HILARE	Híbrida	Organizativa	Alta	Baja	Media
3T	Híbrida	Estados	Alta	Alta	Muy alta
ATLANTIS	Híbrida	Estados	Media	Alta	Alta
Saphira	Híbrida	Modelos	Media	Baja	Baja
TCA	Híbrida	Modelos	Media	Baja	Baja
GLAIR	Híbrida	Niveles	Baja	Media	Baja
BERRA	Híbrida	Niveles	Baja	Media	Baja
Sharp	Híbrida	Niveles	Media	Media	Media
SSS	Híbrida	Niveles	Media	Media	Media
Payton's	Híbrida	Niveles	Media	Media	Media

Tabla 1: Comparativa de arquitecturas Fuente: [Poza and Posadas, 2009]

Una de las arquitecturas más referenciadas en publicaciones y artículos sobre el este tema es la arquitectura de subsunción [Brooks, 1986], puesto que es una de las que implementa el control reactivo basado en niveles con prioridad, y lo hace de forma homogénea y elegante. La mayor prioridad la tienen los comportamientos básicos de supervivencia, y por otro lado los comportamientos deliberativos basados en el razonamiento poseen menor prioridad. Pero el principal problema de estas arquitecturas es que no funcionan de una forma deseable en entornos dinámicos, pues los comportamientos deliberativos pueden no aparecer debido a que los reactivos van a tener siempre una mayor prioridad ante situaciones que cambian constantemente.

Los aspectos más relevantes de la arquitectura NASREM [Albus and Barbera, 2005] son su adaptabilidad, gracias a la estructuración homogénea independientemente del nivel en el que se esté operando (nivel reactivo o nivel deliberativo), y su gran utilidad para el aprendizaje, gracias al método que utiliza para el almacenamiento y compartición del conocimiento global.

Otra característica importante en muchas arquitecturas de las presentadas es la reutilización de componentes en el control reactivo de las mismas. En la arquitectura AuRA [Arkin, 1990] se hace uso de esquemas reutilizables en esta capa para hacer adaptable la arquitectura a diferentes entornos. Una técnica muy parecida es de la que se

hace uso en la arquitectura SFX [Murphy, 2000], en la que se utilizan comportamientos comunicados con agentes en lugar de esquemas. En LAAS [Alami et al., 2000] se hace uso de módulos de control para llevar a cabo la división del control reactivo. Por último, en la arquitectura 3T [Bonasso, 1997] dicha reutilización la llevan a cabo las habilidades, que son activadas por eventos externos al sistema, y la planificación se realiza de una forma jerárquica.

Pasando al control deliberativo, la mayor parte de las arquitecturas poseen módulos altamente específicos para los diferentes usos que hacen éstas de este tipo de control. Aunque algunas de ellas sí poseen cierto grado de modularidad que permite mejorar su diseño y, en el caso de ejecutarse en un sistema distribuido, mejorar también la distribución del control deliberativo.

Entrando en arquitecturas concretas, el control deliberativo en la arquitectura de subsunción se presupone que forma parte de un comportamiento emergente a partir de los comportamientos reactivos del sistema. La mejor modularidad deliberativa se observa en la arquitectura NASREM, en la que los módulos deliberativos son tratados de forma similar a los reactivos, lo que potencia la homogeneidad de esta arquitectura. La modularidad deliberativa en la arquitectura 3T se consigue gracias a la jerarquía de objetivos, que se van transformando en objetivos y metas más concretas hasta alcanzar una asignación automática de tareas. Por último, en la arquitectura ATLANTIS, los módulos deliberativos se homogenizan a través de procesos LISP.

2.4. Plataformas Software para Robótica

Además de estudiar los diferentes paradigmas de la robótica y las arquitecturas más conocidas que se han desarrollado para el control automático de robots, también es importante conocer qué plataformas software están disponibles para el desarrollo de proyectos en este ámbito. Existen multitud de plataformas que permiten recrear con cierto nivel de detalle las características físicas de robots y los entornos en los que operan. En este apartado se van a mostrar algunas de ellas.

- **ERSP**: Se trata de la plataforma de desarrollo de Evolution Robotic [ERSP website]. Este software proporciona las herramientas para desarrollar sistemas

de navegación, visión e interacción entre robots. Las tecnologías más importantes que utiliza esta herramienta, y por las que se diferencia de otros software de desarrollo son ViPR (software de reconocimiento de patrones visuales), vSLAM (sistema de localización y mapeado visual) y ERSA (sistema operativo propio para los robots, que proporciona toda la infraestructura y herramientas para controlar y gestionar todo el hardware y software del robot).

- **Microsoft Robotics Studio**: Microsoft Robotics Developer Studio [MS Robotics website] es un entorno de desarrollo basado en Windows que facilita la creación de aplicaciones robóticas y da la posibilidad de desarrollarlas sobre una gran variedad de plataformas hardware. Las principales características de esta herramienta son la programación visual que facilita la creación y depuración de aplicaciones, los servicios en tiempo de ejecución y el entorno escalable y extensible.

- **Orocos**: Esta herramienta, Open Robot Control Software [Orocos website] es un marco de trabajo en tiempo real para el desarrollo de aplicaciones robóticas, nacido de un proyecto con el objetivo de encontrar alternativas a los software de control automático de robots de hace varias décadas, los cuales no eran demasiado eficaces. Lamentablemente, no dispone de entorno gráfico, aunque es una herramienta completamente “open-source”.

- **Skilligent**: Se trata de un sistema de control que permite la interacción con robots de servicio y realizar funciones de navegación, visión o incluso aprender comportamientos de humanos. Skilligent [Skilligent website] es una serie de paquetes software especialmente diseñado para su integración en robots controlados por PCs, cuyas principales características son un complejo sistema de visión robótica, un sistema de localización por visión, una interfaz robot-humano y un sistema de coordinación capaz de aprender tareas y habilidades.

- **Webots**: Es uno de los entornos de desarrollo de software para robots más

conocidos [Webots website]. Con esta herramienta es posible modelar, programar y simular gran cantidad de robots móviles. Una característica fundamental es que los controladores pueden ser desarrollados gracias a las propias herramientas que proporciona Webots, o mediante otros entornos de desarrollo. Se hablará más en detalle de este software en el apartado 3.1, en la descripción de las herramientas utilizadas para el desarrollo del presente proyecto.

- **Player-Stage-Gazebo**: El proyecto Player [Player website] comprende una serie de herramientas software para el desarrollo de aplicaciones robóticas y de sensores. Player es el conjunto de librerías que permite desarrollar controladores robóticos, pero a su vez es un servidor de red para aplicaciones cliente de control de robots. Las aplicaciones cliente utilizan TCP como protocolo de transporte para interactuar con el robot, permitiendo obtener información de los sensores y enviar órdenes al hardware. Por su parte, Stage y Gazebo son dos simuladores donde ejecutar las aplicaciones creadas con Player. Stage es un simulador en dos dimensiones diseñado para dar soporte a sistemas multi-agente en un entorno “bitmapped” y normalmente interior. Por su parte, Gazebo es un simulador en tres dimensiones diseñado para sistemas con una población de varios robots que interaccionan entre sí, normalmente en entornos exteriores, y proporciona un potente motor físico.

- **CARMEN**: Es una colección de software “open-source” para el control de robots móviles [Carmen website]. Es un software modular, diseñado para proporcionar funciones básicas de navegación, y que utiliza un protocolo de transporte propio (IPC) para comunicar los controladores con el hardware. Sin embargo, este software es uno de los más limitados que se pueden encontrar en el mercado.

Como se puede comprobar, existe una gran cantidad de plataformas software que facilitan el desarrollo de aplicaciones para el control automático de robots. Estos son tan sólo unos ejemplos. La mayoría de estas plataformas están en una fase inicial, y continúan en la actualidad siendo desarrolladas y mejoradas para dotar a los

programadores de este tipo de aplicaciones robóticas de las herramientas necesarias para conseguir comportamientos y funcionalidades de gran utilidad para los robots.

2.5. Proyectos

Para finalizar con el Estado del Arte, a continuación se presentan diferentes proyectos en el ámbito de la robótica. Algunos de ellos son productos software o librerías que tienen cierta fama dentro de este ámbito, y que son utilizados para llevar a cabo todo tipo de proyectos de robótica. El anexo G presenta otro proyecto que se ha considerado interesante mencionar, pero que no se enmarca por completo en el ámbito de este apartado.

Algunos de los proyectos descritos en este apartado fueron estudiados durante el proceso de documentación de este proyecto, para determinar si se hacía uso del software que proporcionaban a la hora de desarrollar la arquitectura híbrida. Por ejemplo, en un primer momento se estudió el uso de URBI, un lenguaje de scripts que forma parte del proyecto GOSTAI, para la implementación de los controladores del robot. Finalmente no fue utilizado, pero es importante y tiene un gran interés exponer de forma breve en qué consisten estos proyectos y por qué son la base de muchos trabajos en robótica en la actualidad.

2.5.1. Proyecto PROTEUS

El proyecto PROTEUS (Plataforma para la organización de la transferencia de conocimiento entre la comunidad robótica) [Martinet and Patin, 2008] se está llevando a cabo en Francia desde el año 2008 con el objetivo de crear un conjunto de herramientas que permitan facilitar el proceso de intercambio de conocimiento entre la comunidad académica e industrial en el ámbito de la robótica.

En este proyecto se tuvieron en cuenta dos consideraciones básicas a la hora de constituir el conjunto de herramientas que formarían PROTEUS [Patin, 2009]. En primer lugar, se tuvieron consideraciones de campo orientadas a través de la producción hacia el mundo académico de conjuntos de escenarios de traducción de problemas industriales. Es decir, se tiene en cuenta la capacidad de los involucrados en este campo

de usar robots reales operados por usuarios finales con el objetivo de evaluar directamente su éxito en materia de cognición o algoritmos de control, por ejemplo, en robots reales. Por otro lado, se tuvieron consideraciones de software orientadas a tener en cuenta herramientas para facilitar la transferencia de conocimiento, la creación de entornos ejecutables y metodologías para hacer estos recursos más fácilmente explotables por una gran comunidad de investigadores.

En cuanto al alcance del proyecto, debido a que la rama de robótica es un campo de trabajo muy grande y poco explorado hasta nuestros tiempos, se decidió limitar su consideración únicamente a los campos más comunes dentro de la robótica: robots terrestres, aéreos y, en algunos casos, humanoides.

Para implementar esta plataforma y verificar su eficacia y capacidad de dar respuesta a los problemas que pretende solucionar, se definió una metodología. Ésta permite, internamente, que PROTEUS pueda validar el conjunto de herramientas y estándares; y externamente, que valide su usabilidad por la comunidad académica e industrial.

Por tanto, el proyecto PROTEUS se dividió en dos partes principales, una dedicada al desarrollo de la propia plataforma y otra dedicada a su validación dentro del campo de trabajo. En la figura 25 se muestra el flujo de trabajo de PROTEUS.

De las múltiples tareas y procesos llevados a cabo para el desarrollo del proyecto PROTEUS, los más interesantes para este apartado son los tres bloques de trabajo principales comunes con este tipo de proyectos de robótica. Estos tres bloques de trabajo son el bloque de robótica, el bloque de lenguajes y herramientas, y el bloque de diseminación y explotación.

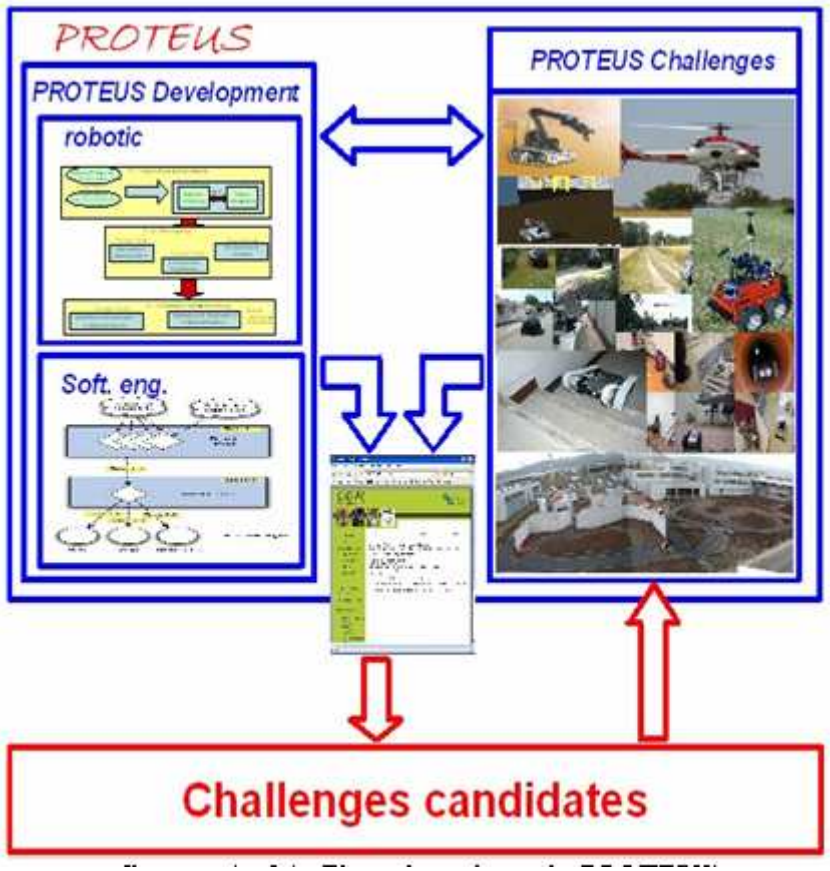


Figura 25: Flujo de trabajo de PROTEUS Fuente: [Martinet and Patin, 2008]

Dentro del bloque de trabajo de robótica (ver figura 26), los esfuerzos de los creadores de la plataforma se centraron en tareas tales como la elaboración de un completo estado del arte, la especificación de escenarios, robots involucrados y problemas asociados, la creación de una ontología robótica y su modelado, la generación de prototipos y, por último, la validación mediante la aplicación de la metodología. La figura 26 muestra la estructura y descripción de este bloque de trabajo.

El segundo bloque de trabajo tiene como objetivo la definición de lenguajes y herramientas que permitan la definición y simulación de escenarios, siendo un escenario la descripción de una serie de robots que llevan a cabo una misión en un determinado entorno. La tarea principal de este bloque es la de modelar un lenguaje específico del dominio (DSLs) que permita especificar misiones, entornos y comportamientos para los robots. Después, es necesario integrar estos lenguajes de dominio específicos en niveles ontológicos para, finalmente, pasar a la transformación de modelos y la generación de código.

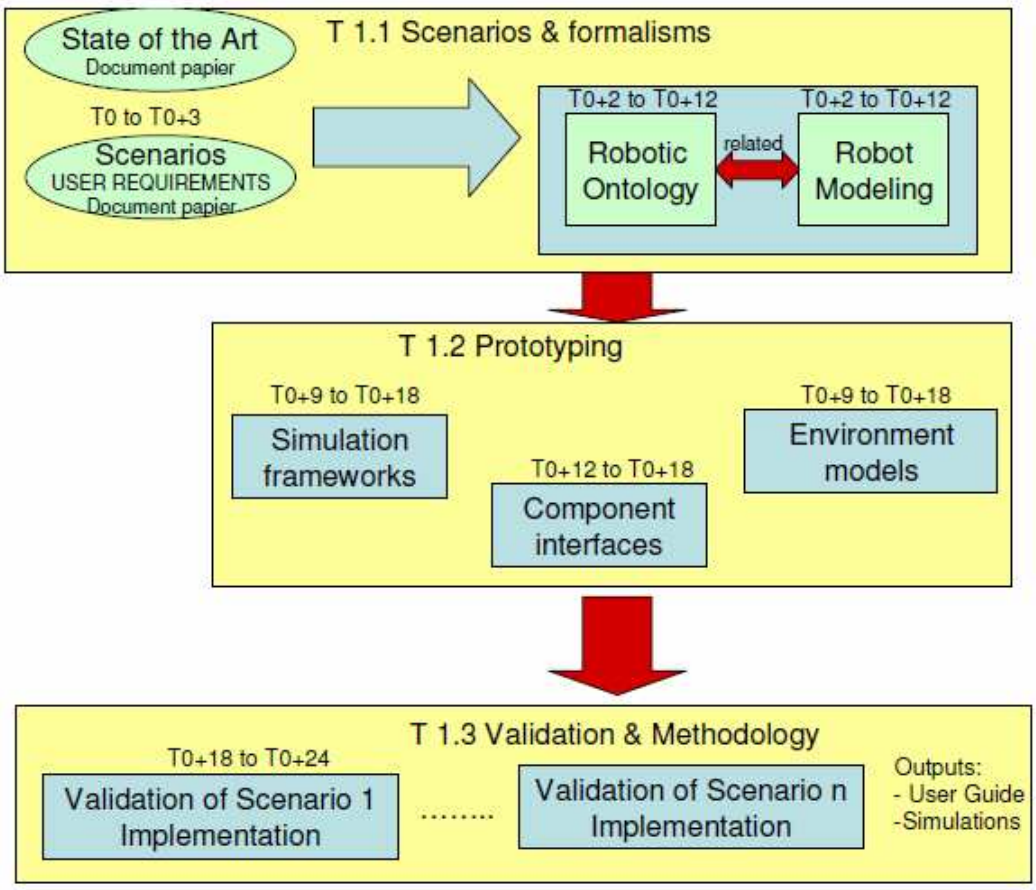


Figura 26: Bloque de trabajo de róbotica en PROTEUS Fuente: [Martinet and Patin, 2008]

El tercer bloque de trabajo surge como necesidad de asegurar uno de los objetivos más difíciles de conseguir del proyecto PROTEUS; la continuidad del uso de la plataforma por parte de la comunidad de robótica tras la finalización del proyecto propiamente dicho. Para ello, es necesaria la distribución de la información sobre el proyecto en diversas conferencias y la explotación de la plataforma.

Dentro del proyecto PROTEUS, es importante mencionar dos de los simuladores de entornos robóticos utilizados en el proyecto: MORSE y Cycab ToolKit (CycabTK).

2.5.1.a. Simulador Morse

MORSE [Echeverria et al., 2010], simulador robótico "*open-source*", proporciona un entorno virtual para el control de robots mediante una arquitectura

basada en componentes que simula sensores, actuadores y robots. Se trata de un simulador muy flexible, capaz de utilizar diversos niveles de abstracción para la simulación de acuerdo a los diferentes sistemas concretos que se estén probando. Es capaz de representar una gran cantidad de robots y simular entornos con múltiples robots.

MORSE utiliza una filosofía de "*Software-in-the-loop*", que ofrece la posibilidad de evaluar los algoritmos embebidos en la arquitectura software del robot en la que posteriormente van a ser integrados. Esto hace, además, que el simulador sea independiente de cualquier arquitectura que posea el robot y su marco de comunicaciones.

Por otro lado, MORSE fue construido sobre Blender, aplicación también "open-source" de modelado y renderizado 3D, cuyo propósito es la generación por ordenador de imágenes y animaciones en 3D (ver figura 27). MORSE utiliza las características de Blender y extiende su funcionalidad a través de scripts en Python. Las simulaciones realizadas con esta herramienta son ejecutadas en modo "*Blender's Game Engine*", que proporciona una visualización gráfica realista de los entornos simulados y permite explotar su motor físico, llamado Bullet.

Figura 27: Imagen tomada del simulador MORSE

El desarrollo de este simulador surgió como alternativa a los simuladores robóticos más populadres, como Player/Stage [Gerkey, 2003] y Gazebo [Koenig, 2004]. La principal limitación de estos simuladores es que utilizan entornos 2D, en lugar de

simular entornos más realistas en 3D. Otra de las motivaciones principales para el desarrollo de MORSE fue hacerlo reutilizable por investigadores e ingenieros, puesto que su desarrollo formó parte de diferentes proyectos, cada uno con diferentes restricciones y requisitos, pero todos orientados a sistemas basados en la interacción y cooperación entre múltiples robots.

También es importante comentar la característica que hace que MORSE pueda integrarse muy fácilmente con cualquier middleware utilizado en robótica, como por ejemplo ROS (ver apartado 1.4.2). El middleware se diseña para conectar componentes separados, de forma que si un elemento está altamente ligado a un middleware, hace que este elemento sea muy complicado de reutilizar en un entorno diferente. MORSE sigue una filosofía que permite mantener el código de un componente completamente separado del middleware, y ligarlo únicamente cuando es necesario. Una vez el módulo es generado, puede conectarse al resto del software del robot e intercambiar datos por medio de estructuras y peticiones.

2.5.1.b. Cycab-Tk

Cycab ToolKit [CycabTK website] ofrece drivers para robots móviles, algunos sensores como láseres, GPS o cámaras, y un simulador 3D (ver figura 28). Este conjunto de simulador y drivers proporciona a los usuarios de Cycab un marco de desarrollo y prueba de aplicaciones para proyectos de robótica. Este marco de desarrollo, además, está basado en Hugr, un middleware que permite compartir variables entre diferentes programas y que puede ser utilizado independientemente por cualquier aplicación robótica o de sensores.

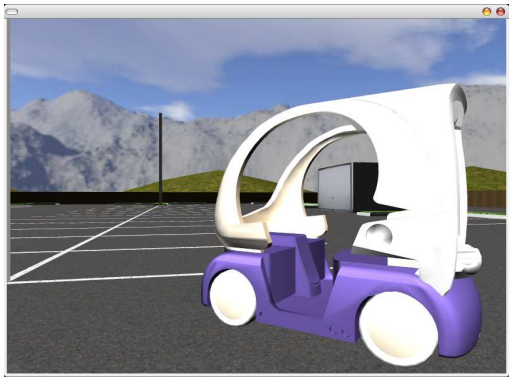


Figura 28: Captura del simulador CycabTK

Este simulador, sin embargo, está más limitado que otros, al contrario que MORSE, debido a que es utilizado básicamente para la simulación de proyectos basados en robots Cycab, también llamados Cyber Cars, que pueden verse en la figura 29. Estos vehículos robóticos utilizan tecnología de gran precisión en GPS, láseres y cámaras para una navegación segura por rutas programadas.

Figura 29: Cyber Car

2.5.2. ROS.org

ROS es un meta-sistema operativo “open-source” para robots [ROS website]. Proporciona los mismos servicios que cualquier sistema operativo (abstracción a nivel harware, control de dispositivos de bajo nivel, etc.), pero también ofrece herramientas y librerías para obtener, construir, escribir y ejecutar código en múltiples ordenadores, de forma que se considera similar a un marco de desarrollo robótico, como los vistos hasta ahora. Aunque no se trata de un marco de desarrollo de tiempo real, es posible integrar ROS con código de tiempo real.

El principal objetivo de ROS, como el de la mayoría de herramientas de desarrollo de robótica, es permitir la reutilización de código en el desarrollo e investigación robótica. ROS es un marco de trabajo de procesos distribuidos que permite diseñar individualmente los ejecutables y acoplarlos de forma flexible en

tiempo de ejecución. Pero ROS también soporta un sistema de código de repositorio que permite la distribución de código e información entre la comunidad de investigación robótica. Este diseño, desde el nivel de sistema hasta el nivel de comunidad, posibilita decisiones independientes sobre desarrollo e implementación, pero todas pueden llevarse a cabo con las herramientas de infraestructura proporcionadas por ROS.

Junto con este objetivo principal de ROS, hay otras metas que se han buscado con el desarrollo de este meta-sistema operativo para robótica, tales como: facilidad de integración con otros marcos de trabajo robóticos (Player, por ejemplo), conseguir un modelo de desarrollo con librerías sencillas, "limpias" y funcionales, independencia del lenguaje de programación (Python, C++, Java, etc.), facilidad de depurado y prueba, y escalabilidad, permitiendo sistemas con un gran número de procesos y tiempos de ejecución grandes.

2.5.3. GOSTAI

GOSTAI es una de las empresas más importantes del panorama robótico actual [GOSTAI website]. Ofrece una gran cantidad de productos, tanto hardware como software, para el desarrollo de proyectos de robótica. GOSTAI centra sus esfuerzos en tres actividades diferenciadas:

- La robótica de consumo, especialmente en el mercado de robots de bajo coste, proporcionando aplicaciones robustas y realmente avanzadas para el entretenimiento, seguridad, vigilancia y cuidados.
- La robótica industrial, con el propósito de alcanzar desafíos y metas en términos de complejidad, seguridad y conformidad con estándares internacionales.
- La investigación, involucrándose en gran cantidad de proyectos internacionales y contribuyendo a la investigación en el ámbito de la robótica y la inteligencia artificial mediante su programa de apoyo académico.

En materia de hardware, Jazz es el robot comercial más conocido de GOSTAI. Se trata de un robot de telepresencia a control remoto capaz de moverse, ver, ser visto, oír y hablar en un lugar distante como si el propio usuario estuviera en dicho lugar.

Entre las funcionalidades básicas de este robot se encuentran las ya nombradas (ver, oír, hablar, etc.), es operado de forma remota mediante una interfaz web, es capaz de evitar obstáculos, detectar movimiento o llevar a cabo patrullas autónomas. Tiene tres diseños distintos, dependiendo de la funcionalidad básica requerida: telepresencia, seguridad mediante videovigilancia y ocio y entretenimiento, cada uno de ellos con sus propias características y funcionalidades.

Entre los productos software que ofrece GOSTAI, se encuentran entornos de desarrollo, simuladores y lenguajes de script. Entre los primeros destacar GOSTAI Suite, un entorno de desarrollo que incluye GOSTAI Studio y GOSTAI Lab. El primero es un editor gráfico de comportamientos con el que es posible crear máquinas de estado finito jerárquicas y librerías que posteriormente pueden ser reutilizadas por la comunidad, y visualizar la ejecución en tiempo real, entre otras cosas. Por su parte, GOSTAI Lab es un control remoto universal que facilita la programación e interacción con el robot real, monitorizando sus sensores o enviando comandos directamente mediante un terminal. Este entorno de desarrollo se basa en una arquitectura cliente-servidor.

El simulador Webots no es un producto de GOSTAI, sino de la empresa Cyberbotics, pero ambas colaboran mutuamente para ofrecer a la comunidad diversos productos que ayuden en sus proyectos. Se explicará en detalle más adelante en este documento, ya que es la herramienta utilizada para llevar a cabo las simulaciones de la arquitectura implementada, antes de pasar a las pruebas sobre el robot real. Básicamente, Webots es uno de los simuladores de robótica más utilizados en la comunidad académica internacional, pues ofrece un entorno 3D realista y fácil de aprender y utilizar. El verdadero producto de GOSTAI es URBI, un lenguaje de script que complementa a Webots para el desarrollo de controladores para robots móviles autónomos.

Urbiscript es el lenguaje de programación propiamente dicho, diseñado especialmente para trabajar en el ámbito de la robótica. Se trata de un lenguaje de script dinámico, basado en prototipos y orientado a objetos. Soporta y permite hacer énfasis en programación paralela y basada en eventos, paradigma muy común en robótica, proporcionando primitivas y construcciones para este fin. Por otro lado, el URBI SDK

(Software Development Kit) es el entorno que proporciona los medios para integrar organizaciones de componentes muy complejas, típicas en estos sistemas. Se basa en una arquitectura middleware que coordina componentes llamados Uobjects para hacer posible el uso de componentes remotos como si fueran locales, lo que permite ejecutar de forma concurrente para realizar peticiones síncronas y asíncronas de datos al robot.

Capítulo 3

Implementación de la Arquitectura 3T para el Control del P3-DX

En este capítulo central del documento se detalla todo el trabajo realizado para el diseño y la implementación de una arquitectura híbrida. Esta arquitectura finalmente se basó en la arquitectura 3T vista en el apartado 2.3.2.a. En los siguientes apartados se explica desde el hardware y software utilizado para llevar a cabo este desarrollo, hasta los requisitos identificados en el análisis del problema y el diseño (tanto arquitectónico como detallado) del sistema.

Además, se explica en detalle cómo funciona la arquitectura y qué tipo de comportamientos inteligentes se han implementado para el robot; comportamientos que posteriormente permitieron probar el funcionamiento descrito. Por último, se referencia al lector a los anexos C, D, E y F. El anexo C contiene los diagramas mostrados en el apartado 3.3 a mayor tamaño para facilitar su lectura. Los anexos D y E contienen las tablas de componentes y clases del diseño arquitectónico y detallado. Por último, el anexo F contiene las librerías de algunos de los elementos característicos de la arquitectura que se han incluido en el dominio de las pruebas sobre el robot.

3.1. Hardware y Software Utilizado

Este apartado describe las herramientas, tanto hardware como software, que se han utilizado para el desarrollo del proyecto. Las herramientas más importantes son el propio robot Pioneer 3-DX, la interfaz que proporciona para programar los controladores y un simulador de mundos virtuales y robots móviles. También se ha hecho uso de otras herramientas en diversas fases del proyecto, que se describen brevemente al final del apartado.

3.1.1. Robot Pioneer 3-DX

Los robots Pioneer pertenecen a una familia de robots móviles, de dos y cuatro ruedas, fabricados por Mobile Robots [MobileRobots website]. Puede verse una imagen del P3-DX en la figura 30. Todos los robots pertenecientes a esta familia (Pioneer 1, Pioneer AT, Pioneer 2-DX, Pioneer 3-DX, etc) comparten una arquitectura común para investigación y desarrollo de aplicaciones robóticas.

Figura 30: Robot Pioneer 3-DX de Mobile Robots

Este robot es uno de los más pequeños del mercado, pero tiene la capacidad de llevar a cabo gran cantidad de comportamientos inteligentes, previamente programados, gracias a las especificaciones técnicas de esta familia de robots. Apenas mide 22 cm. de alto y menos de 40 cm. de diámetro. La figura 31 muestra las medidas de este pequeño robot.

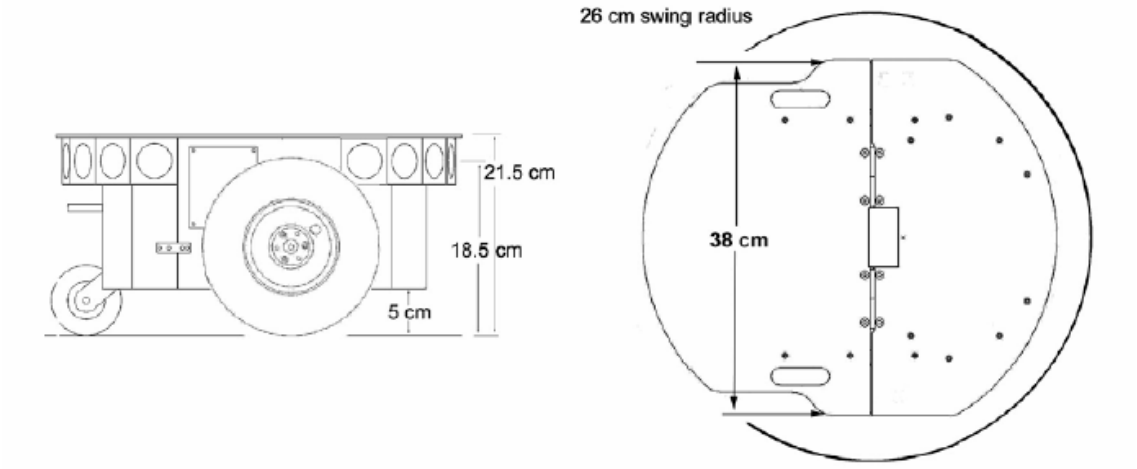


Figura 31: Dimensiones del robot Pioneer 3-DX

Su peso es de alrededor de 10 Kg. con una única batería, pero puede soportar hasta 23 Kg. de carga adicional, pudiendo incorporar elementos como portátiles, brazos robóticas, cámaras o láser.

A continuación se describen los elementos más importantes del Pioneer 3-DX, que han sido utilizados durante el desarrollo del proyecto y, a los que se hace mención en próximos capítulos del presente documento.

Panel de control: Consiste en un panel de botones e indicadores que permiten controlar el robot y obtener información del microcontrolador incorporado en éste. Posee un puerto serie. La figura 32 muestra el panel de control del robot, y a continuación se explica brevemente cada elemento.

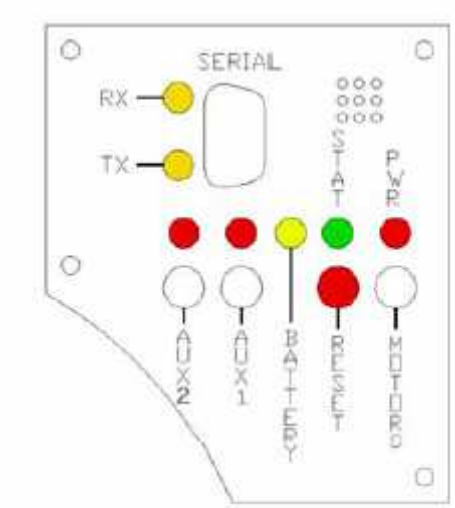


Figura 32: Panel de control del Pioneer 3-DX

- **Serial:** Puerto de conexión serie, a través del cual se realiza la conexión con el robot. Posee dos indicadores: RX (indicador de entrada de datos) y TX (indicador de salida de datos).
- **PWR:** Indicador de encendido.
- **Stat:** Indicador de estado del robot. Parpadea lentamente cuando está a la espera de una conexión con el cliente. Si conecta con el cliente parpadeará rápidamente.
- **Battery:** Indicador del nivel de carga de la batería del robot. Con carga completa, el LED se encuentra verde, y va cambiando de amarillo a rojo a medida que la carga disminuye.

- **Motors:** Botón que cambia el estado de los motores, habilitándolos o deshabilitándolos.
- **Reset:** Reinicia el robot.
- **Aux1/Aux2:** Otros indicadores de conexiones auxiliares.

Ultrasonidos (sónar): El Pioneer 3-DX posee ocho sensores de ultrasonidos colocados en la parte delantera del robot. Dos de ellos se encuentran en las caras laterales, mientras que los otros seis se encuentran en la cara delantera, con una orientación de 20º de diferencia entre cada uno de ellos, lo que permite al robot detectar obstáculos que se encuentren en un ángulo de 180º con respecto al eje de dirección del robot. Para la detección de objetos, estos sensores emiten señales de ultrasonidos y miden el tiempo que tardan en recibir el eco de la señal para calcular la distancia a la que se encuentran los objetos. Como se puede ver en la figura 33, cada uno de los ocho sensores de ultrasonidos se identifica por un número de 0 a 7, de derecha a izquierda, mirando de frente al robot.

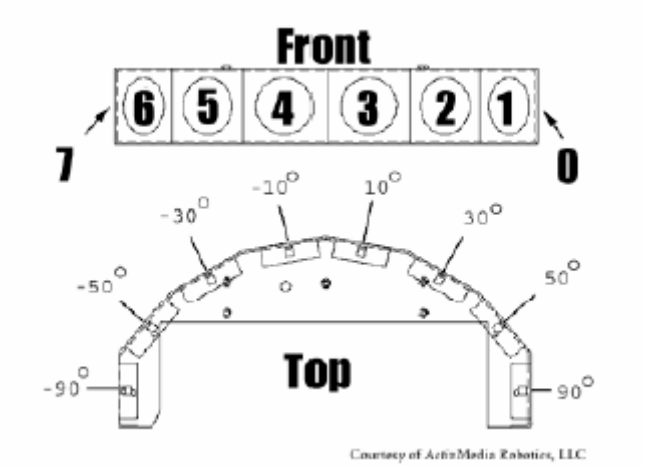


Figura 33: Disposición de los sensores de ultrasonidos

Parachoques (bumpers): Se trata de otro tipo de sensores que permiten al robot detectar colisiones con los objetos del entorno. El Pioneer 3-DX cuenta con diez parachoques: cinco delanteros y cinco traseros, colocados de tal manera que evitan los choques desde cualquier ángulo con el cuerpo del robot. Estos parachoques son sensores que envían una señal al microcontrolador sólo en caso de que se activen por una colisión. Mientras este hecho no se da, los parachoques no emiten ningún tipo de señal. La figura 34 muestra la disposición de los parachoques del robot.

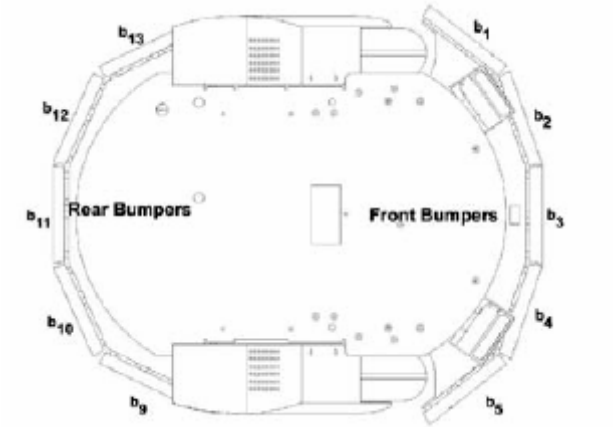


Figura 34: Disposición de los parachoques

Motores y ruedas: El sistema de dirección del robot cuenta con dos ruedas motrices, con sus respectivos motores, cada uno equipado con un "encoder" óptico de alta resolución que permite conocer la posición, giro y velocidad de las ruedas en todo momento. También posee una pequeña rueda posterior que aporta estabilidad al robot.

3.1.2. Simulador Webots

Webots es un simulador de robots móviles en tres dimensiones, desarrollado originalmente como una herramienta de investigación sobre algoritmos de control para robots móviles. Permite multitud de lenguajes de programación para la creación de controladores, tales como C, C++, Java, Phyton o Matlab, etc.

Ofrece un entorno de desarrollo que permite al usuario crear mundos virtuales 3D con propiedades físicas como masa, coeficientes de fricción, etc. En estos mundos pueden incluirse objetos pasivos o activos, estos últimos llamados robots móviles. Existen una gran variedad de robots móviles ya diseñados, como por ejemplo robots con ruedas, con piernas o incluso voladores. Además, estos robots pueden ser equipados con una gran cantidad de sensores y actuadores (ultrasonidos, sensores de choque, cámaras, GPS, etc.) Por último, el usuario puede programar el comportamiento de todos estos robots de forma individual para que muestren el comportamiento deseado.

Este entorno de desarrollo ofrecido por Webots posee una interfaz muy completa que facilita al usuario la creación de los mundos virtuales, la implementación de los

controladores de los robots y la monitorización de la simulación. En la figura 35 se muestra una captura de pantalla de la interfaz de Webots.

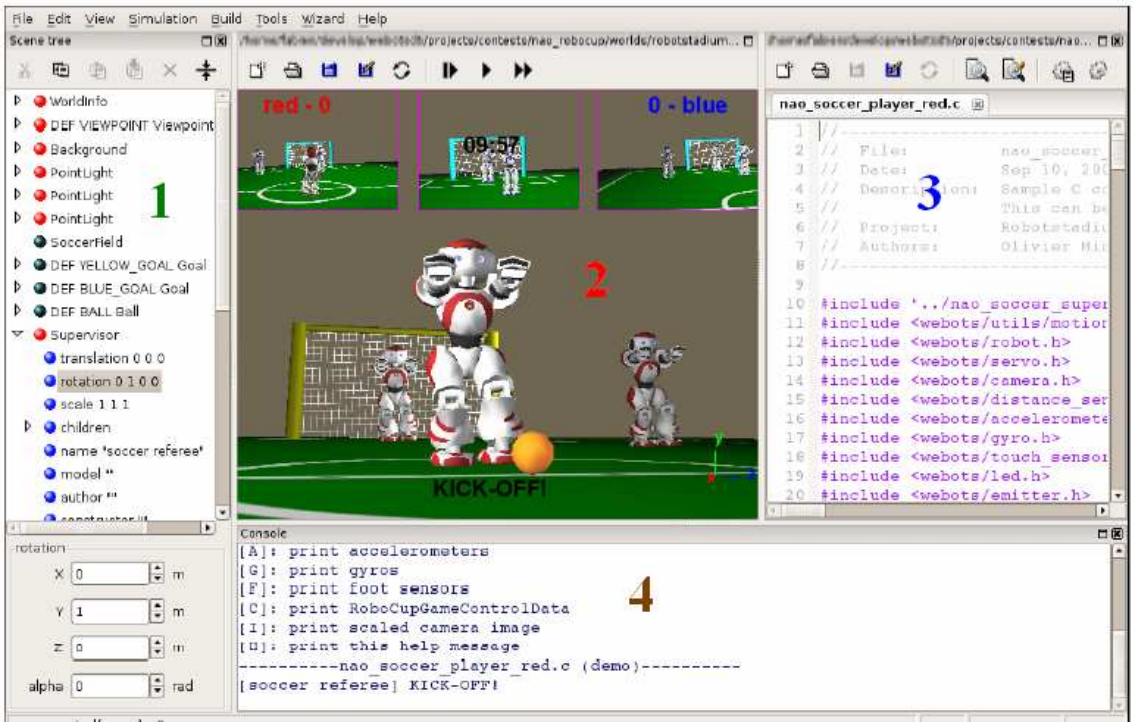


Figura 35: Interfaz gráfica de Webots

La interfaz se divide en cuatro marcos, que se explican a continuación:

- **Árbol de escena (1):** Muestra todos los elementos que forman parte del mundo virtual creado, permitiendo definir sus características. El mundo virtual se crea añadiendo elementos a este árbol, tales como sólidos (suelo, paredes, objetos), robots y luces, entre otras cosas.

- **Ventana 3D (2):** Muestra el mundo virtual 3D de forma gráfica, con todos los elementos que forman el árbol de escena. Esta ventana permite cambiar el ángulo y zoom de la cámara que muestra el mundo virtual, así como seleccionar los objetos, moverlos y rotarlos. Durante la ejecución de la simulación, la ventana 3D muestra los resultados de forma gráfica, pudiendo verse los robots en movimiento.

- **Editor de texto (3):** Este marco permite visualizar el código de los controladores implementados para los robots que forman parte de la escena. Se trata de un editor que permite el resaltado para los lenguajes de programación más comunes. Desde este editor es posible compilar el código fuente, pero una de las características del simulador es que también permite importar código objeto ya

compilado, copiando el archivo ejecutable en una ruta específica para el controlador que se vaya a utilizar.

- **Consola (4):** La consola muestra la salida por defecto del compilador y de los controladores utilizados. Aquí se pueden consultar los mensajes de error proporcionados por el compilador o por el propio código del controlador, y es una herramienta más orientada al programador para comprobar los resultados de la ejecución de la simulación.

Para realizar una simulación en Webots, se utilizan tres elementos, de los cuales dos son obligatorios, y es necesario crearlos para que la simulación se pueda llevar a cabo:

- Un archivo de mundo Webots (Webots World file .wbt): Define uno o más robots móviles y el entorno por el que se mueven. El archivo .wbt puede depender adicionalmente de archivos de texturas y prototipos (como por ejemplo robots ya programados en el simulador).
- Uno o varios archivos controladores de los robots: Pueden ser programados en multitud de lenguajes. Estos archivos son los que contienen las instrucciones a ejecutar por el robot en la escena del mundo virtual.
- Opcionalmente, un “plugin” de motor físico: Permite modificar el comportamiento normal del motor físico incluido en el simulador.

Finalmente, es importante comentar que el simulador Webots funciona en los tres principales sistemas operativos: Linux, Windows y MAC OS. Sin embargo, Webots no es “opem-source” y requiere comprar una licencia para poder utilizarlo fuera de los 30 días de prueba, que no obstante, no ofrece toda la funcionalidad del simulador.

3.1.3. ARIA

Todos los robots de Mobile Robots, incluido el Pioneer 3-DX, viene con un software llamado Advanced Robotics Interface for Applications (ARIA). ARIA es un entorno de desarrollo “open-source” basado en C++ que proporciona una interfaz cliente robusta para una gran variedad de sistemas robóticos inteligentes. Por otro lado, ARNetworking, proporcionado junto a ARIA, ofrece una capa de transporte TCP/IP de comunicaciones con la plataforma hardware.

ARIA proporciona las librerías necesarias para la integración de software de control propio con el robot que será controlado por dicha aplicación. Es una de las opciones más sencillas para integrar el software propio con el robot, puesto que ARIA se encarga de manejar de forma transparente todos los detalles de bajo nivel de la interacción cliente servidor: comunicaciones por red y/o puerto serie, procesado de paquetes de comandos e información del servidor, control de ciclo de reloj y paralelismo a nivel de hilos. Además, permite y facilita el uso de dispositivos y controles, como los diversos tipos de sensores y actuadores del robot.

La figura 36 muestra la arquitectura de ARIA:

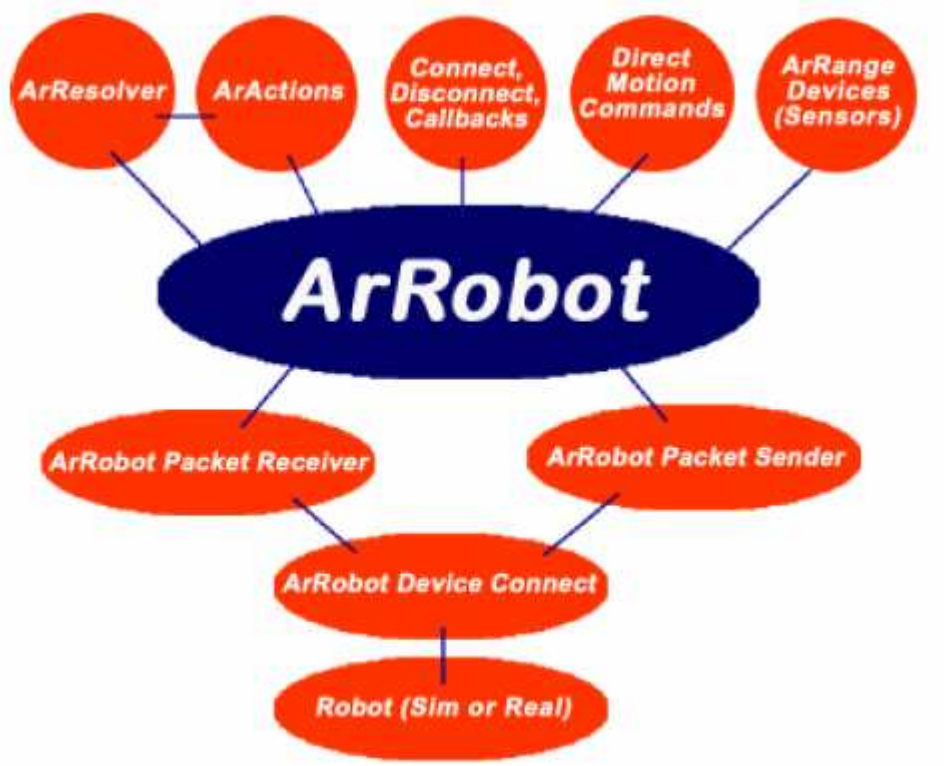


Figura 36: Arquitectura de ARIA

Como se puede observar, en la parte inferior de la figura se encuentra la plataforma sobre la que se trabaja (ya sea el robot real o un simulador). Por encima se encuentra el sistema de conexión con el robot y los gestores de paquetes enviados y recibidos. El elemento central de ARIA es la representación que ésta hace del robot, y que permite situar por encima las librerías que ofrece al desarrollador para el control de

éste, entre las que se encuentran las funciones de conexión y desconexión, comandos de movimiento, acceso a los dispositivos y accesorios tales como los sensores del robot, etc.

3.1.4. Otras herramientas

Aparte del hardware y software mencionado hasta ahora, durante el desarrollo del proyecto se han utilizado otras herramientas software y hardware. En este apartado se enumeran y describen brevemente qué tipo de herramientas son y en qué momentos y para qué se han utilizado cada una de ellas.

- **Herramientas de oficina (Microsoft Office y OpenOffice):** Se han utilizado las herramientas de oficina proporcionadas por Microsoft (MS Office Word, MS Office Excel y MS Office PowerPoint), así como el software libre OpenOffice (OO Writer) desde el comienzo del desarrollo del proyecto para elaborar los diferentes documentos asociados al proyecto: la presente memoria, el presupuesto y la presentación.

- **Lenguaje de programación Java:** A la hora de programar la arquitectura de control automático para el robot, se ha hecho utilizado el SDK 6.0 de Java. Se ha programado utilizando únicamente este lenguaje, apoyado en las librerías proporcionadas por el simulador Webots y por el software incluido en el robot Pioneer 3-DX (ARIA).

- **Entorno de desarrollo Eclipse:** Para facilitar el proceso de implementación de la arquitectura, que finalmente constó de un gran número de componentes y clases, se ha utilizado el entorno de desarrollo Eclipse. Con esta herramienta se ha implementado la arquitectura, y se ha utilizado este entorno para llevar a cabo las ejecuciones sobre el robot real (ya que en el simulador es el propio simulador el que sirve como entorno de ejecución).

- **Gantt Project:** Esta herramienta de gestión de proyecto se utilizó en las primeras fases del mismo para elaborar la planificación. Permite crear fases para el proyectos y asignarles una franja de tiempo, así como unas precondiciones del

tipo "*la fase X debe comenzar una vez que la fase Y haya finalizado*". Con esta herramienta se elaboró el diagrama Gantt de la planificación del proyecto, que puede consultarse en el siguiente apartado.

- **Altova UModel:** Esta herramienta de modelado software se utilizó en dos momentos diferentes del proyecto para elaborar el diseño arquitectónico y detallado de la arquitectura. Se trata de una herramienta software para diseñar sistemas, y ofrece unos diagramas muy claros y a su vez detallados. Sin embargo, su uso requiere la compra de una licencia, aunque se puede utilizar una versión de prueba de 30 días. Durante la fase de diseño planificada se utilizó Altova UModel para elaborar el diseño inicial de la arquitectura. Posteriormente, la implementación de ésta lleva a cambios constantes en el diseño (más o menos importantes). Pero el diseño inicial no podía ser modificado tras expirar la licencia. Por eso, se obtuvo una nueva licencia para otro PC en la fase final de proyecto de elaboración de la memoria, para poder realizar el diseño final de la arquitectura asegurándose de que no habría nuevos cambios.

- **Cámara de fotos Canon:** Por último, aunque no se trate de una herramienta hardware o software como tal, se utilizó una cámara de fotos durante la fase de pruebas para documentar con mayor precisión y claridad los resultados obtenidos en el proyecto, ya que las características del mismo hacen que éstos no sean resultados clásicos como los que se pueden obtener con un proyecto más tradicional. Se tomaron fotografías y vídeos para incluir tanto en la memoria del proyecto, como en la presentación.

3.2. Requisitos de usuario

En este apartado se presentan las tablas de requisitos identificados en el análisis del problema a resolver. Para cada uno de ellos se adjunta un identificador, así como una breve descripción del requisito. A continuación se han marcado tres propiedades del requisito:

- **Necesidad:** Importancia de la implementación o no del requisito en cuestión. Los posibles valores son: esencial, deseable u opcional. Aquellos catalogados como esenciales son obligatorios.

- **Prioridad:** Indica el orden en que deben implementarse. Puede ser: alta, media o baja. Los requisitos de "alta" prioridad deben crearse antes que los de prioridad "media", y éstos, antes que los de prioridad "baja".

- **Estabilidad:** Aquellos requisitos con menor estabilidad, se consideran más susceptibles de modificaciones durante el periodo de desarrollo del proyecto.

Identificador: REQ-01	
Descripción	Creación de una arquitectura híbrida (deliberativa + reactiva) para el control de un robot Pioneer 3-DX.
Necesidad	√Esencial ☐Deseable ☐Opcional
Prioridad	√Alta ☐Media ☐Baja
Estabilidad	√Alta ☐Media ☐Baja

Tabla 2: Requisito REQ-01

Identificador: REQ-02	
Descripción	La arquitectura debe ser altamente modular y reutilizable.
Necesidad	√Esencial ☐Deseable ☐Opcional
Prioridad	√Alta ☐Media ☐Baja
Estabilidad	√Alta ☐Media ☐Baja

Tabla 3: Requisito REQ-02

Identificador: REQ-03	
Descripción	La arquitectura deberá basarse en el modelo de arquitectura de 3 capas (3-Tiered Architecture) de Bonasso y Kortenkamp [Bonasso, 1997].
Necesidad	☐Esencial ☐Deseable √Opcional
Prioridad	√Alta ☐Media ☐Baja
Estabilidad	☐Alta √Media ☐Baja

Tabla 4: Requisito REQ-03

Identificador: REQ-04	
Descripción	La capa de planificación operará de forma deliberativa a la hora de elaborar planes y tareas para alcanzar una meta.
Necesidad	√Esencial ☐Deseable ☐Opcional
Prioridad	√Alta ☐Media ☐Baja
Estabilidad	√Alta ☐Media ☐Baja

Tabla 5: Requisito REQ-04

Identificador: REQ-05	
Descripción	La capa de secuenciación operará de forma deliberativa y reactiva a la hora seleccionar conjuntos de habilidades para llevar a cabo las tareas.
Necesidad	√Esencial ☐Deseable ☐Opcional
Prioridad	√Alta ☐Media ☐Baja
Estabilidad	√Alta ☐Media ☐Baja

Tabla 6: Requisito REQ-05

Identificador: REQ-06	
Descripción	La capa de habilidades operará de forma reactiva a la hora de ejecutar las habilidades indicadas por el secuenciador.
Necesidad	√Esencial ☐Deseable ☐Opcional
Prioridad	√Alta ☐Media ☐Baja
Estabilidad	√Alta ☐Media ☐Baja

Tabla 7: Requisito REQ-06

Identificador: REQ-07	
Descripción	Los datos de entrada del sistema serán proporcionados por los sensores del robot: sonar y laser.
Necesidad	√Esencial ☐Deseable ☐Opcional
Prioridad	√Alta ☐Media ☐Baja
Estabilidad	☐Alta √Media ☐Baja

Tabla 8: Requisito REQ-07

Identificador: REQ-08	
Descripción	La arquitectura deberá proporcionar interfaces para su simulación en Webots y su ejecución en el robot mediante Player.
Necesidad	☐Esencial √Deseable ☐Opcional
Prioridad	√Alta ☐Media ☐Baja
Estabilidad	☐Alta √Media ☐Baja

Tabla 9: Requisito REQ-08

Identificador: REQ-09	
Descripción	Cada capa de la arquitectura se ejecutará en un hilo de ejecución distinto e independiente, pasándose entre ellos mensajes cuando lo necesiten.
Necesidad	☐Esencial ☐Deseable √Opcional
Prioridad	☐Alta √Media ☐Baja
Estabilidad	☐Alta √Media ☐Baja

Tabla 10: Requisito REQ-09

Identificador: REQ-10	
Descripción	El planificador tomará información del modelo del mundo para elaborar un plan que satisfaga la/s meta/s, lo descompondrá en tareas y se las pasará al secuenciador para su tratamiento.
Necesidad	√Esencial ☐Deseable ☐Opcional
Prioridad	√Alta ☐Media ☐Baja
Estabilidad	√Alta ☐Media ☐Baja

Tabla 11: Requisito REQ-10

Identificador: REQ-11	
Descripción	El secuenciador recibirá las tareas del planificador y seleccionará un conjunto de habilidades que permitan llevar a cabo la tarea.
Necesidad	√Esencial ☐Deseable ☐Opcional
Prioridad	√Alta ☐Media ☐Baja
Estabilidad	√Alta ☐Media ☐Baja

Tabla 12: Requisito REQ-11

Identificador: REQ-12	
Descripción	El secuenciador activará y desactivará habilidades del conjunto seleccionado previamente según se requiera modificar el comportamiento del robot.
Necesidad	√Esencial ☐Deseable ☐Opcional
Prioridad	√Alta ☐Media ☐Baja
Estabilidad	√Alta ☐Media ☐Baja

Tabla 13: Requisito REQ-12

Identificador: REQ-13	
Descripción	Cada habilidad ejecutará un comportamiento simple para el robot. Estas habilidades harán uso de los sensores y actuadores del robot.
Necesidad	√Esencial ☐Deseable ☐Opcional
Prioridad	√Alta ☐Media ☐Baja
Estabilidad	√Alta ☐Media ☐Baja

Tabla 14: Requisito REQ-13

Identificador: REQ-14	
Descripción	Las habilidades harán uso de eventos para comprobar el éxito o no a la hora de realizar una tarea.
Necesidad	☐Esencial √Deseable ☐Opcional
Prioridad	☐Alta √Media ☐Baja
Estabilidad	√Alta ☐Media ☐Baja

Tabla 15: Requisito REQ-14

Identificador: REQ-15			
Descripción	La arquitectura permitirá ejecutar las siguientes habilidades del robot: mantener una dirección, seguir una pared, evitar obstáculos e ir hacia un punto concreto.		
Necesidad	☐Esencial	√Deseable	☐Opcional
Prioridad	☐Alta	√Media	☐Baja
Estabilidad	☐Alta	☐Media	√Baja

Tabla 16: Requisito REQ-15

Identificador: REQ-16			
Descripción	El robot contará con un estado global en el sistema. Este estado será común para las 3 capas de la arquitectura.		
Necesidad	√Esencial	☐Deseable	☐Opcional
Prioridad	☐Alta	√Media	☐Baja
Estabilidad	√Alta	☐Media	☐Baja

Tabla 17: Requisito REQ-16

Identificador: REQ-17			
Descripción	La comunicación con el robot se realizará mediante el puerto serie del mismo.		
Necesidad	☐Esencial	√Deseable	☐Opcional
Prioridad	☐Alta	√Media	☐Baja
Estabilidad	☐Alta	☐Media	√Baja

Tabla 18: Requisito REQ-17

3.3. Diseño de la Arquitectura

Este apartado detalla el diseño realizado de la arquitectura 3T, con sus simplificaciones y características únicas. Se ha realizado un diseño en dos niveles de detalle: a bajo nivel (diseño arquitectónico), en el que se describe el sistema en función de sus subsistemas y componentes; y a alto nivel (diseño detallado), en el que el sistema es descrito mediante la descomposición de los componentes en clases, con sus atributos y métodos.

Adicionalmente a la descripción de ambos diseños con sus correspondientes diagramas, se adjuntan las tablas de componentes y clases a modo de referencia para el futuro uso de la arquitectura en desarrollo, y para consultar de manera detallada el

propósito de cada uno de los elementos de los diagramas del diseño arquitectónico y detallado.

3.3.1. Diseño arquitectónico

La arquitectura diseñada para el control automático del robot, se basa en la arquitectura 3T propuesta por Bonnaso y Kortenkamp, presentada en el capítulo 2.3.2.a. Sin embargo, ha sido necesario realizar algunas modificaciones para simplificarla y adaptarla a los objetivos concretos que se persiguen en este proyecto.

Tal y como se puede observar en la figura 37, se identifica perfectamente la arquitectura en tres capas, compuesta por el planificador, el secuenciador y el controlador. Cada una de estas capas únicamente se comunica con la capa inmediatamente superior o inferior (en el caso del secuenciador, con ambas) mediante el uso de una interfaz única, para así lograr la mayor independencia posible entre capas y la encapsulación de sus operaciones.

Además de las tres capas o subsistemas principales de la arquitectura 3T, se ha incluido un subsistema más llamado "*Infraestructura*". Este subsistema permitie el uso de la arquitectura para cualquier robot con características similares al Pioneer 3-DX, independientemente de la plataforma cliente que haga uso de ella. Como ejemplo, se presentan dos clientes independientes que pueden utilizar la arquitectura para el control automático del robot en un entorno simulado (Webots) o en un entorno real con el propio robot (ARIA).

Ahora se pasa a detallar cada subsistema de la arquitectura, con los componentes que lo forman y su propósito. En primer lugar se tiene la infraestructura, que forma el único componente de este elemento ajeno a la arquitectura 3T original (ver figura 38). Aunque se mostrará su contenido en el diseño detallado, cabe destacar que existe una clase principal, *Robot*, que representa el robot, como su propio nombre indica. Es esta clase la que proporciona las dos interfaces necesarias para que las diferentes aplicaciones clientes puedan utilizar la arquitectura, y para que la propia arquitectura en tres capas pueda acceder a la información del robot que permite llevar a cabo su control automático.

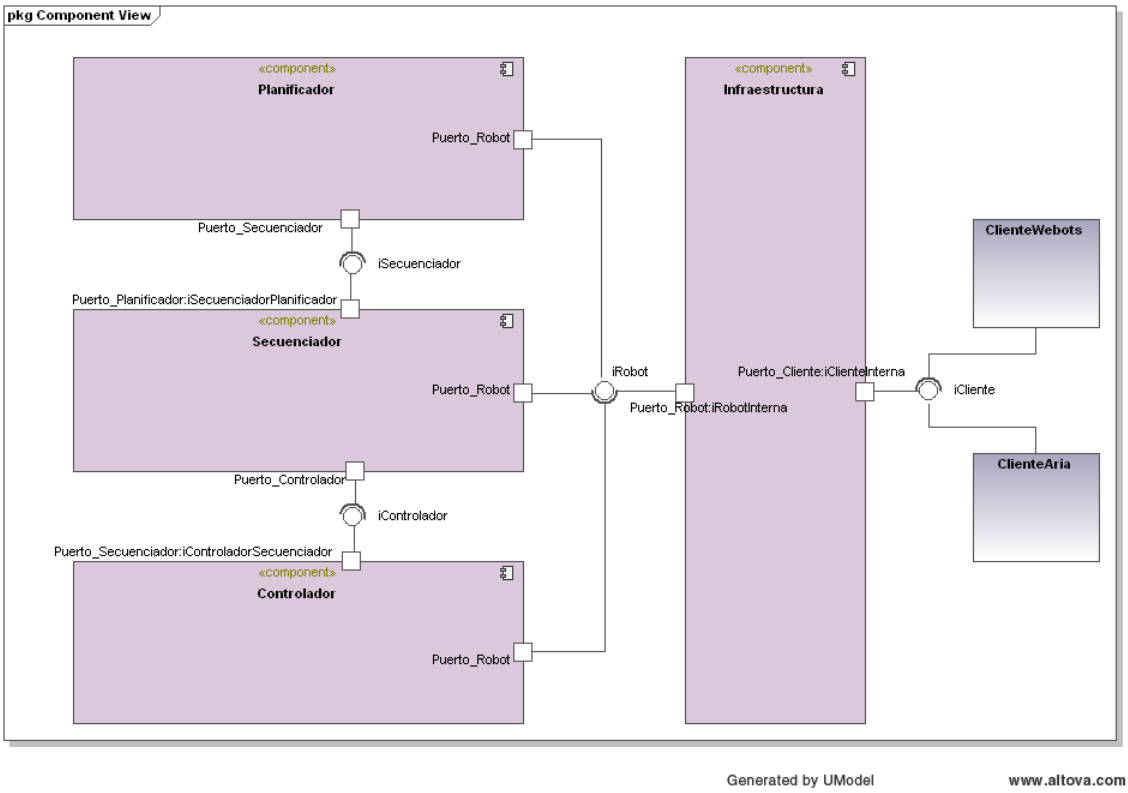


Figura 37: Diseño arquitectónico completo de la arquitectura 3T

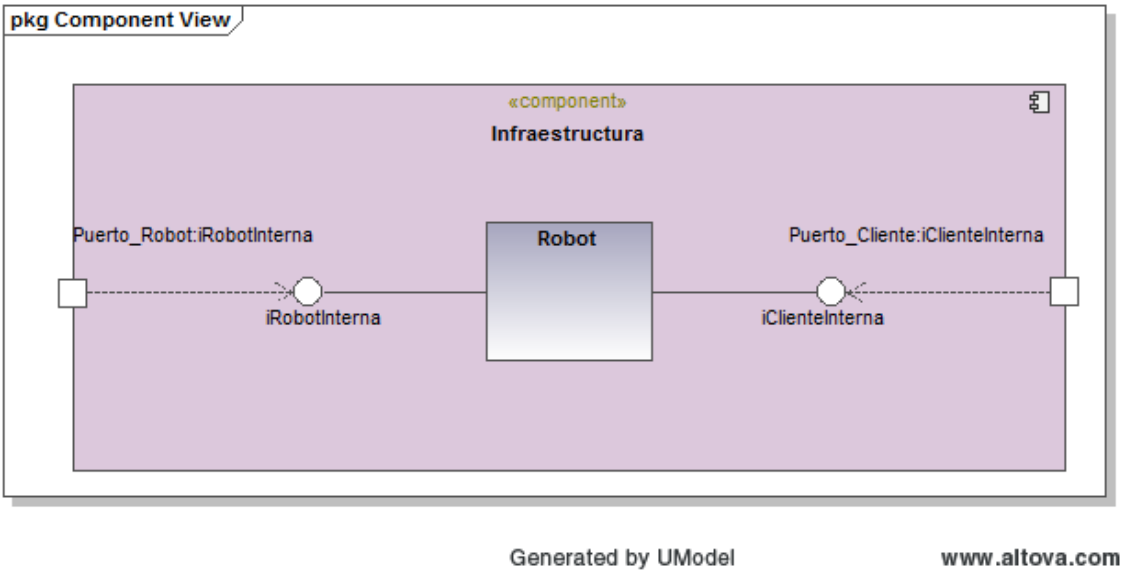


Figura 38: Subsistema Infraestructura

Se sigue con la explicación de la arquitectura de abajo a arriba. El siguiente subsistema es el controlador (también conocido en otros modelos de tres capas como "*skills*" o habilidades). Como se puede observar en la figura 39, el controlador está formado por tres componentes. El primero de ellos, el gestor de habilidades, es el encargado de proporcionar al *Secuenciador* las operaciones necesarias para trabajar con las habilidades del robot, activando o desactivando conjuntos de éstas según sea necesario. Además, este gestor está a cargo del hilo de ejecución de esta capa, gestionando sus comunicaciones y monitorizando su ejecución.

Por otro lado, se presentan dos componentes asociados, relacionados con las habilidades propiamente dichas del robot. En este proyecto se implementó el componente de navegación, y se dejó el componente de visión como ejemplo de cómo incluir nuevas habilidades. Es importante mencionar que la arquitectura es extensible, pudiendo introducir nuevos conjuntos de habilidades, como la visión o el mapeado. Para ello sólo es necesario incluir un nuevo componente similar al de navegación e implementar las diferentes habilidades que lo compongan. Estos componentes son capaces de comunicarse entre sí para dar lugar a comportamientos más complejos, como por ejemplo navegar por un mapa utilizando la visión.

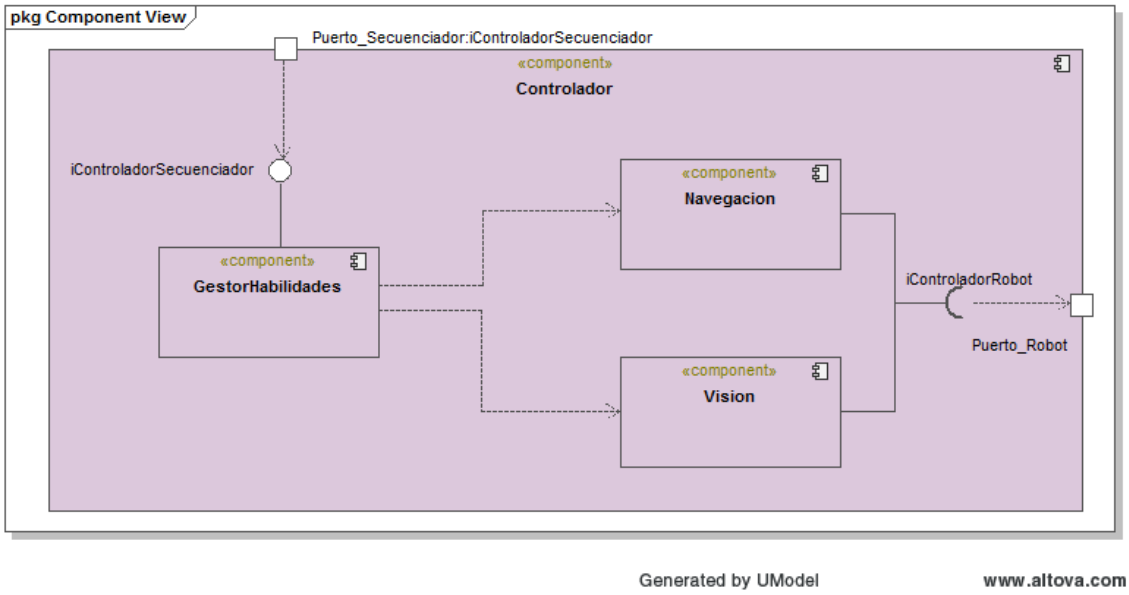


Figura 39: Subsistema Controlador

A continuación se muestra el subsistema *Secuenciador*. Está formado por dos componentes: *Interprete* y *Memoria*. El intérprete se encarga de interpretar los planes que le pasa el planificador y convertir las tareas y acciones en habilidades a ejecutar por el controlador. También gestiona las comunicaciones del hilo de ejecución de esta capa intermedia y se encarga de activar y desactivar conjuntos de habilidades, cuya secuenciación permitie al robot cumplir con las tareas designadas para alcanzar la meta. Por otro lado, se encuentra la memoria de la arquitectura, encargada de almacenar planes pasados y su resolución, así como el estado del robot en cada momento del plan que se esté ejecutando. Monitorizando esta información, es capaz de realizar las modificaciones necesarias en el plan y de ayudar al *Interprete* con la secuenciación de tareas. Este componente, por tanto, sirve de apoyo al *Interprete* para elaborar la secuencia de habilidades a ejecutar por el controlador.

Tal y como se puede observar en la figura 40, los dos componentes del subsistema se encargan de proporcionar al *Planificador* la interfaz que incluye las operaciones necesarias para la correcta comunicación entre estas dos capas. Asimismo, el *Interprete* es el componente que hace uso de la interfaz proporcionada por el controlador.

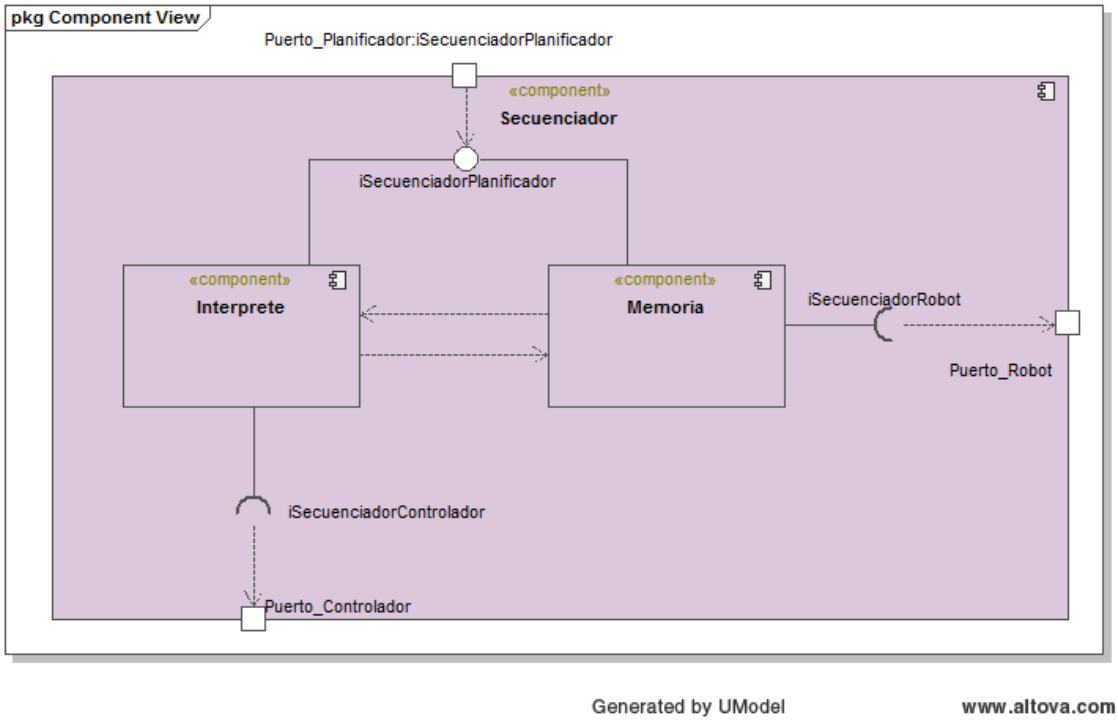


Figura 40: Subsistema Secuenciador

Por último, el subsistema *Planificador* está compuesto por dos componentes que trabajan en conjunto para elaborar los planes que harán que el robot alcance sus objetivos (ver figura 41). Por un lado se tiene un gestor de planes, que al igual que los componentes similares de las otras dos capas, se encarga de la monitorización y gestión de comunicaciones del hilo de ejecución del *Planificador*. Además, implementa un planificador genérico que utiliza un algoritmo de búsqueda para la creación de los planes a partir de las metas que persiga el robot y una serie de acciones posibles de llevar a cabo. Para poder realizar esta gestión, el componente debe poder acceder a la información del robot, mediante la interfaz proporcionada por el subsistema *Infraestructura*. También se comunica con el *Secuenciador* mediante la interfaz proporcionada por éste para poder enviarle los planes que debe tratar.

El otro componente de este subsistema es el modelo del mundo, que representa el entorno del robot. Este componente contiene el mapa del que hará uso el robot, así como toda la gestión de la posición de éste dentro del mapa. Esta representación del modelo del mundo ha sido diseñada de forma que sea extensible, como las habilidades del robot. Así, por ejemplo, se podrá añadir al modelo del mundo el estado de la batería del robot, si es un elemento más del modelo del mundo en el problema concreto a resolver, o incluso permitir el uso de diferentes tipos de mapas y localización.

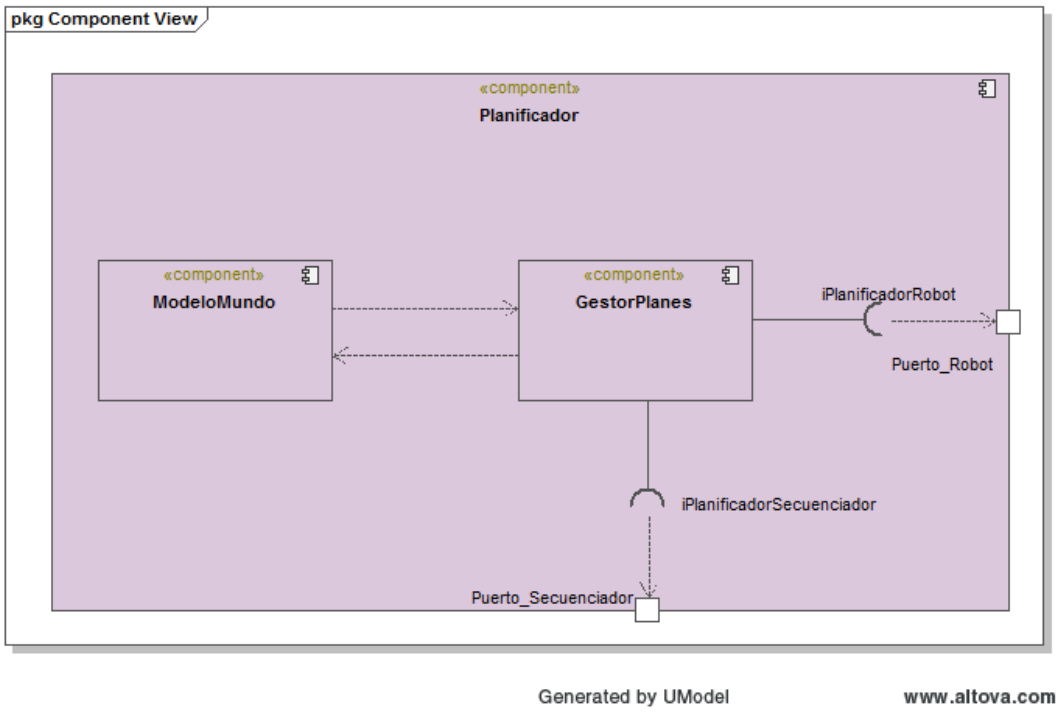


Figura 41: Subsistema Planificador

3.3.2. Diseño detallado

Tomando como referencia el diseño arquitectónico, visto en el apartado anterior, en este apartado se presenta el diseño detallado de la arquitectura 3T. Para cada uno de los subsistemas, se mostrarán dos diagramas. El primero de ellos contiene las clases de cada componente del subsistema, y se utiliza para exponer de forma clara el funcionamiento general del subsistema completo, como un todo. Estos diagramas no contienen un alto nivel de detalle; sólo muestran aquellos atributos u operaciones relevantes que ayuden a entenderlos. Una vez expuesto, el segundo diagrama muestra el diseño detallado, con todos los atributos y métodos relevantes de las clases de cada uno de los subsistemas. Cabe destacar que no todos los métodos están presentes en los diagramas. Aquellas clases más grandes y que recogen más funcionalidad muestran sus métodos más importantes, obviando los métodos de acceso (*getters* y *setters*), que se presupone forman parte de dichas clases.

En primer lugar se presenta el subsistema *Infraestructura*. Tal y como se puede observar en la figura 42, el elemento central de este subsistema es el robot y todos los elementos que lo componen. La clase *Robot* representa el propio robot físico que hace uso de la arquitectura implementada. Por ello, posee una serie de atributos tales como el estado, los sensores, los actuadores o la batería. Como se observa, la clase *Bateria* es una clase abstracta, que no fue implementada, y que sirve de ejemplo sobre cómo incluir nuevos elementos para el robot, en caso de ser necesario. Lo mismo ocurre con los sensores o los actuadores. Por otro lado, la clase *Robot* proporciona algunas operaciones que son utilizadas por las diferentes capas de la arquitectura para acceder a la información del robot, mientras que otras permiten el uso de la arquitectura independientemente de la aplicación cliente que haga uso de ella, haciendo de "*parser*" entre diferentes lenguajes de programación para simuladores o el propio robot físico.

Entrando en detalle en el diagrama de clases, la clase *Robot* permite acceder al resto de la arquitectura a su estado y componentes, ofreciendo los métodos de acceso correspondientes (ver figura 43). Esta es la función principal de este subsistema, aparte de servir como único punto de entrada a las diferentes aplicaciones clientes para hacer de "*parser*", como ya se ha comentado. De esta forma, lo más destacable es el método *inicializar()* de la clase *Robot*, que es el encargado de transformar la inicialización del

robot dada por la aplicación cliente, adaptándola a esta arquitectura. Por otro lado, los diferentes elementos que componen el robot, como los sensores o los actuadores, son identificados por una cadena de caracteres única, que permite al resto del sistema acceder a ellos y ejecutar las operaciones necesarias, como por ejemplo establecer una cierta velocidad a las ruedas u obtener la información captada por los sónar. Cabe destacar el atributo de tipo *Arquitectura* del robot, que se encarga de la creación e inicialización única de los tres hilos de ejecución de la arquitectura diseñada.

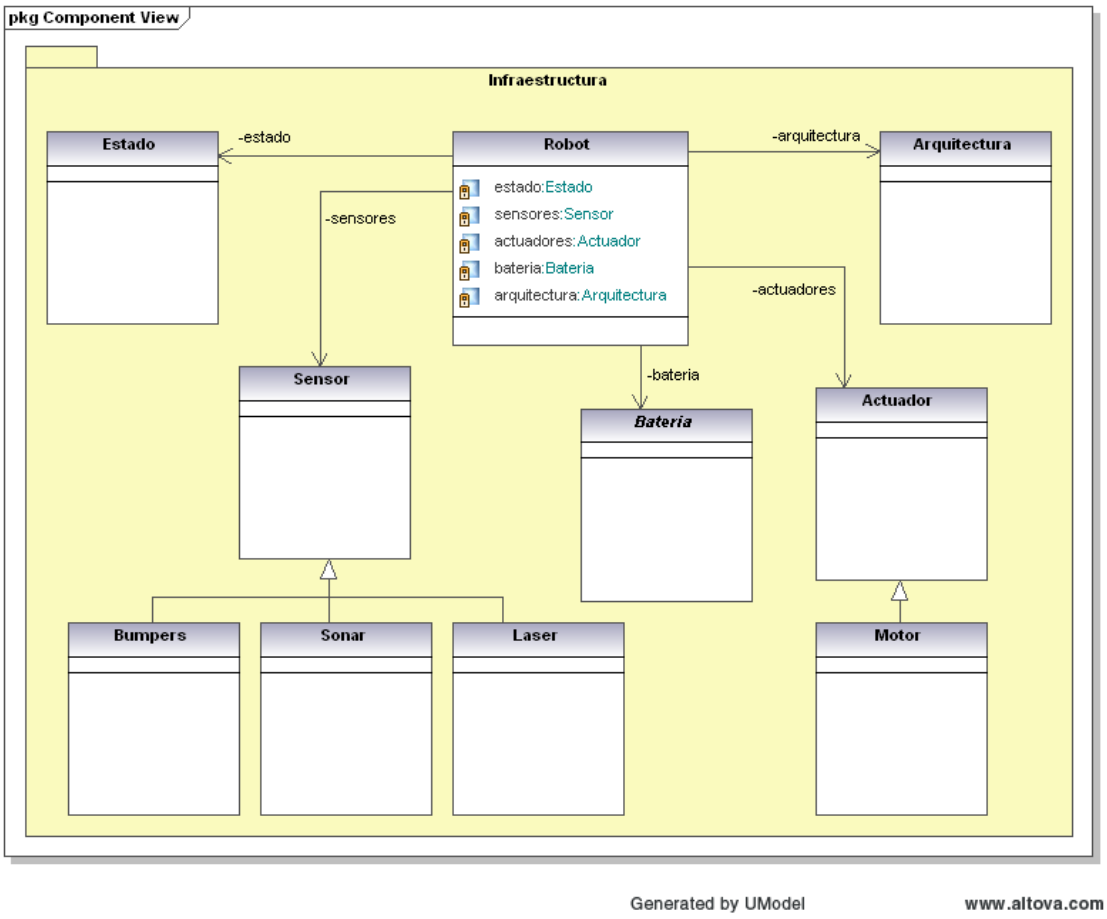


Figura 42: Clases del subsistema Infraestructura

El siguiente subsistema a tratar es el *Controlador* (ver figura 44). Este subsistema es la capa reactiva de la arquitectura, la que trabajaba con las habilidades del robot, es decir, las acciones de bajo nivel que podía llevar a cabo el robot para cumplir con las tareas indicadas desde capas superiores.

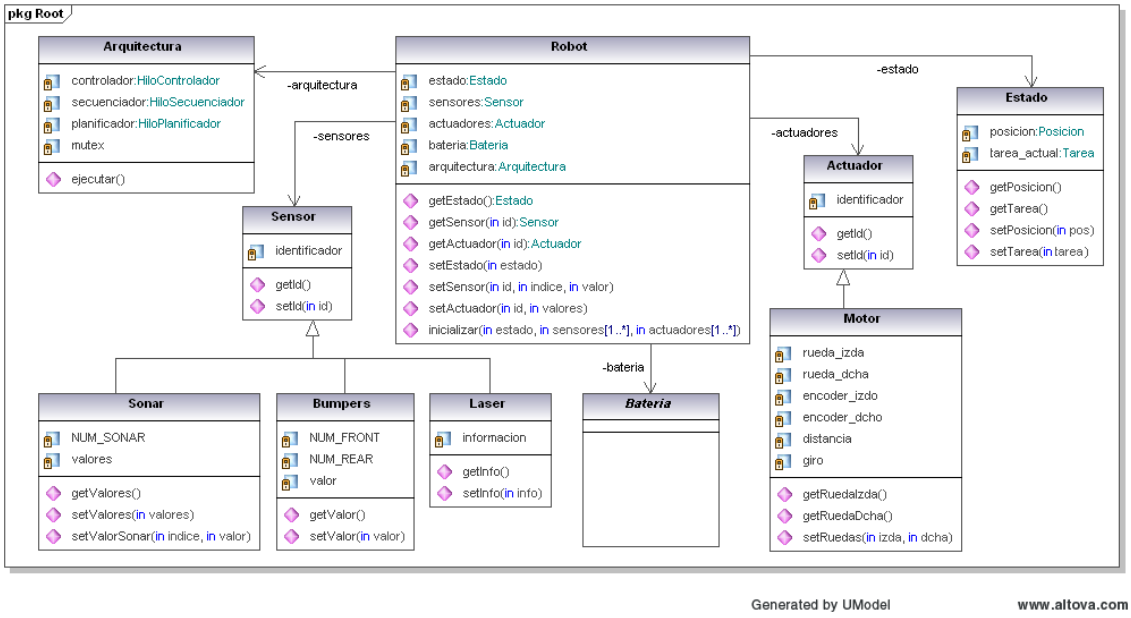


Figura 43: Diseño detallado del subsistema Infraestructura

El *Controlador* está formado por un componente principal, *GestorHabilidades*, y por tantos componentes como tipos de habilidades pueda llevar a cabo el robot. En este caso, se presenta el componente *Navegacion* y se aporta el componente *Vision* como ejemplo de extensibilidad de la capa reactiva. El gestor de habilidades contiene la clase *HiloControlador*, que es el hilo de ejecución de esta capa. Esta clase proporciona las operaciones necesarias para que el *Secuenciador* (capa inmediatamente superior) haga uso de ellas. Este hilo de ejecución tiene varias instancias de la clase *Habilidades*, que a su vez es un agregado de *Habilidad*. Una de estas instancias es la lista de todas las habilidades posibles que tiene el robot, y que por tanto pueden ejecutarse para cumplir las tareas indicadas. De la clase *Habilidad* heredan todas las clases que forman el resto de componentes del subsistema. Cada clase de los componentes *Navegacion*, *Vision*, etc. son habilidades concretas, y todas ellas hacen uso de las operaciones de acceso al robot, proporcionadas por la *Infraestructura*, para poder acceder a la información de los sensores y mandar órdenes a los actuadores del robot. Además, existen dos habilidades especiales, situadas en el componente *GestorHabilidades*, que si bien no son habilidades del robot propiamente dichas, sirven para gestionar la ejecución del robot en determinadas circunstancias especiales. Estas habilidades son *Esperar* y *Finalizar*.

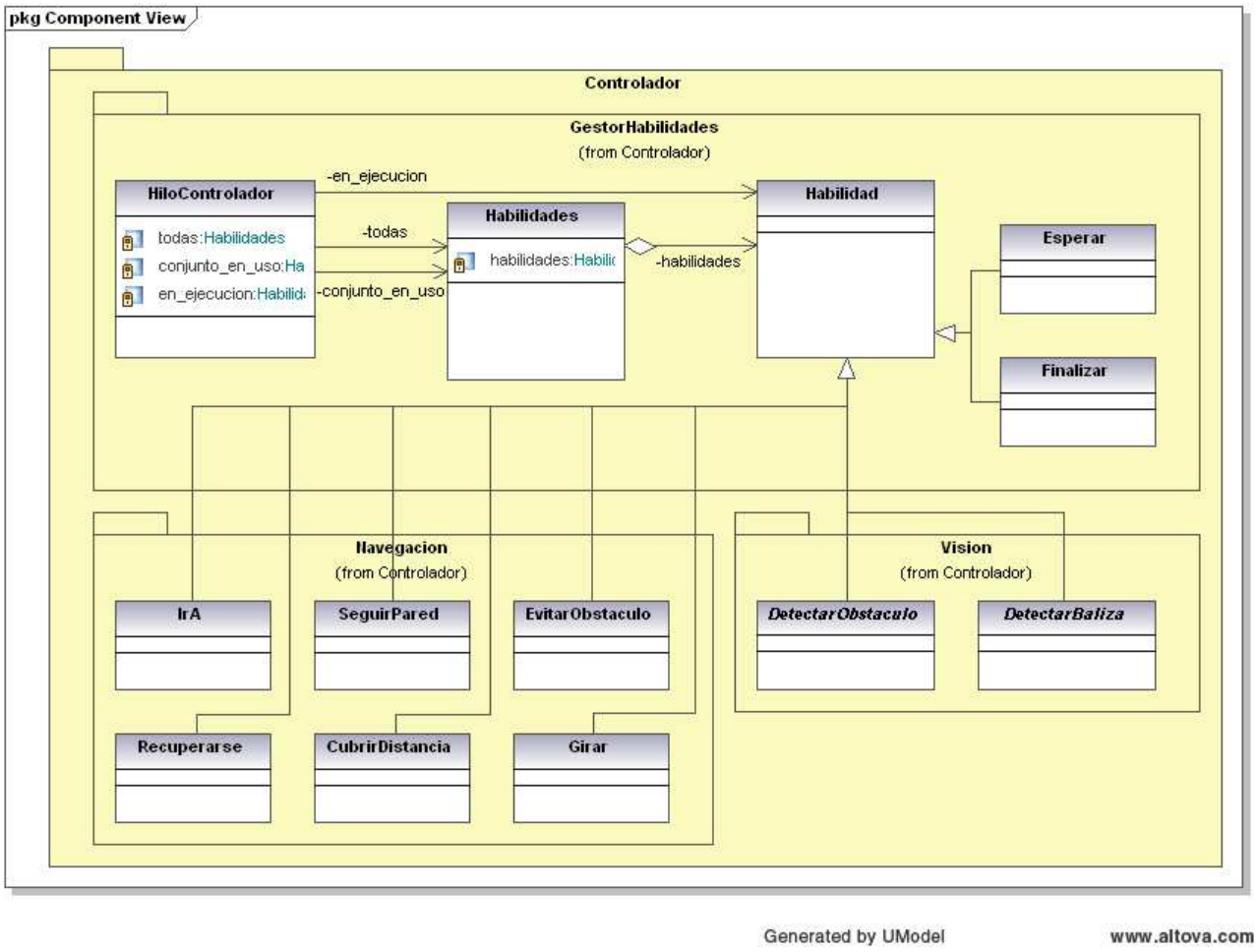


Figura 44: Clases del subsistema Controlador

Como se observa en la figura 45, el *HiloControlador* contiene la mayor funcionalidad de esta capa. Además de tener la lista con todas las habilidades del robot, tiene una tarea a llevar a cabo en cada momento, así como un atributo que indica si se ha alcanzado el éxito para dicha tarea. También posee como atributo un flujo de comunicación con el hilo de ejecución del *Secuenciador*. Las operaciones de esta clase, como es obvio, están relacionadas con estos atributos, de forma que el hilo pueda cambiar la tarea a ejecutar cuando el *Secuenciador* le pase una nueva y activar o desactivar conjuntos de habilidades en consecuencia. Por supuesto, también es capaz de leer mensajes en el flujo de comunicación. Por su parte, las habilidades, que tienen métodos específicos dependiendo de la función que realicen, heredan de la superclase los métodos *ejecutar()* y *comprobarExito()*, que son reescritos por éstas, ya que cada habilidad tendrá que ejecutar diferentes operaciones, y el éxito o no se comprueba de

distinta manera. Asimismo, los algoritmos de las habilidades pueden diferir en función del entorno donde se ejecuten (simulador o entorno real), por lo que la mayoría de las habilidades contienen métodos de ejecución y comprobación de éxito diferenciados, los cuales se ejecutan o no dependiendo del valor del atributo "*sim*" de la *Habilidad*. La figura 45, por simplicidad, no muestra todas clases de las habilidades implementadas, pero sí todos aquellos métodos relevantes de algunas de ellas.

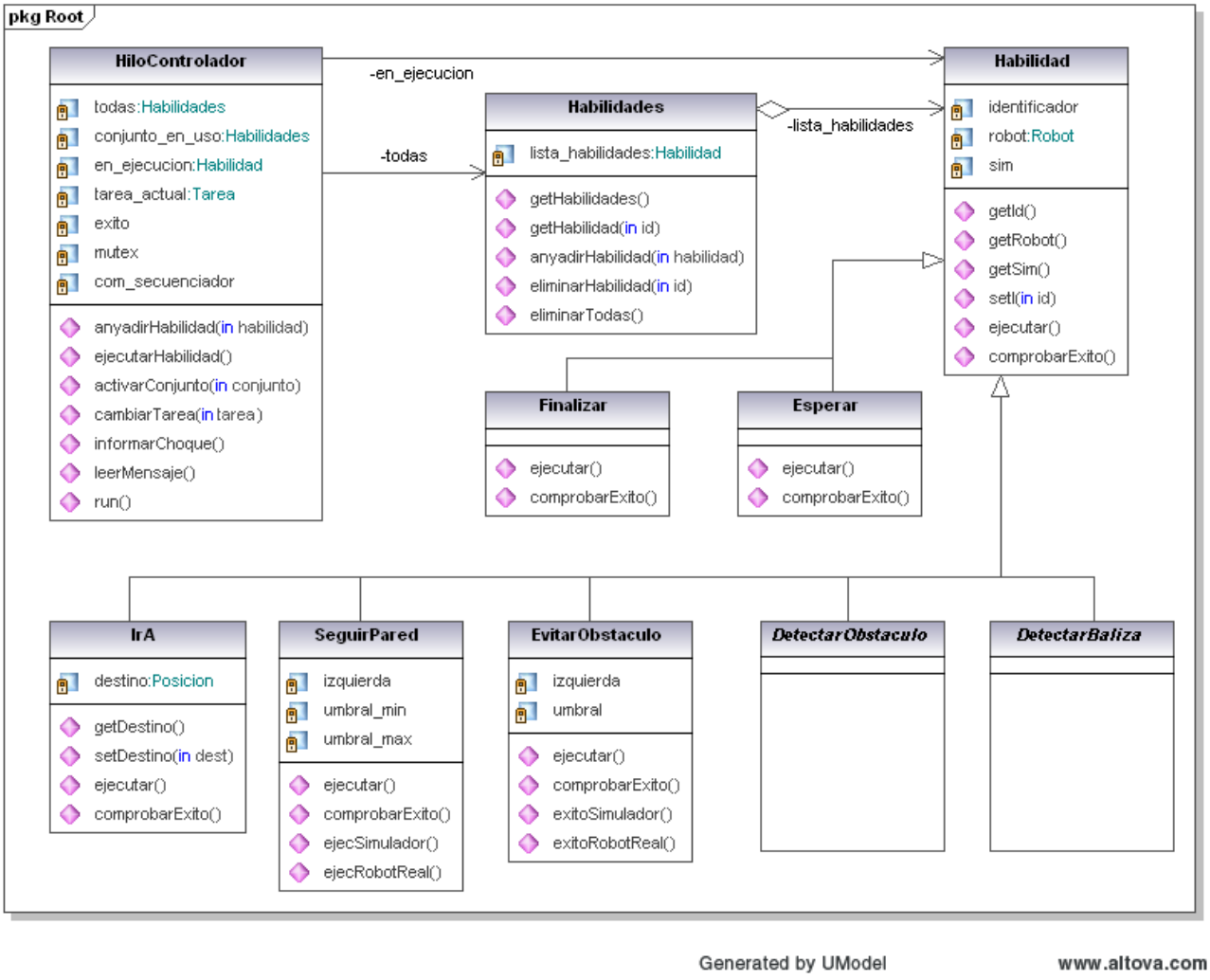


Figura 45: Diseño detallado del subsistema Controlador

El subsistema *Secuenciador* es la capa intermedia de la arquitectura, que trabaja tanto a nivel reactivo como a nivel deliberativo (ver figura 46). Entre sus componentes, en primer lugar tenemos el *Interprete*. En este componente se encuentra la clase *HiloSecuenciador*, cuyo propósito es similar a la de la clase *HiloControlador* de la capa inferior. El *HiloSecuenciador* hace uso de una agenda de tareas (clase *Agenda*) para almacenar los planes de la capa superior como elementos más utilizables (*Tareas*) en

esta capa, previa secuenciación de éstos. Este componente también contiene la clase *Secuenciacion*, encargada de realizar las operaciones necesarias para secuenciar correctamente las tareas y asignarles conjuntos de habilidades para llevarlas a cabo y de proporcionar estas tareas al controlador mediante la interfaz correspondiente. El hilo de ejecución implementa operaciones de las que hace uso la capa superior, el *Planificador*. También aporta operaciones la clase *Monitor* (parte del otro componente del *Secuenciador*, la *Memoria*). Como su propio nombre indica, esta clase monitoriza la ejecución general del sistema y el estado del robot para informar de los cambios a la clase *HiloSecuenciador*, y así replanificar la agenda si es necesario. La memoria se completa con el historial de planes pasados, que facilitan la secuenciación de tareas, y la librería de RAP. Los RAP implementados son simplificaciones de los paquetes de acciones reactivas desarrollados por Firby para la implementación de la arquitectura 3T [Firby, 1989]. La librería de RAP contiene todos los RAP definidos para la arquitectura y las diferentes tareas que puede llevar a cabo el robot. Se explicará más en profundidad este concepto en el capítulo 3.5.2.

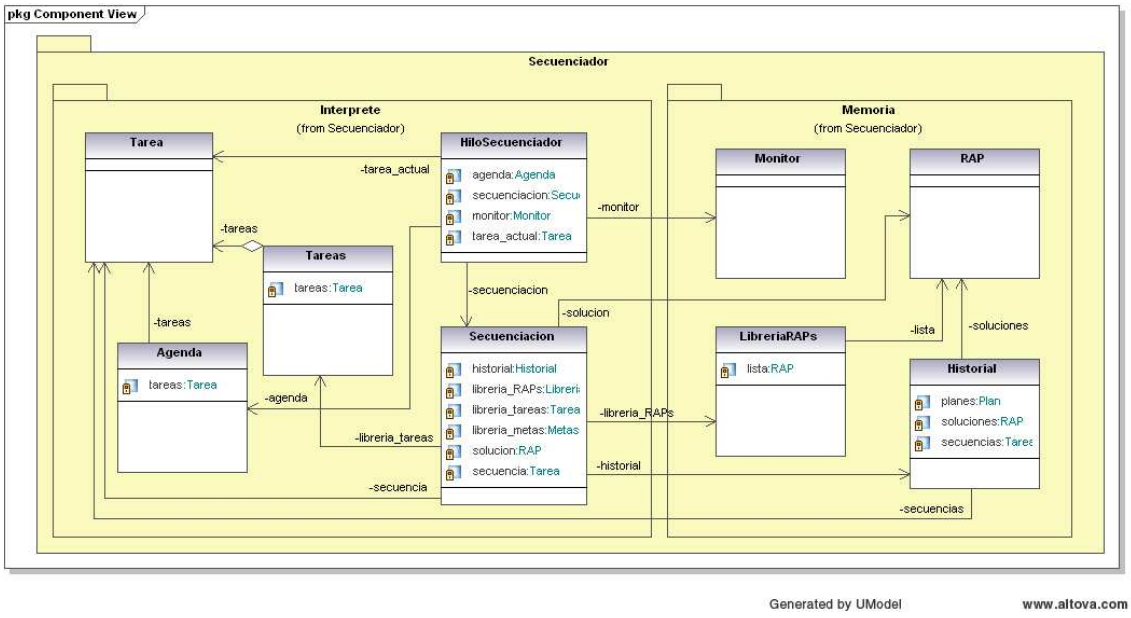


Figura 46: Clases del subsistema Secuenciador

Pasando al diagrama detallado del *Secuenciador*, mostrado en la figura 47, se puede observar que la estructura de la clase *HiloSecuenciador* es muy parecida a la del hilo de ejecución del controlador. Esta clase contiene como atributos los elementos que necesita para operar de forma eficiente (la agenda, el monitor y la instancia de la clase

Secuenciacion, entre otros), así como los dos flujos de comunicación con la capa superior y la inferior. Este hilo de ejecución puede modificar la agenda cuando sea necesario para realizar una pequeña replanificación del plan inicial sin necesidad de subir a la capa deliberativa, que opera de forma más lenta. Esto es posible gracias a que el plan contenido en la agenda no se almacena tal y como lo genera el planificador, sino que se utiliza la estructura de las tareas para hacerlo más manejable por el *Secuenciador*. Esto permite modificarlo y replanificar en esta capa intermedia. Por su parte, el monitor contiene una copia del estado del robot y su posición, aunque en un futuro se podrán incluir más elementos, para mantener actualizado en todo momento el estado global del modelo del mundo. El hilo puede consultar esta información y modificar la agenda en consecuencia.

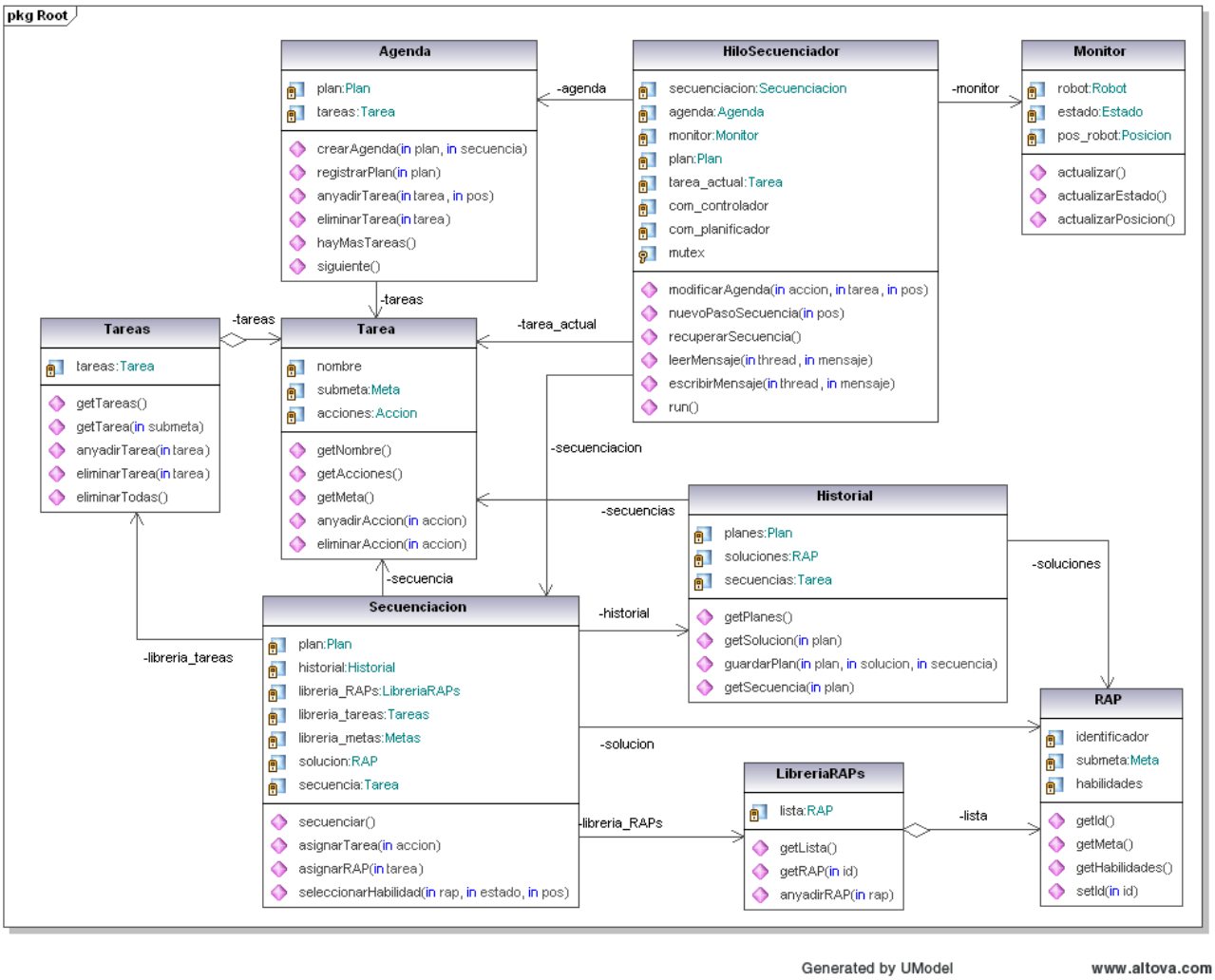


Figura 47: Diseño detallado del subsistema Secuenciador

En cuanto al conjunto de clases que generan la secuencia de habilidades, la clase *Secuenciacion* tiene un método *secuenciar()*, con el que se lleva a cabo esta secuenciación, asignando un RAP a una tarea concreta, y seleccionando de este RAP el conjunto de habilidades o habilidad única que van a permitir al robot llevar a cabo dicha tarea. Si es necesario, se pueden añadir RAP al sistema para poder alcanzar nuevas metas de nuevas tareas. Por último, el *Historial* permite obtener planes antiguos y sus correspondientes soluciones en forma de secuencias de tareas y RAP utilizados, lo que facilita al *Secuenciador* generar las secuencias del plan actual. También es posible guardar el plan actual una vez finalizado para consultarlo en posteriores ejecuciones.

Por último se presenta la capa deliberativa de la arquitectura: el subsistema *Planificador*. Éste se divide en dos componentes: el gestor de planes y el modelo del mundo. Este segundo componente, en principio, contiene aquellos elementos que representan el modelo del mundo para los problemas acotados que se resolvieron en este proyecto. Es por ello que contiene una clase *Mapa*, de la que pueden heredar tantas clases como tipos de mapas se deseen utilizar. Como ejemplo, se aportan mapas de coordenadas, compuesto de posiciones, y mapas de movimientos, que son grafos cuyos nodos representan localizaciones y cuyos arcos representan conexiones entre éstas. Este componente, como ya se explicó, puede extenderse con nuevos elementos según sea necesario para tener una representación completa del modelo del mundo.

En cuanto al gestor de planes, se puede observar en el siguiente diagrama, mostrado en la figura 48, que el elemento central del componente es, de nuevo, la clase *HiloPlanificador*, que representa el hilo de ejecución de esta capa. Este hilo contiene el planificador, que no ha sido implementado, y que puede ser sustituido por un elemento externo a la arquitectura de forma sencilla. Las clases que forman el planificador son *Planificacion*, *Predicado*, *Accion* y *Elemento*. El propósito de la clase *Planificacion* es similar al de la clase *Secuenciacion* en el *Secuenciador*. Esta clase se encarga de elaborar un plan a partir de unas metas y acciones definidas en la clase *Plan*. Para ello hace uso de los predicados, acciones y los elementos que los componen, tal y como hace un planificador clásico. Por otro lado, tenemos la clase *Plan*, compuesta por metas y acciones. Las metas no son más que predicados concretos que deben cumplirse en la ejecución del sistema. Las acciones, por su parte, pueden asociarse posteriormente a tareas por medio las submetas, lo que permite seleccionar RAP y habilidades para

llevarlas a cabo en la ejecución real. Mencionar que el hilo de ejecución del planificador hace uso de las operaciones proporcionadas tanto por el subsistema *Secuenciador*, como por la clase *Robot* del subsistema *Infraestructura*.

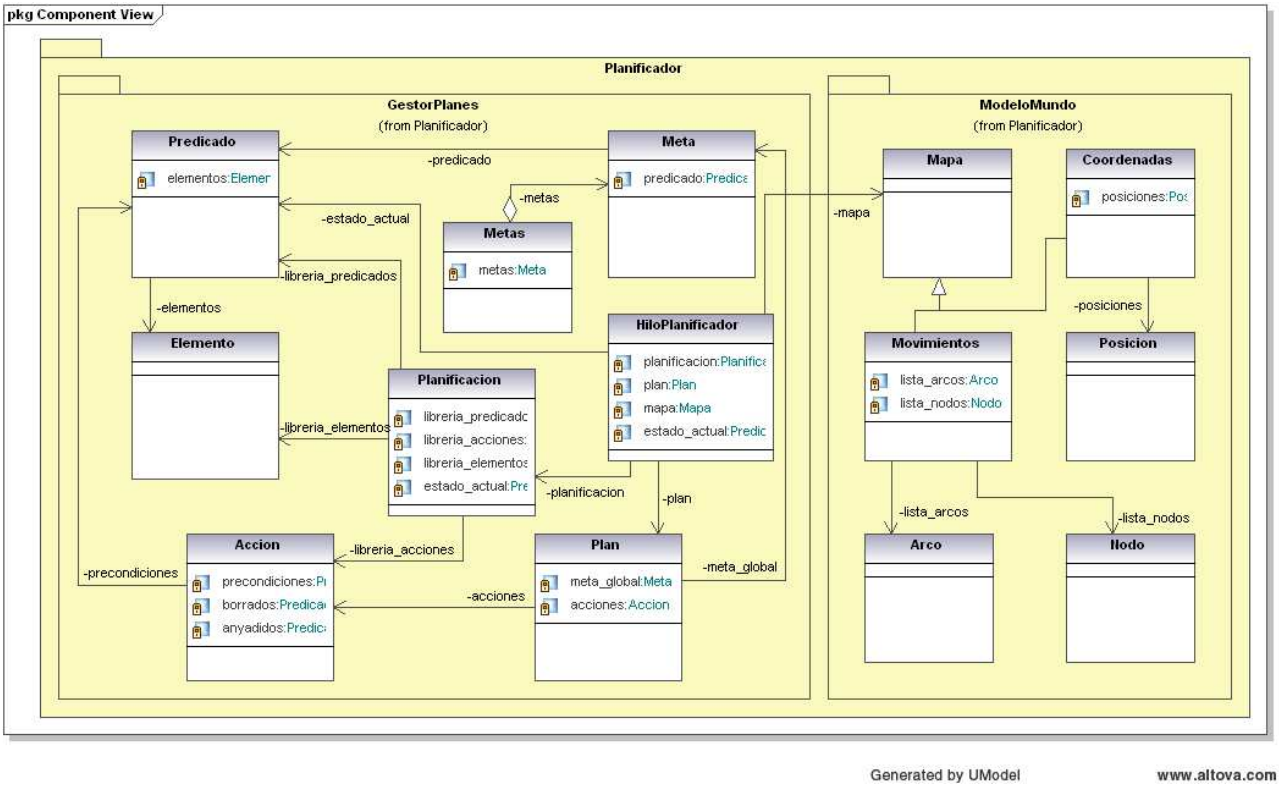


Figura 48: Clases del subsistema Planificador

Para terminar con el diseño detallado, comentar de nuevo que la clase *HiloPlanificador* funciona igual que las correspondientes de los otros dos subsistemas: contiene el plan a generar o ya generado, la instancia de la clase *Planificacion*, el mapa del que hace uso y el flujo de comunicación con el hilo de ejecución del *Secuenciador*, y las operaciones correspondientes (ver figura 49). La clase *Planificacion* contiene el método *planificar()*, que ejecuta el algoritmo de búsqueda sobre la colección de acciones y predicados para elaborar el plan objetivo. También es posible replanificar más adelante, pero esta acción se lleva a cabo en el *Secuenciador* si es posible, como se ha visto anteriormente. Por último, la mayor parte de clases de este subsistema, como *Meta*, *Elemento* o *Mapa*, por dar algún ejemplo, sirven como apoyo a la capa inmediatamente inferior para llevar a cabo las operaciones necesarias de secuenciación y así ejecutar las habilidades correctas que permitan al robot alcanzar su meta, de forma que este tipo de clases se limitan a ofrecer los métodos de acceso, añadir nuevos elementos a las listas que tienen como atributos o implementar métodos como *esIgual()*,

comprobación muy útil a la hora de secuenciar las tareas. Y es que es este tipo de información la que se utiliza en capas inferiores para llevar a cabo operaciones específicas. Así pues, otros métodos no son necesarios.

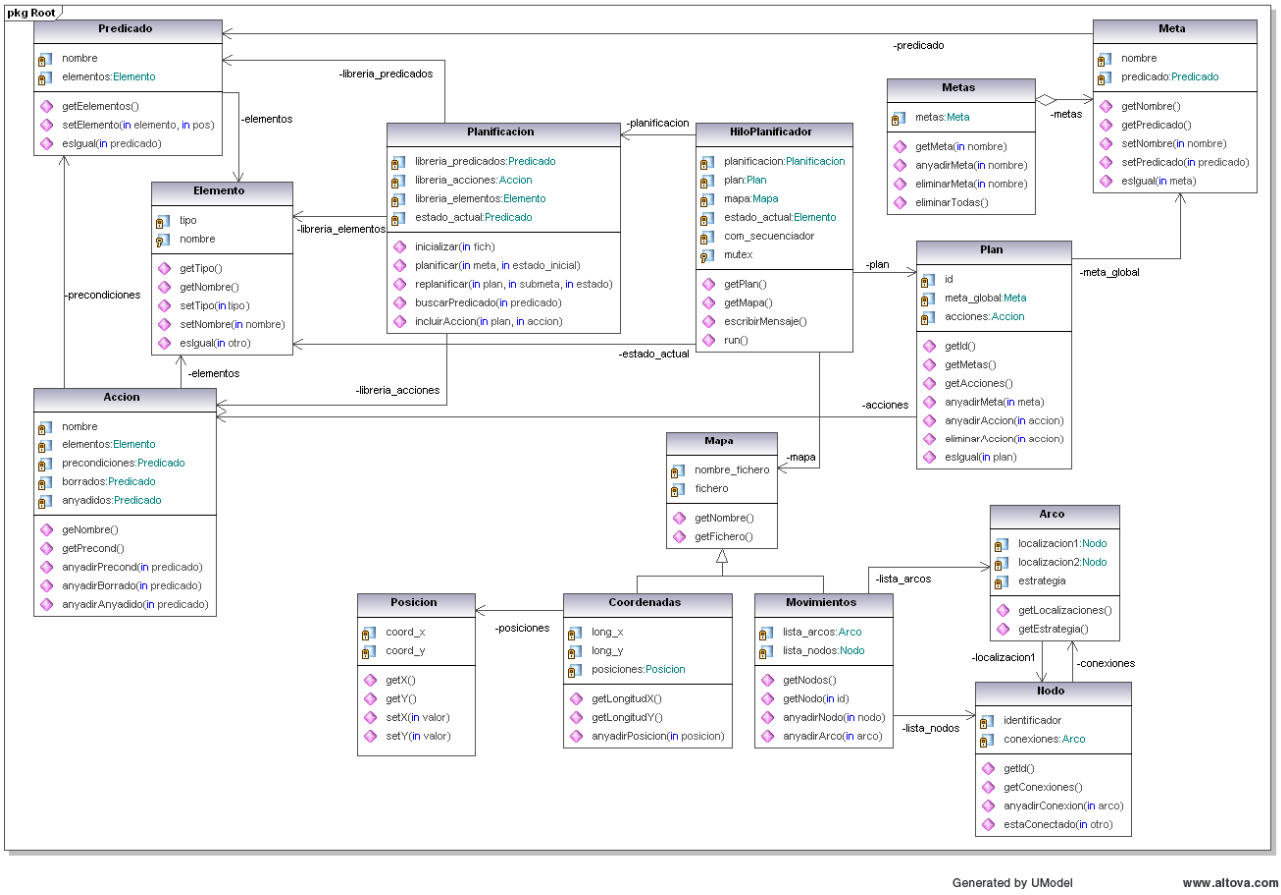


Figura 49: Diseño detallado del subsistema Planificador

3.4. Funcionamiento de la Arquitectura

A continuación se detallan las operaciones a las que se dedican los diversos componentes definidos en el diseño de la arquitectura y cómo se comunican entre sí para obtener el funcionamiento deseado de ésta. Primero se explica el funcionamiento general de la arquitectura completa, resultado de una posible ejecución genérica en un robot. Por otra parte, en el apartado 3.4.2. se detallan los comportamientos inteligentes de los que se ha dotado al robot, es decir, las habilidades concretas implementadas que pueda desempeñar el robot y de esta forma probar el funcionamiento de la arquitectura.

3.4.1. Funcionalidad general

En el apartado anterior se ha descrito de forma exhaustiva el diseño de la arquitectura desarrollada en este proyecto, dejando ver algunas de las características del funcionamiento de ésta. Ahora se va a tratar más en detalle cómo funciona la arquitectura en conjunto, desde la conexión del robot real o simulado, pasando por las operaciones y funcionalidades principales de las tres capas que componen la arquitectura. Además, se expone el comportamiento de cada una de las habilidades de navegación básica implementadas, comentando qué tareas puede llevar a cabo el robot y cómo lo hace. Por otro lado, este apartado no pretende ser un manual de uso de la arquitectura, por lo que ciertos detalles se deberán consultar en el Manual de Usuario, incluido en el presente documento (ver anexo A).

Aunque la arquitectura se ha dividido en tres capas, junto con un subsistema de infraestructura, y se ha diseñado de forma que sean lo más independientes posible unas capas de otras, es difícil explicar cómo funciona cada una de ellas sin verlas como un todo, por lo que los siguientes párrafos no se limitan a dividir el funcionamiento exclusivo de cada una de las partes de la arquitectura, sino que dan una visión general de cómo funciona el sistema en su totalidad.

Para empezar, la arquitectura se ha pensado como código de apoyo a una aplicación cliente, que hará uso de la arquitectura para controlar un robot de forma automática. Esta aplicación cliente puede estar programada para funcionar en diversas plataformas software o hardware, pero para poder entenderse con la arquitectura, debe hacer uso de ciertas operaciones y funcionalidades que proporciona el subsistema de infraestructura de la arquitectura.

La aplicación cliente, por tanto, es el proceso principal en ejecución, encargado de comunicarse de forma directa con el robot a través de las librerías que proporcione el simulador o el hardware robótico, y con la arquitectura a través de las operaciones que ésta ofrece. El elemento básico de esta segunda comunicación, que es la que interesa en este momento, es la instancia única de la clase Robot, que representa el robot simulado o real, con todos sus elementos. La aplicación cliente debe definir de qué elementos consta el robot a ser controlado: sensores, actuadores, batería, etc. Al crearse la

instancia del robot, el subsistema de infraestructura se encarga de inicializar los elementos necesarios que componen el subsistema.

No sólo es necesario que la aplicación cliente defina las características del robot, sino también el entorno en el que va a operar, en caso de que se haga uso de la capa superior de la arquitectura, el planificador. Se recuerda que esta capa no fue implementada, pero que la arquitectura proporciona las herramientas necesarias para hacer uso de ella. Una alternativa para definir el entorno del robot es utilizar ficheros de configuración para que el subsistema planificador elabore su propia representación del modelo del mundo (mapas) y el dominio del planificador (elementos, predicados y acciones). Una vez todos estos elementos son definidos e inicializados, el bucle de ejecución puede comenzar.

La figura 50 muestra un ejemplo sencillo de la ejecución típica de las diferentes capas y elementos de la arquitectura. A continuación se explica con todo detalle el diagrama y cómo se lleva a cabo una ejecución típica.

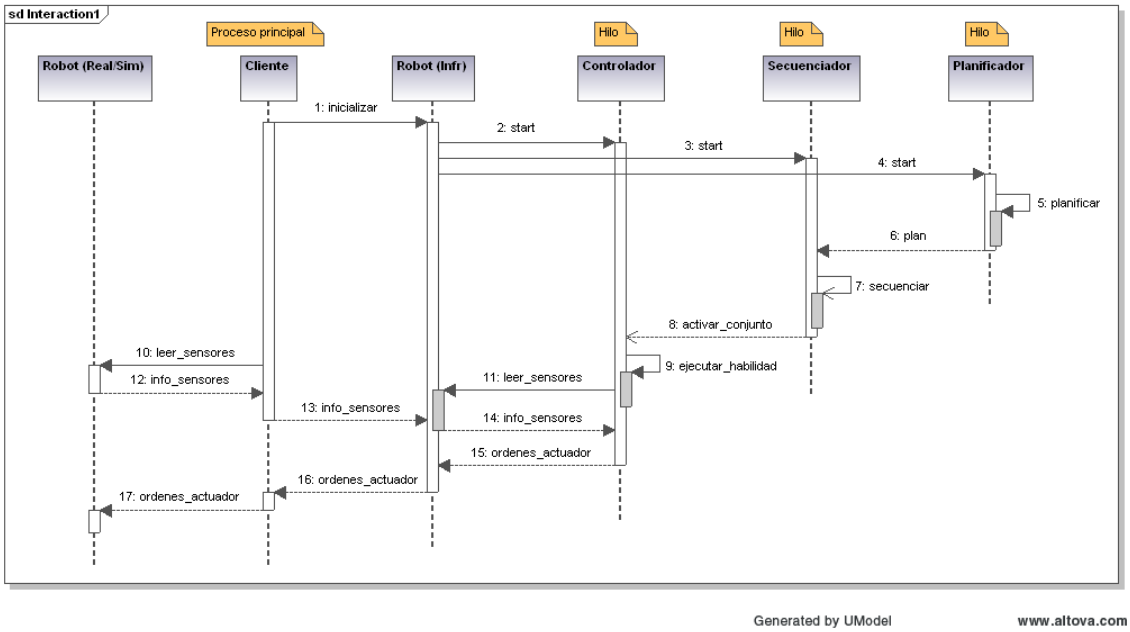


Figura 50: Diagrama de secuencia de una ejecución genérica

El bucle de ejecución principal comienza tras la inicialización comentada anteriormente. Una vez se ha entrado en este bucle, existen cuatro procesos en ejecución: un proceso principal, que es la aplicación cliente, y tres hilos de ejecución, correspondientes a cada una de las capas de la arquitectura.

Aunque el bucle de ejecución que se va a explicar a continuación es siempre el mismo y se repite hasta que el robot terminar de llevar a cabo el plan establecido, algunos hilos necesitan entrar en espera hasta que se realizan otras operaciones previas a la ejecución normal del robot. El primer paso lo da el *Planificador*, elaborando el plan a partir de una meta global introducida junto al estado inicial del dominio mediante un fichero de configuración. Mientras tanto, el proceso principal toma información del entorno del robot, aunque no es utilizada por el momento, de forma que no manda órdenes a los actuadores del robot (en realidad manda la orden de no mover las ruedas, asignando una velocidad de 0 a cada una de ellas). Por su parte, el *Controlador* está esperando a recibir la primera tarea que tiene que llevar a cabo; esto lo hace ejecutando una habilidad especial de gestión llamada "*Esperar*". El *Secuenciador* también debe esperar a que el *Planificador* elabore el plan.

Una vez se ha llevado a cabo la actividad deliberativa principal, el *Planificador* envía el plan al *Secuenciador*, y éste se pone a trabajar en la secuencia de tareas. Ahora es el *Planificador* es que se mantiene en espera (y se queda así durante toda la ejecución si no es necesario replanificar a nivel deliberativo). El proceso principal y el controlador también esperan, como venían haciendo hasta ahora.

La secuenciación de tareas se lleva a cabo de la siguiente manera: el *Secuenciador* tiene las librerías de tareas, metas y RAP. Cada acción descrita en el plan tiene asociada una tarea, y cada tarea se identifica por cumplir un objetivo o meta. De forma que el *Secuenciador* elabora una lista ordenada de tareas en función de las acciones del plan recibido. Para llevar a cabo cada una de las tareas debe ejecutarse una habilidad o conjunto de habilidades. Para poder asignar habilidades a las tareas es donde entra en juego el elemento conocido como RAP (Reactive Action Package).

Se ha definido e implementado una simplificación del concepto de RAP desarrollado por Firby para la implementación de la arquitectura 3T [Firby, 1989]. En la arquitectura implementada en este proyecto, un RAP se identifica, además de por una cadena de caracteres única, por una meta. Para cada meta definida en el sistema existe un RAP, que contiene un conjunto de habilidades con las que es posible alcanzar dicha meta si se ejecuta una o varias de esas habilidades. Por simplificación, y aunque los

RAP incluidos en el sistema contienen un conjunto de habilidades, la primera de ellas basta para alcanzar la meta que identifica al RAP.

Así pues, cuando el *Secuenciador* ha generado la secuencia de tareas, a cada una de ellas le asigna un RAP, y de cada RAP toma una habilidad para poner en ejecución por parte del controlador. La secuencia de tareas es almacenada en una agenda, que mantiene ordenadas las tareas y que permite realizar cambios sencillos en tiempo de ejecución sin necesidad de que el *Planificador* tenga que rehacer el plan.

Con la secuencia de tareas ya generada por el *Secuenciador*, el bucle de ejecución entra verdaderamente en la repetición de instrucciones que sigue:

1. El bucle de ejecución principal se ejecuta siempre que la tarea actual asignada al robot no sea la tarea especial de finalización.
2. El proceso principal (la aplicación cliente) se comunica con el robot (real o simulado) para pedirle información del entorno. Éste le devuelve los valores leídos por los sensores. Una vez los tiene, el proceso principal actualiza estos valores en la instancia del robot que tiene el subsistema de infraestructura.
3. El *Controlador* comprueba si la tarea actual asignada ha sido llevada a cabo con éxito. Para ello, ejecuta el método que comprueba la condición de éxito de la habilidad actual en ejecución.
    a. Si se ha alcanzado la meta de la tarea, se informa de ello.
    b. Si no, se ejecuta una iteración del método de ejecución de la habilidad actual. Esta ejecución requiere la lectura de los valores de los sensores y de reaccionar a los datos obtenidos del entorno, por lo que el *Controlador* accede a éstos a través de la instancia del robot de la arquitectura (no a través del cliente). Tras llevarse a cabo las operaciones adecuadas, el *Controlador* asigna una velocidad a cada rueda de la instancia del robot de la arquitectura, lo que posteriormente se convierte en una orden de la aplicación cliente al robot real o simulado.
4. El *Secuenciador* comprueba si la tarea actual asignada ha sido llevada a cabo con éxito.
    a. Si es así, se comprueba si hay más tareas pendientes en la agenda.

i. En caso de que existan tareas pendientes, el *Secuenciador* cambia la tarea actual en ejecución por la siguiente de la agenda.
ii. En caso contrario, la secuencia de tareas se ha completado, lo que quiere decir que el plan se ha llevado a cabo con éxito. El *Secuenciador* cambia la tarea actual por una tarea especial de control para que el controlador sepa que ha finalizado la ejecución. Todos los hilos terminan de forma segura su ejecución, y la aplicación cliente se desconecta del robot también de forma segura.

b. Si no se ha terminado de llevar a cabo la tarea actual, el *Secuenciador* espera en su bucle.

5. El proceso principal (la aplicación cliente) se comunica con el robot (real o simulado) para dar órdenes a sus actuadores, mandando la velocidad asignada por el control a las ruedas.

Todo este bucle se repite, como se ha dicho, hasta que la agenda del *Secuenciador* se queda sin tareas. En ese momento, el *Controlador* pone a ejecutar una tarea especial de control llamada "*Finalizar*" y, como se ha comentado, los hilos terminan su ejecución, y luego la aplicación cliente se desconecta.

Aunque esta es la rama principal de ejecución, también puede ocurrir que algo falle mientras se esté llevando a cabo el plan. La arquitectura ha sido diseñada e implementada para presentar robustez frente a este tipo de contratiempos.

Como se verá en apartados posteriores, el robot puede fallar a la hora de llevar a cabo una tarea, por ejemplo al producirse una colisión con un obstáculo. En este caso, el *Controlador* informa al *Secuenciador* de este fallo, se para la ejecución normal, y el *Secuenciador* modifica la agenda para intentar recuperarse del fallo. Esta modificación consiste en introducir como primera tarea de la agenda una tarea especial llamada "*Retroceder*" (ver anexo F) y la tarea actual volver a introducirla en la agenda a continuación de esta otra tarea especial. El resto de tareas pendientes de la agenda se mantienen en el mismo orden, pero más "atrasadas".

En la ejecución del controlador, la tarea "*Retroceder*" lleva a cabo las operaciones necesarias para que el robot se recupere de la colisión, y una vez se encuentre en una posición segura, continuar con la ejecución normal. La tarea especial de recuperación tiene una condición de éxito, al igual que el resto de tareas, por lo que tras la modificación de la agenda, el bucle de ejecución continúa como se ha descrito anteriormente.

Pero puede ocurrir que esta modificación de la agenda no sea suficiente para recuperarse de un fallo en la secuencia de tareas. En este caso, sería necesario elaborar un nuevo plan a partir de la submeta que no pudo alcanzarse con la tarea que falló. Debido a que no se implementó el *Planificador*, esta funcionalidad no está implementada tampoco, pero consistiría en tomar de nuevo la meta global que se dio al *Planificador* al inicio de la ejecución, y a partir del estado del modelo del mundo en el momento en que el plan falló, utilizar el algoritmo de planificación implementado para elaborar un nuevo plan. Una vez hecho esto, los pasos a seguir serían los mismos que tras la elaboración del plan original, y la ejecución continuaría.

3.4.2. Habilidades implementadas

En el apartado anterior se ha explicado el funcionamiento de la arquitectura, pero este funcionamiento no se ve reflejado por completo en el comportamiento del robot. Para probar que la arquitectura funciona correctamente es necesario diseñar e implementar una serie de funcionalidades que pueda ejecutar el robot y que se reflejen en la ejecución (ya sea en simulador o en el robot real).

Debido a que el objetivo principal de este proyecto es crear un marco y asentar las bases de una arquitectura híbrida, diseñada e implementada desde cero, la funcionalidad implementada para el robot se ha limitado a habilidades de navegación básicas. Estas habilidades deben permitir al robot moverse por su entorno de forma más o menos inteligente, sin chocar con obstáculos y también deben permitirle llevar a cabo planes de navegación sencillos.

A continuación se explican de forma genérica (sin entrar en las peculiaridades del algoritmo en función de su uso en simulación o en entorno real) las seis habilidades

de navegación básica que se han implementado y probado, así como las dos habilidades especiales de gestión y otra habilidad de navegación que sirve como ejemplo de cómo incluir nuevas habilidades que hacen uso de nuevos sensores que no se han tenido en cuenta en este proyecto.

- **Avanzar:** Esta habilidad se limita a asignar una determinada velocidad a las ruedas del robot (la misma en ambas ruedas) para que éste se desplace en línea recta. Como condición de éxito se ha impuesto la detección de un objeto delante del robot, es decir, el robot avanza en línea recta todo lo que el entorno le permite, y esta habilidad deja de ejecutarse en cuanto el robot se encuentra con un obstáculo en su trayectoria rectilínea.

- **Girar:** Esta habilidad también es muy sencilla en cuanto al comportamiento que presenta el robot. También se limita a asignar una velocidad a las ruedas, es función de si se quiere girar a la izquierda o a la derecha, del ángulo que se quiere trazar y un factor de velocidad, de forma que a mayor valor de este factor, el robot efectúa el giro más rápido. Se considera que el robot inicia la ejecución de esta habilidad con un ángulo de 0 grados respecto de su eje de dirección, por lo que la condición de éxito de la habilidad se cumple cuando se alcanzan los grados indicados en la inicialización de la habilidad.

- **EvitarObstaculo:** Esta habilidad consiste en realizar un giro suficiente como para evitar un obstáculo situado frente al robot. Éste detecta con los sensores de ultrasonidos un obstáculo demasiado cerca y calcula cuál es la dirección más adecuada por la que esquivarlo. Si los valores de los sónar izquierdos son más críticos que los derechos, el robot efectúa un giro hacia la derecha, y viceversa. El robot sigue girando hasta que deja de detectar el obstáculo con la mayoría de sónar frontales. La condición de éxito es precisamente esta. En cuanto el robot detecta que no hay peligro, finaliza la ejecución de la habilidad. Como puede entenderse de esta explicación, el robot se limita a efectuar un giro que evite la colisión con el obstáculo, independientemente del resultado de este giro. Es decir, el robot no retoma la trayectoria anterior, sólo esquiva el obstáculo. Es

otra habilidad la que se encarga de rodear los obstáculos para seguir con la trayectoria original del robot.

- **SeguirPared:** Esta habilidad consiste en seguir un contorno continuado en sentido horario. Normalmente, este contorno es una pared, pero puede ser cualquier otro objeto con un tamaño considerable que permita la correcta ejecución de la habilidad. Cabe destacar que el robot puede utilizar esta habilidad para seguir contornos irregulares, con salientes, entrantes y esquinas, siempre que estas últimas no sean demasiado pronunciadas o rincones que no permitan la movilidad del robot. Para llevar a cabo esta habilidad, el robot hace un uso exhaustivo de todos sus sensores de ultrasonidos. Con diversas medidas de los sensores más laterales, y utilizando ciertos valores de éstos como "umbrales", el robot crea la imagen de un "pasillo virtual" que tiene la misma forma que el contorno a seguir. Este pasillo le marca una guía, de forma que si se sale del límite izquierdo (es decir, se pega demasiado a la pared), el robot gira a la derecha y entra de nuevo en su pasillo virtual. Asimismo, si el robot se sale del límite derecho (es decir, se aleja demasiado de la pared), gira a la izquierda para entrar otra vez en el pasillo. Durante toda esta ejecución, el contorno puede presentar salientes o entrantes, que el robot detectará como obstáculos y procederá a esquivar, pero gracias a la utilización de este pasillo virtual, podrá rodearlos y considerarlos como lo que son: parte del contorno. No se ha implementado condición de éxito para esta habilidad, por lo que el robot continuará siguiendo una pared de forma indefinida. La mejor condición de éxito para esta habilidad es alcanzar un punto o posición concreta del mapa, pero debido a que no se ha implementado esta funcionalidad de la arquitectura, se deja abierta la puerta a una futura implementación de esta condición de éxito.

- **Recuperarse:** Esta habilidad es ejecutada normalmente como protocolo de recuperación de colisiones, puesto que no tiene sentido utilizarla en otro caso. La ejecución de la habilidad, que comienza cuando se ha detectado un choque del robot con un obstáculo, consiste en retroceder con velocidades distintas en cada rueda (en función de la posición de los parachoques activados) para que este movimiento no sea rectilíneo, sino que venga acompañado de un ligero giro que posteriormente permita continuar con la habilidad que se estaba ejecutando (o la

siguiente) sin volver a colisionar con el obstáculo. De esta forma, el robot describe un arco hacia atrás hasta que los sensores de ultrasonidos indiquen que no existe peligro de chocar con el mismo obstáculo (o no en la misma posición), siendo esta la condición de éxito de la habilidad.

- **CubrirDistancia:** Esta habilidad es parecida a la de avanzar en línea recta, pero con peculiaridades que hacen que el robot muestre un comportamiento más inteligente. Consiste en avanzar en línea recta, pero teniendo en cuenta una distancia objetivo. La condición de éxito, por tanto, es alcanzar dicha distancia avanzando en línea recta. Pero la habilidad también reacciona ante imprevistos. Si el robot se encuentra un obstáculo en la trayectoria, lo esquiva de forma reactiva, pero con el objetivo de volver a tomar la trayectoria original. Para ello, el robot hace uso de los valores odométricos medidos (distancia recorrida y giros efectuados) para rodear el obstáculo, en lugar de simplemente esquivarlo. Cuando los sensores indican que no hay peligro de colisión con el obstáculo, pues ha sido totalmente superado, y puede seguirse la trayectoria que se tenía originalmente, el robot se reorienta hacia el punto al que avanzaba y continúa su avance rectilíneo.

- **IrA:** Esta habilidad hace uso de un sensor de GPS para ir del punto inicial a un punto objetivo del mapa por el camino más corto, orientándose hacia dicho punto para intentar llegar a él con un avance en línea recta. Como los mapas y localizaciones no se han implementado, esta habilidad no puede ser ejecutada hasta que se implemente dicha funcionalidad, pero se deja la implementación en simulador como ejemplo de inclusión de nuevas habilidades y sensores. Como la habilidad utiliza en todo momento el GPS, si se encuentra un obstáculo, se limita a rodearlo tal y como hace en la habilidad CubrirDistancia. La diferencia con esta habilidad es que la reorientación hacia el punto y la condición de éxito no se llevan a cabo utilizando la odometría del robot, sino realizando lecturas sobre la posición y orientación del robot con respecto a los ejes del mapa mediante el GPS.

- **Esperar:** Habilidad especial de gestión. Su ejecución consiste en dar una velocidad de 0 a las ruedas del robot para que éste no se mueva. Esta habilidad

sólo se encuentra en ejecución mientras el controlador está esperando a que el secuenciador le mande la primera tarea a llevar a cabo. No tiene condición de éxito, pues en ningún momento es necesario realizar esta comprobación.

- **Finalizar:** Habilidad especial de gestión. Su ejecución consiste en asignar una tarea especial al estado del robot, llamada “Fin”. Cuando el bucle de ejecución se encuentra esta tarea como tarea actual para el robot, finaliza el bucle y se ejecutan las operaciones necesarias para que los hilos de ejecución finalicen de forma segura. No tiene condición de éxito, pues en ningún momento es necesario realizar esta comprobación.

Capítulo 4

Experimentación y Resultados

4.1. Pruebas del Sistema

Tras la implementación del subsistema *Infraestructura* y la parte básica de la capa reactiva de la arquitectura (subsistema *Controlador*, sin hilos ni comunicaciones), se decidió empezar con las pruebas para asegurar el buen funcionamiento de la conexión básica del sistema con las diferentes aplicaciones cliente. De esta forma, la implementación del resto de capas de la arquitectura se llevó a cabo una vez se tuvo la seguridad de que se podía ejecutar algo correctamente con cualquier aplicación cliente (simulador o robot real), y que los posibles problemas no derivan de una mala conexión entre cliente y arquitectura).

Prueba 1: Ejecución de una única habilidad (SeguirPared)

La primera prueba diseñada es la ejecución por parte del robot de una única habilidad, inicializada desde el principio. Es decir, el robot no tiene una meta, ni un plan, ni tareas a llevar a cabo. Simplemente se le "carga" una habilidad y se ejecuta indefinidamente. La habilidad seleccionada ha sido "*SeguirPared*", y se probó el correcto funcionamiento de su ejecución tanto en el simulador como en el robot real. Así, se consiguió una aplicación y su conexión con la arquitectura para cada uno de los dos tipos de aplicaciones cliente que componen el alcance de este proyecto (simulador Webots y robot real Pioneer 3-DX, utilizando las librerías de ARIA).

- **Mundo del simulador:** Se trata de un mundo sencillo rodeado de paredes con pequeñas formas trapezoidales en cada una de las 4 paredes. El robot se situó junto a una de las paredes, de forma que pudiera seguirlas en sentido horario (ver figura 51).

- **Controlador:** Se ha utilizado un controlador muy simple, llamado *pruebaControlador1 y pruebaAriaControlador1*, respectivamente (esta nomenclatura ha sido homogénea para todas las pruebas). El comportamiento del robot con este simulador consiste en seguir la pared junto a la que se encuentra, en sentido horario, procurando no perder en ningún momento la referencia por la pared. El controlador se ha programado

como dos aplicaciones cliente independientes: una para el simulador Webots y otra para funcionar con el robot real mediante ARIA.

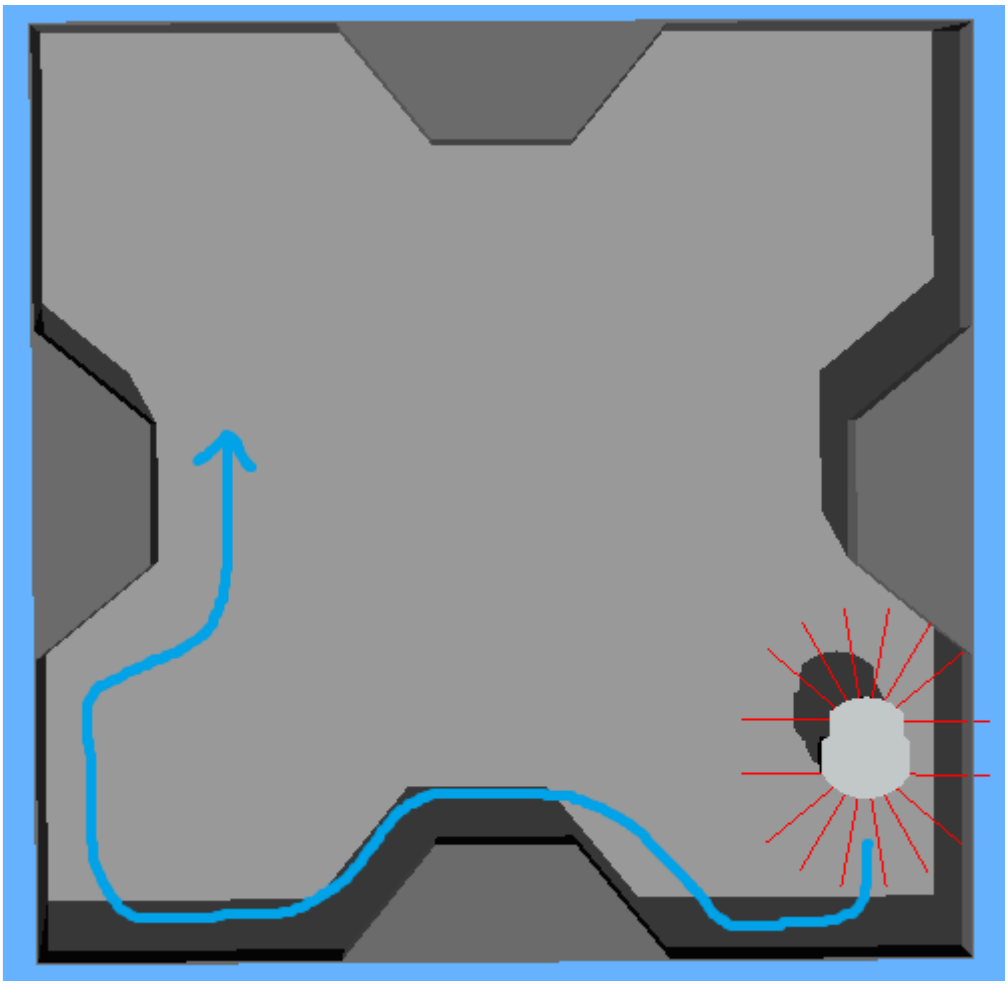


Figura 51: Prueba 1 del controlador en simulador

Resultados en simulador: El controlador logró inicializar correctamente la instancia del robot en la arquitectura, con sus correspondientes sensores y actuadores. A continuación se introdujo en el bucle de ejecución que ejecuta la habilidad “SeguirPared” en cada iteración. Se pudo observar en la recreación 3D del mundo en el simulador cómo el robot comenzó a seguir la pared junto a la que se encontraba, siguiendo siempre un “pasillo virtual” para evitar chocarse con la pared o separarse demasiado de ella y perder la guía. Tras dejar el simulador ejecutando durante un tiempo prudencial, se observó que el robot continuó con su camino, sin ninguna incidencia. Por tanto, se estableció que la prueba ha sido superada con éxito, lo que garantiza la correcta conexión del simulador con la arquitectura implementada.

Resultados en robot real: Tras unas pruebas iniciales de conexión con el robot, que permitieron comprobar la correcta conexión de la arquitectura con el robot real y adaptar el controlador a los valores e información concreta que proporciona éste, se ejecutó la habilidad "*SeguirPared*" en dos entornos diferentes. Cabe destacar que el robot intenta seguir una pared en sentido horario.

El primero de ellos se trata de un laboratorio de informática del edificio Sabatini, en la Universidad Carlos III de Madrid. Debido a la cantidad de mesas, sillas, armarios y demás mobiliario típico, el robot no tenía un espacio continuo por el que seguir una pared. Se utilizó una de las paredes, que contenía menos obstáculos, y una caja para simular esquinas y obstáculos. Varias de las pruebas realizadas resultaron satisfactorias, consiguiendo que el robot recorriera toda la pared, esquivando las cajas, de manera eficaz. Otras de las pruebas con el mismo controlador, en cambio, dieron resultados diferentes. En casi todas ellas, el robot acababa chocando con la caja a la hora de esquivarla, o con la pared, una vez que ya había superado la caja.

El segundo entorno probado fue uno de los pasillos del mismo edificio, el cual posee algunas características diferentes frente al primer entorno, entre las que cabe destacar los materiales de las paredes y la acústica. La prueba sobre este entorno fue similar a la primera: el robot colocado junto a una pared, en la que hay varias esquinas hacia dentro y hacia afuera, en forma de obstáculos que debe superar el robot. Los resultados de la ejecución de la habilidad fueron parecidos. El robot fue capaz de seguir la pared plana correctamente, pero en esta ocasión tuvo más problemas para afrontar las esquinas, especialmente las interiores (giros de 90º a la derecha).

Como se pudo observar en vista de los resultados de las primeras pruebas realizadas, la arquitectura funcionó correctamente tanto en simulador como en el robot real, permitiendo una gran independencia de la aplicación cliente. Este objetivo principal de la prueba (la conexión entre arquitectura y diferentes clientes) se consiguió con éxito. Sin embargo, hay que destacar las diferencias en los resultados obtenidos entre el simulador y el robot real. Por un lado, la simulación tuvo un funcionamiento perfecto: el robot no se chocó y siguió la pared del entorno virtual de forma correcta en

todas las ejecuciones, y no hubo diferencia de comportamiento entre ellas. Por otro lado, las pruebas en el robot real resultaron satisfactorias sólo en algunos casos. Esto demostró que en un entorno real se dan situaciones distintas y no existen dos ejecuciones iguales, por lo que el robot se comporta de manera distinta. Esto era algo de lo que ya se tenía conocimiento, y que se pudo probar empíricamente. Además, también se observaron diferencias entre distintos entornos reales, debidas principalmente a la acústica y los materiales del entorno. En el segundo entorno, los materiales y la acústica producían eco, lo que impidió el funcionamiento ideal de los sensores sónar del robot, haciendo que éste se "perdiera" en algún momento, y provocando más choques que en el primer entorno.

Prueba 2: Ejecución de una única habilidad (Recuperarse)

Antes de pasar a realizar las pruebas sobre el secuenciador, una vez se comprobó que la ejecución de habilidades y la conexión con el robot funcionan bien, se decidió hacer otras pruebas, similares a la primera, pero ejecutando otras de las habilidades básicas de navegación que se han implementado. De esta forma, se aseguró el correcto funcionamiento de éstas, y las pruebas sobre el secuenciador se centraron más en esta nueva capa y fueron más esclarecedoras. Así pues, la siguiente prueba que se realizó consistía en probar el funcionamiento de la recuperación del robot, es decir, su reacción frente a los choques en caso de que una habilidad de navegación falle (ya que es imposible asegurar el correcto funcionamiento de éstas en el 100% de los casos). Al igual que para la prueba anterior, se diseñó un controlador tanto para ejecutar en simulador como para ejecutar en un entorno real.

- **Mundo del simulador:** Se reutilizó el mismo mundo que para la prueba anterior, que consistía únicamente en 4 paredes con salientes en forma de obstáculos.

- **Controlador:** El comportamiento del robot con este controlador consiste avanzar en línea recta hasta chocarse (intencionadamente) con una pared. Una vez ocurra esto, el robot debería retroceder y girar en consecuencia para superar el obstáculo (si choca por el lado derecho, retrocediendo y girando a

la izquierda). Se puede observar gráficamente el comportamiento del robot en la figura 52.

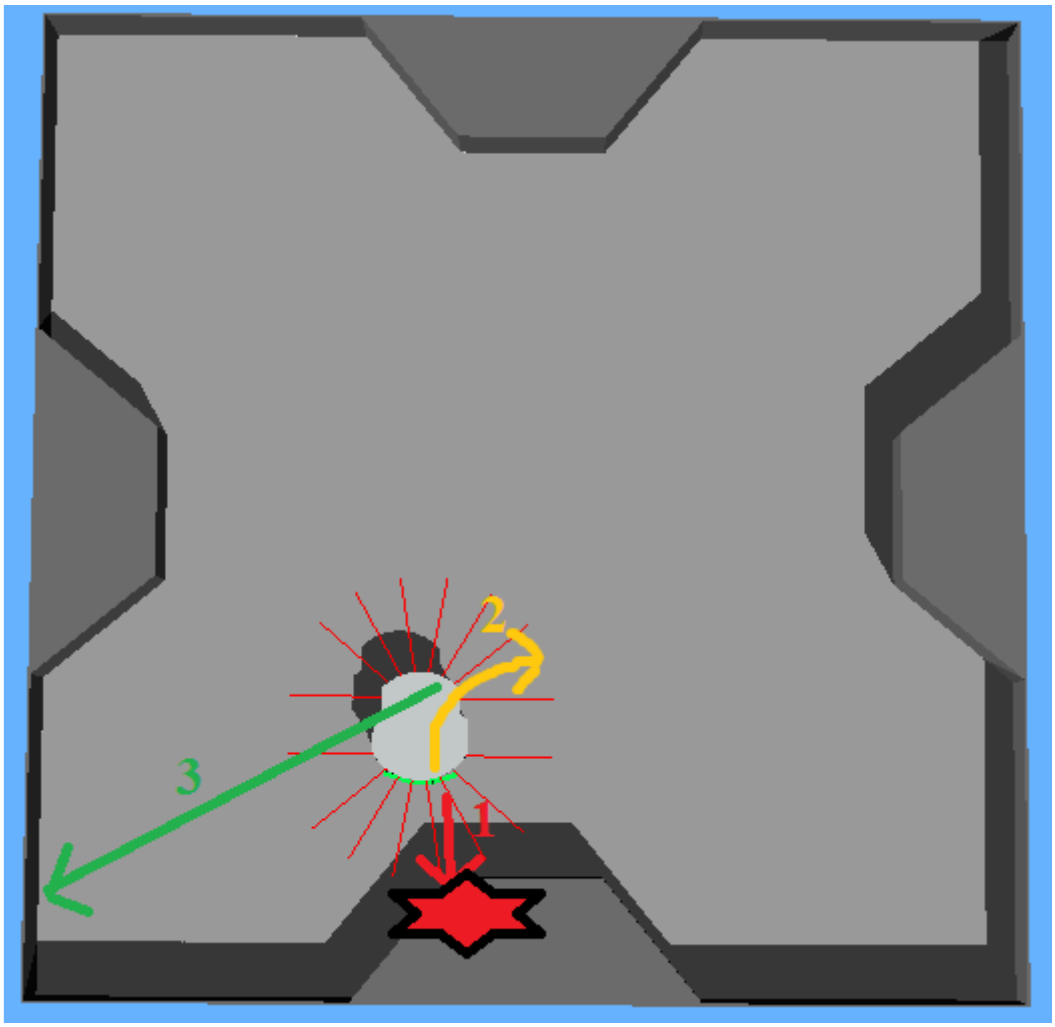


Figura 52: Prueba 2 del controlador en simulador

Resultados en simulador: El controlador realizó la conexión correctamente, como ya se había probado. A continuación se introdujo en el bucle de ejecución que ejecuta la habilidad "*Avanzar*" en cada iteración, hasta que el robot choca. Se pudo observar en la recreación 3D del mundo en el simulador cómo el robot avanzó en línea recta sin ningún problema. A continuación chocó con una de las paredes del mundo con el bumper frontal. Seguidamente retrocedió y giró hacia la derecha, para posteriormente seguir avanzando. Se dejó más tiempo la ejecución, provocando nuevos choques del robot con los distintos bumpers (izquierdos, derechos y frontal), y en todas las ocasiones mostró el comportamiento deseado. Por tanto, se estableció que la prueba fue superada con

éxito.

Resultados en robot real: En esta prueba sobre el robot real no fue necesario comprobar la correcta conexión con el robot. Simplemente se probó el mismo comportamiento diseñado para el simulador. Se utilizó el mismo laboratorio de informática que en la prueba anterior como entorno. Se colocó al robot en medio de unos cuantos obstáculos como armarios, paredes y cajas, y se dejó que el robot avanzara autónomamente. El robot avanzó en línea recta hasta chocar con el primer obstáculo (una pared). Tras esto, retrocedió un poco y giró hasta dejar de detectar la pared, momento en el que siguió su trayectoria rectilínea. A continuación chocó con un armario, y su comportamiento fue el mismo. Durante el tiempo que estuvo ejecutando el controlador, se dieron choques con los diferentes bumpers, y en ocasiones con varios a la vez, y en todas las ocasiones el robot se recuperó del choque de forma satisfactoria.

El resultado de esta prueba es que aseguró la recuperación del robot frente a choques si se da el caso mientras está llevando a cabo cualquier tarea, algo importante a la hora de poder seguir con la ejecución y replanificar en caso de fallo. No sólo eso, además esta es la primera prueba en la que entra en juego más de un tipo de sensor. Mientras que para seguir una pared sólo era necesario controlar los sensores sónar, en este caso se utilizan tanto éstos como los bumpers para detectar las colisiones. Así pues se probó satisfactoriamente el uso de varios sensores a la vez y sobre la misma habilidad, algo necesario para que el robot muestre comportamientos inteligentes de navegación.

Prueba 3: Ejecución de una única habilidad (EvitarObstaculo)

Para terminar con las pruebas de habilidades, se probó otro de los comportamientos básicos de navegación, que permite al robot moverse adecuadamente por diferentes entornos. Se trata de la habilidad "*EvitarObstaculo*", que consiste en esquivar cualquier tipo de objeto que detecte el robot durante su camino.

- **Mundo del simulador:** Se reutilizó el mismo mundo que hasta ahora, que consistía únicamente en 4 paredes con salientes, y se han incluido dos cajas, situadas en la zona central del mundo, que hacen las veces de obstáculos.

- **Controlador:** El comportamiento del robot con este controlador consiste en avanzar en línea recta en todo momento, pero si se detecta cualquier obstáculo, evitarlo realizando un giro suficiente como para no chocarse con él y poder seguir con una trayectoria rectilínea (ver figura 53). En este caso, las propias paredes también son obstáculos a evitar. El controlador se programó como dos aplicaciones cliente independientes: una para el simulador Webots y otra para funcionar con el robot real mediante ARIA.

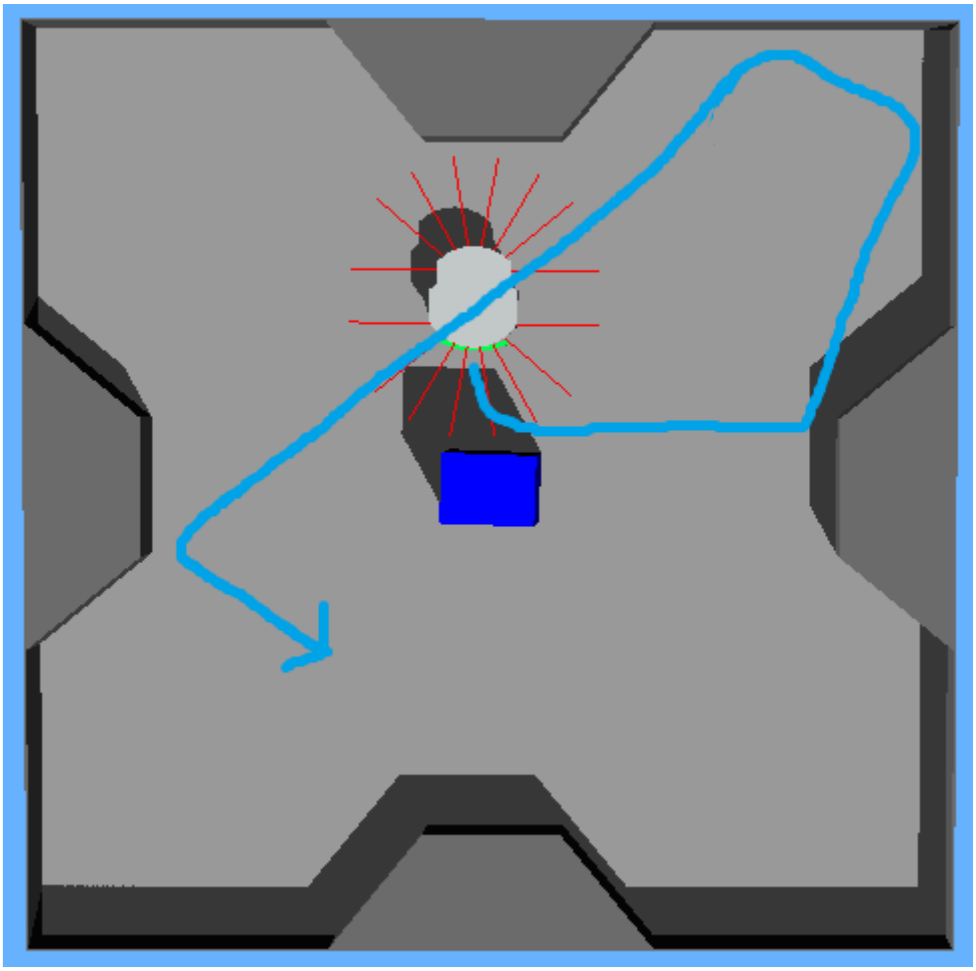


Figura 53: Prueba 3 del controlador en simulador

Resultados en simulador: El controlador ingresó en el bucle de ejecución que ejecuta la habilidad "*Avanzar*" en cada iteración, siempre que el camino esté libre para el robot. Se pudo en la recreación 3D del mundo en el simulador cómo el robot avanzó en línea recta sin ningún problema hasta detectar un obstáculo en su camino, momento en el cual el robot efectuó un giro con suficiente antelación como para no chocar con el obstáculo. Se dejó la ejecución durante un tiempo prudencial, comprobando que el robot no se chocó en ningún momento ni con los obstáculos del centro del entorno, ni con las paredes. Por tanto, se estableció que la prueba fue superada con éxito.

Resultados en robot real: Para la prueba sobre el robot real se probó también el mismo comportamiento diseñado para el simulador directamente, utilizando como entorno el Laboratorio de Informática. Se colocó al robot en medio de la estancia y se le dejó vagar (siempre en línea recta) libremente. El laboratorio contiene obstáculos como armarios, paredes, cajas, sillas o las propias personas que se encontraban en él en el momento de la prueba. El robot avanzó en todo momento en línea recta hasta detectar el primer obstáculo que se encontró (un armario que hacía esquina con una caja). El robot giró a la izquierda, a una buena distancia para evitar el obstáculo sin chocarse, pero inmediatamente se encontró con la caja, que hacía que el robot estuviera en un hueco muy estrecho para maniobrar. Aun así, detectó este nuevo obstáculo a tiempo y giró de nuevo a la izquierda, produciendo así un giro de 180º desde el momento en que llegó al primer armario. Este era uno de los rincones más complicados para el robot en el laboratorio, y salió de él con éxito. Se dejó al robot avanzar más tiempo, encontrándose con otros obstáculos que evitó correctamente. Incluso fue capaz de esquivar un obstáculo en movimiento (una persona que se cruzó en su camino). Por tanto, la prueba fue superada satisfactoriamente.

Con la correcta superación de esta prueba, y las anteriores, se consiguió asegurar el buen funcionamiento de las diferentes habilidades de navegación, tanto en simulador como sobre el robot real, lo que facilitó en su momento llevar a cabo pruebas sobre las capas superiores de la arquitectura. Además, el buen funcionamiento de estas habilidades era básico para poder seguir con el desarrollo de la arquitectura, ya que a partir de este momento, las pruebas se basaron en la secuenciación de diversas tareas

(que necesitan ejecutar distintas habilidad de forma correcta).

Prueba 4: Ejecución de un plan simple (Secuenciador básico)

Una vez implementado el *Secuenciador*, se preparó una prueba básica para verificar el correcto funcionamiento de éste, y de su integración con la capa inferior, el *Controlador*. Para ello, se elaboró un plan muy simple, consistente en 2 acciones, y sin una meta fija, para que el *Secuenciador* transformara dicho plan en tareas y comprobar la correcta asignación de RAP y habilidades. El plan consistió en que el robot siguiera indefinidamente una pared, pero para ello debía encontrarla previamente (ver figura 54), por lo que hicieron falta dos tareas, secuenciarlas bien y asignar dos habilidades distintas.

- **Mundo del simulador:** Se utilizó el mundo sencillo con paredes y obstáculos, incluyendo también un muro de color azul en el entorno para posteriormente permitir planes más complejos, haciendo que el mundo se divida en varias localizaciones.

- **Controlador:** Este controlador define un plan establecido en el código, muy simple, que se le pasa al robot nada más inicializarlo para que éste lo registre. En ese momento, la capa reactiva y la capa intermedia de la arquitectura ejecutan concurrentemente, junto con el propio controlador del robot. Al principio de la ejecución, el *Secuenciador* toma el mando y elabora la secuencia de tareas, con sus correspondientes RAP y habilidades asociadas. Le pasa la tarea actual a la capa reactiva para que ejecute la habilidad correspondiente, y se comprueba en cada iteración si se ha cumplido la condición de éxito. Cuando esto ocurre, el *Secuenciador* envía una nueva tarea al controlador. Por otro lado, en todo momento, el hilo principal de la aplicación toma información del robot y la envía al controlador, y éste le devuelve el valor de los actuadores para llevar a cabo los movimientos necesarios.

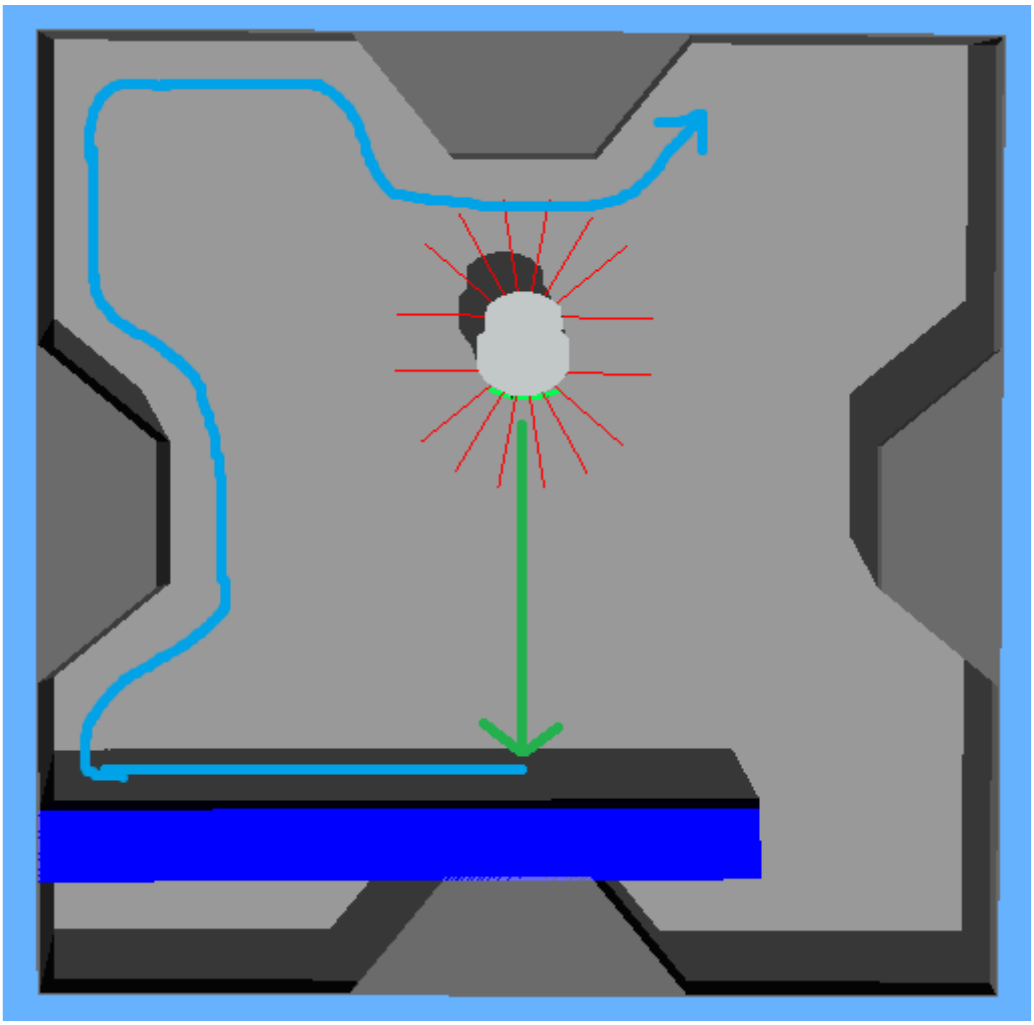


Figura 54: Prueba 1 del secuenciador en simulador

Resultados en simulador: Se compruebó un pequeño tiempo al inicio de la ejecución en el que el robot está parado. Este tiempo de alrededor de 2 segundos significa que el *Secuenciador* estuvo trabajando transformando el plan en la secuencia de tareas, y asignando los correspondientes RAP y habilidades. Además, se llevó a cabo un retardo (*delay*) de 1 segundo para evitar lecturas erróneas de la información de los sensores del robot. Una vez pasó este tiempo, se comprobó que el robot, situado en el centro del entorno, sin obstáculos alrededor, empezó a llevar a cabo la primera tarea: avanzar en línea recta. Esta tarea le permitió alcanzar una de las paredes del entorno. En ese momento, el robot detectó la pared, por lo que se cumplió la condición de éxito impuesta para la primera tarea, y el *Secuenciador* dio un nuevo paso en la secuencia, proporcionando al *Controlador* la segunda tarea: seguir la pared. Así, el robot

giró a su derecha y se alineó con la pared para poder seguir el contorno. Debido a que no se impuso en esta prueba una condición de éxito para esta tarea, el robot quedó dando vueltas al mundo de forma indefinida, que es lo que se buscaba. Por tanto, la prueba sobre el simulador se llevó a cabo satisfactoriamente.

Resultados en robot real: Los resultados obtenidos en el robot real fueron similares a los obtenidos en simulación. Como la prueba diseñada fue la misma para simulador y robot real, se colocó en robot en medio del laboratorio, orientado hacia una de las paredes formadas por armarios y cajas (ver figura 55). Tras la conexión del robot con la arquitectura, la primera operación fue realizar la secuencia de tareas. La primera diferencia que se observó con respecto al simulador es que el robot tardó ligeramente más tiempo en empezar a moverse, desde que se ejecutó la aplicación cliente hasta que se conectó con el robot, se calculó la secuencia de tareas y se llevó a cabo la primera iteración de la primera habilidad. Una vez hecho esto, el robot comenzó a avanzar despacio (para controlar cualquier fallo o choque y poder monitorizar por pantalla qué operación llevaba a cabo en cada momento). El robot alcanzó la pared y, detectándola a tiempo, se cumplió la condición de éxito de la primera tarea, por lo que el robot pasó a la segunda: seguir la pared. Por tanto, el robot giró a la derecha y evitó el choque, permitiéndole seguir la pared por el lado izquierdo. Debido a que esta tarea no tenía un fin preestablecido, el robot continuó siguiendo la pared (evitando obstáculos o recovecos) hasta que se detuvo la ejecución manualmente. Por tanto, la primera prueba del secuenciador sobre el robot real fue satisfactoria.

De esta primera prueba sobre el *Secuenciador* se pudo sacar un resultado positivo y una conclusión importante de cara al resto de pruebas. El resultado es que se comprobó que la capa intermedia de la arquitectura es capaz de elaborar una secuencia de tareas sencilla de al menos 2 tareas, y que el *Controlador* es capaz de pasar de una a otra correctamente, comprobando en cada momento la condición de éxito de cada habilidad. En cuanto a la conclusión que se sacó es que, al estar trabajando ya sobre una capa superior de la arquitectura, su funcionamiento es completamente independiente de la aplicación cliente. Mientras que las pruebas sobre el *Controlador* requerían

contemplar ciertas diferencias dependiendo de si se ejecutaban las habilidades sobre el simulador o sobre el robot real (valores de parámetros o pequeñas diferencias en los algoritmos, por ejemplo), el *Secuenciador* funciona igual tanto en simulación como en un entorno real, por lo que su buen funcionamiento en esta prueba aseguró el correcto funcionamiento en las siguientes a la hora de secuenciar tareas. Es la forma de encarar las habilidades y de refinar el comportamiento del robot lo que produjo diferencias en la pruebas entre el simulador y el robot real.

Figura 55: Prueba 1 del secuenciador en entorno real

Prueba 5: Ejecución de un plan complejo (Secuenciador)

Con el funcionamiento del *Secuenciador* ya probado, y una vez comprobado que es capaz de secuenciar un mínimo de 2 tareas y realizar correctamente el cambio entre una y otra, se diseñó una nueva prueba para el *Secuenciador*, pero esta vez con un plan más complejo. El plan consistió en 5 acciones, que se transformaron en tareas, y que el robot debía ejecutar una detrás de otra para llegar a su meta final. El robot se situó en la esquina superior derecha del entorno utilizado en simulación, orientado hacia su meta: la esquina inferior derecha, pero para llegar a ella no podía avanzar simplemente en

línea recta, y por ello debía llevar a cabo 5 tareas diferentes, que se exponen a continuación.

- **Mundo del simulador:** A partir del mundo de la prueba anterior, con las 4 paredes con salientes y el muro azul, se añadió un obstáculo más y se cambió de localización el muro, complicando el movimiento del robot. Además, aunque el robot se situóde tal forma que podría llegar a su destino avanzando en línea recta, la localización de los obstáculos permitió que el plan elaborado contuviera varias tareas diferentes, a llevar a cabo con hasta 4 habilidades distintas (ver figura 56).

- **Controlador:** Al igual que para la prueba anterior, se define un plan que consiste en 5 acciones (que se transforman en las correspondientes 5 tareas). Como ya se explicó qué funciones llevan a cabo las dos capas para secuenciar el plan, seleccionar RAP y habilidades asociadas a las tareas, se pasa a comentar el resultado esperado del plan. Para poder llegar a la esquina inferior derecha, el robot debía acceder a la parte inferior del entorno por un pequeño hueco dejado a la izquierda del muro azul. Para ello, primero se debían ejecutar las siguientes tareas: girar 45 grados a la derecha y avanzar en línea recta. Pero esto provocaría un choque con el obstáculo azul. Por tanto, tras estas dos tareas, habría que rodear en la medida de lo posible el obstáculo. A continuación se podría seguir avanzando en línea recta. Por último, la forma más sencilla con las habilidades implementadas de llegar al destino, fue seguir el contorno del muro azul. Por tanto, la quinta y última tarea del plan consistió en seguir la pared que forma este obstáculo.

Resultados en simulador: Se comprobó mediante mensajes en pantalla las tareas y habilidades seleccionadas para cada una de ellas, verificando la correcta secuenciación de tareas. Después se observaron los movimientos del robot: en primer lugar el robot giró correctamente 45º y comenzó a avanzar en línea recta, encarando el primer obstáculo azul. Al detectarlo, realizó un giro a la izquierda y lo esquivó. Las tres primeras tareas del plan, por tanto, se secuenciaron y llevaron a cabo correctamente. A continuación el robot avanzó de nuevo en línea

recta hasta llegar al muro azul, y una vez detectado éste, giró para poder seguir en sentido antihorario. Este movimiento final permitió al robot acceder por el hueco izquierdo a la parte inferior derecha del entorno, llegando así a la zona objetivo. Por tanto, la prueba sobre el simulador se llevó a cabo satisfactoriamente.

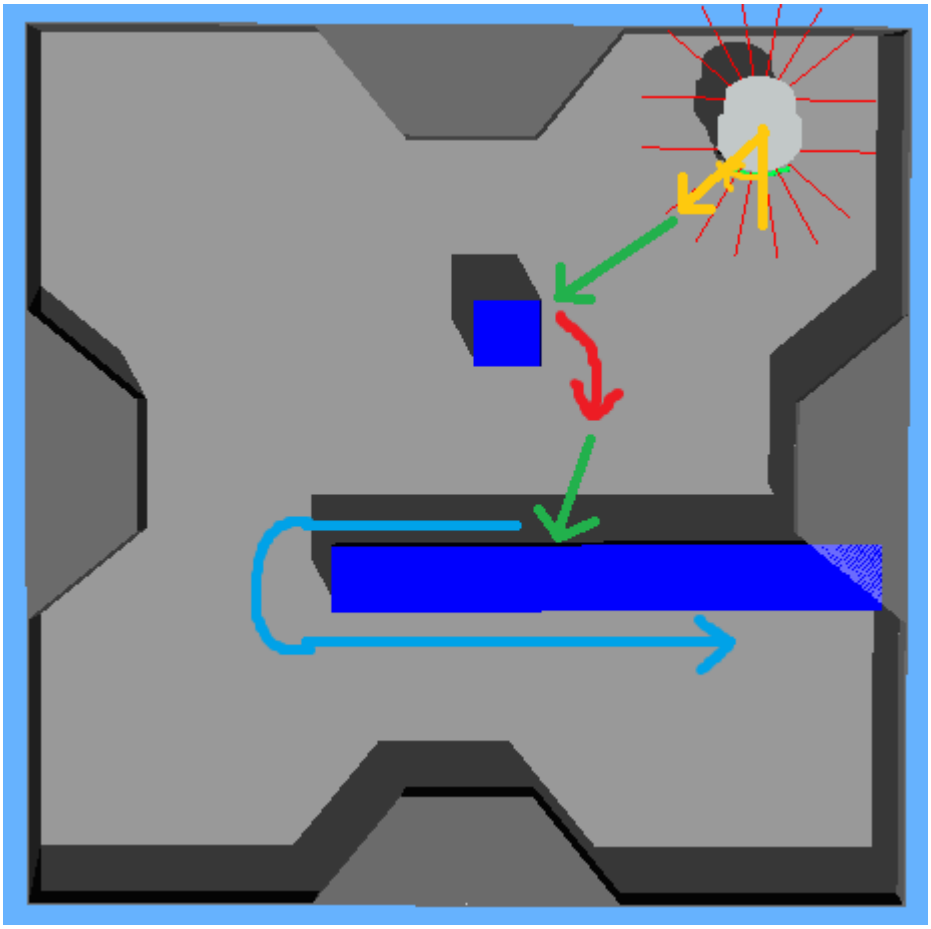


Figura 56: Prueba 2 del secuenciador en simulador

Resultados en robot real: Para esta prueba se preparó el entorno real de manera que el robot pudiera llevar a cabo un plan igual al probado en el simulador, es decir, cuya secuencia de tareas fuera la misma. Para ello, se colocó al robot en medio del entorno orientado hacia la puerta (ver figura 57). La meta se encontraba tras él, en una esquina opuesta (ver figura 58). Para llegar a la meta el robot necesitaría orientarse hacia la pared, llegar hasta ella (trayectoria en la que se encontraría un obstáculo) y finalmente seguir dicha pared hasta la esquina deseada. Una vez puesta la aplicación en ejecución, y tras esperar a la conexión

y secuenciación, se comprobó que la secuencia de tareas era correcta. Tras esto, el robot comenzó a moverse, y su comportamiento fue el siguiente: el robot realizó en primer lugar un giro de 90 grados hasta quedarse mirando hacia la pared (primer objetivo del plan para llegar a la meta), por lo que la primera tarea la llevó a cabo correctamente. La segunda tarea de la secuencia era avanzar en línea recta hasta la pared, sabiendo *a priori* que un obstáculo se interpondría en la trayectoria. El robot avanzó hasta el obstáculo, lo esquivó y continuó avanzando en línea recta. Al esquivar el obstáculo, el ángulo del robot cambió, permitiéndole alcanzar la pared en un punto más cercano a la meta. En esta prueba, el robot detectó la caja (que hacía a su vez de obstáculo de pared), por lo que en ese mismo momento la secuencia de tareas alcanzó la quinta y última tarea. Así, siguiendo el contorno de la pared (y los obstáculos pegados a ella, en este caso), el robot llegó a la esquina deseada. Por tanto, la prueba fue superada con éxito.

Figura 57: Prueba 2 del secuenciador en entorno real (a)

Figura 58: Prueba 2 del secuenciador en entorno real (b)

Como se esperaba tras los resultados de la primera prueba sobre el *Secuenciador*, la secuencia de tareas para esta prueba se llevó a cabo de forma correcta tanto en el simulador como en el robot real, pues una vez se aseguró que el algoritmo de secuenciación funcionaba correctamente, no hay diferencias a la hora de ejecutarse, sea la aplicación cliente un simulador o un robot real. Sí existieron diferencias, en cambio, en la ejecución de cada una de las tareas de la secuencia, como se pudo observar. Por ejemplo, los giros a la hora de esquivar un obstáculo. En el simulador estos giros fueron más pronunciados, mientras que en el robot real fueron más suaves, lo que se refleja en diferentes trayectorias para un mismo plan o secuencia, si bien es verdad que los entornos eran diferentes. Esto es normal, debido a los algoritmos y valores de los parámetros que se utilizaron para implementar las habilidades. Y este ha sido uno de los problemas a la hora de realizar estas pruebas sobre el *Secuenciador*; pese a que el plan y la secuencia de tareas eran los mismos, lo que se ha tenido que refinar para que se llevaran a cabo correctamente eran realmente las habilidades y sus parámetros. La conclusión principal de esto es que, posteriormente, cuando se incluyan nuevas funcionalidades, las diferencias o problemas en la ejecución vendrán de la capa inferior de la arquitectura, mientras que si se prueba el correcto funcionamiento de cualquier módulo de capas superiores, se estará probando para cualquier tipo de cliente de la

arquitectura.

Prueba 6: Recuperación de un fallo en la secuencia

Las pruebas anteriores aseguran el buen funcionamiento de las operaciones del secuenciador, su conexión con el controlador y la ejecución de planes más o menos complejos, con cierto número de tareas y habilidades diferentes. En este momento se decidió comprobar qué ocurre si una de estas secuencias "falla" y es necesario incluir nuevas tareas en una secuencia ya establecida. Para ello se utilizó el plan simple elaborado para la prueba 4, consistente en avanzar en línea recta hasta encontrar una pared y posteriormente seguirla de forma indefinida, de manera que el fallo en la secuencia y la recuperación esté más controlada que en un plan mucho más complejo.

- **Mundo del simulador:** Se utilizó el mismo mundo que para la prueba 4.
- **Controlador:** Como ya se ha comentado, el plan a llevar a cabo consistió en avanzar en línea recta hasta el muro azul y a continuación seguir la pared, en sentido horario. Para probar la funcionalidad concreta de recuperación de fallos en la secuencia, y debido a que las habilidades se han implementado buscando la optimización de los comportamientos, se tuvo que forzar el choque del robot con el muro azul (en caso contrario su comportamiento habría sido exactamente el mismo que para la prueba 4). Por ello, al avanzar en línea recta, el robot obvia en esta prueba la detección del muro azul. El resultado deseado era que el robot chocara, se recuperara (cuya tarea no se encuentra establecida en la secuencia de tareas) y luego siguiera la pared (ver figura 59).

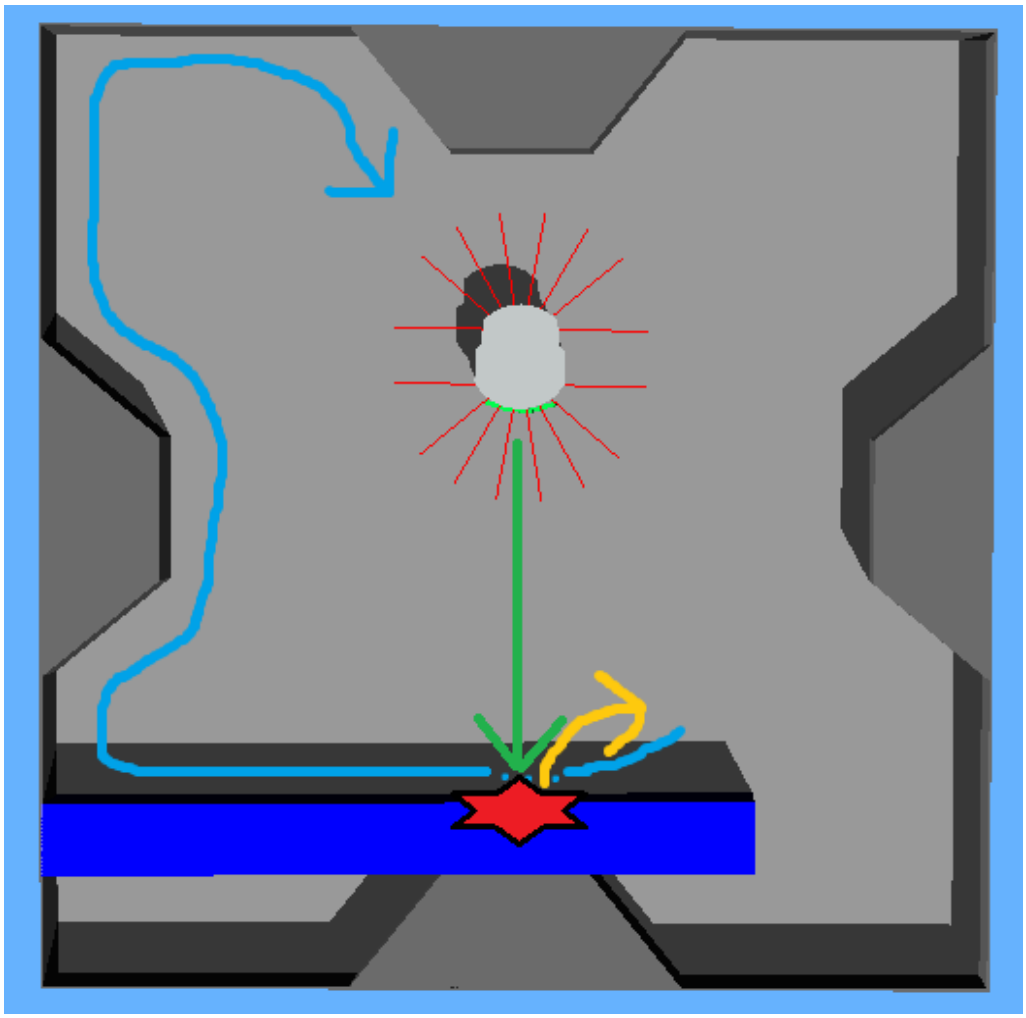


Figura 59: Prueba 3 del secuenciador en simulador

Resultados en simulador: Obviando las operaciones previas a la ejecución de la secuencia de tareas, una vez el robot comenzó a avanzar en línea recta, se comprobó si la ejecución en simulador se llevaba a cabo como se espera. Se observó que el robot llevó a cabo la primera de las dos tareas del plan: avanzar en línea recta hasta el muro azul. Una vez llegó hasta él, debido a que se estaba obviando la detección de obstáculos en esta prueba de forma premeditada, el robot chocó contra el muro. En este momento se esperaba que el robot modificara en tiempo de ejecución la secuencia de tareas, incluyendo como primera tarea en la agenda la recuperación del choque, y dejando el resto de tareas pendientes en el mismo orden en la agenda. En este caso, la agenda de tareas quedó de nuevo con dos de ellas: recuperarse del choque y seguir la pared. Se comprobó cómo el robot, efectivamente, dio marcha atrás y realizó un giro

para evitar el muro, alineándose además con el mismo. Una vez que la capa reactiva consideró que no había peligro, la tarea de recuperación se dio por concluida y tomó la siguiente tarea de la agenda, de forma que el robot se puso a seguir la pared en sentido horario, como estaba predefinido. Se comprobó, por tanto, que el robot es capaz de recuperarse de choques mientras ejecuta una secuencia de tareas, modificándola y retomándola a continuación de forma correcta, por lo que la prueba sobre el simulador se llevó a cabo satisfactoriamente.

Resultados en robot real: Al igual que para la primera prueba sobre el *Secuenciador*, a la hora de realizar la prueba sobre el robot real se utilizó el mismo entorno y el mismo plan que en dicha prueba. Así, el robot se situó en mitad del laboratorio orientado hacia una de las paredes, con el objetivo de avanzar hasta ella y luego seguir su contorno (ver figura 60). También para la prueba sobre el robot real fue necesario "obligar" a la primera tarea a fallar, por lo que al avanzar en línea recta se obvió intencionadamente la detección de la pared. Una vez explicado esto, se pasa a comentar el comportamiento del robot. Se puso el robot en ejecución y comenzó a avanzar en línea recta, tal y como en la prueba 4. Llegó a la pared, y tal y como estaba previsto, el robot chocó con ella, activando los bumpers. El robot permaneció un instante parado mientras se modificaba la secuencia, incluyendo una nueva tarea (recuperarse del choque) en el primer lugar de la lista de tareas. A continuación, el robot dio marcha atrás a la vez que realizaba un giro para alejarse del obstáculo y colocarse para evitarlo. Una vez hizo esto, y por tanto la tarea de recuperación se cumplió con éxito, el robot tomó la siguiente tarea pendiente (seguir la pared). Se pudo ver cómo el robot tuvo que hacer una maniobra para colocarse correctamente para poder seguir el contorno (se recuerda que existe un "pasillo virtual" que crea el robot para no perder la referencia de la pared). Una vez hecho esto, el robot continuó con esta habilidad como normalmente, siguiendo la pared indefinidamente hasta que se paró su ejecución manualmente. Al igual que en el simulador, el robot real también superó un fallo en la secuencia, por lo que la prueba se llevó a cabo satisfactoriamente.

Figura 60: Prueba 3 del secuenciador en entorno real

Con la correcta superación de esta prueba, y las anteriores, se consiguió asegurar el buen funcionamiento de la capa intermedia de la arquitectura, el *Secuenciador*. Se comprobó que su ejecución y funcionamiento es independiente del cliente, y que es la capa inferior, el *Controlador*, el que influye directamente en los problemas y en el funcionamiento de los comportamientos del robot al llevar a cabo un plan. Por tanto, se puede decir que se alcanzó cierto nivel de robustez para las dos primeras capaz de la arquitectura, y que se asegura su buen funcionamiento para el simulador Webots y para el robot Pioneer P3-DX manejado mediante Aria.

Prueba 7: Prueba avanzada (Deliberación + Reactividad)

Las pruebas anteriores se centraron en probar el funcionamiento de cada una de las dos capas de la arquitectura implementadas. Aunque con las pruebas realizadas para el *Secuenciador*, la capa inferior (*Controlador*) funcionó conjuntamente con ésta, los planes elaborados para dichas pruebas tenían en cuenta todos los objetos del entorno como si el robot poseyera un mapa del mismo, y el *Planificador* habría contado con

estos obstáculos a la hora de elaborar el plan. Por tanto se diseñó esta nueva prueba, que combina el comportamiento del *Controlador* con el del *Secuenciador* (con un plan sencillo), pero que demuestra la reactividad de la arquitectura, al obviar el plan posibles obstáculos que puedan aparecer en la trayectoria del robot.

- **Mundo del simulador:** Se utilizó el mismo mundo que para la prueba 3: el entorno cerrado por cuatro paredes y un obstáculo azul en el centro del mismo.

- **Controlador:** El plan que debía llevar a cabo el robot, que se situó en la parte superior del entorno, consistía en cubrir una distancia de cierto número de centímetros (la distancia que lo separa de la pared inferior), y a continuación seguir la pared de forma indefinida (ver figura 61). Este plan, que constó de dos acciones, no tiene en cuenta que el obstáculo azul se encontraba dentro de la trayectoria del robot, por lo que éste debería reaccionar ante el obstáculo, evitarlo y luego orientarse de nuevo para continuar con su plan.

Resultados en simulador: En primer lugar, se comprobó que el *Secuenciador* crea la secuencia de tareas correcta. En este caso, la secuencia constó de dos tareas, como ya se ha comentado: mantener la trayectoria hasta cubrir la distancia deseada y seguir la pared. Se comprobó que el robot empezó a avanzar en línea recta, como si no existiera ningún obstáculo delante. Cuando detecta el cubo azul, el robot lo esquivó por la izquierda, pero no sólo lo evitó, sino que lo rodeó, hasta superarlo por completo. En ese momento, como ya podía volver a mantenerse en su trayectoria, giró de nuevo para seguir con la línea recta que el obstáculo había interrumpido. Una vez alcanzó la pared inferior, giró para poder seguirla. En una segunda ejecución de esta misma prueba, el obstáculo azul se situó fuera de la trayectoria del robot, y a mitad de su recorrido hasta la pared inferior, se movió el obstáculo para situarlo frente al robot. Su respuesta reactiva al obstáculo fue la misma: evitarlo correctamente y retomar luego la trayectoria. Ambas ejecuciones demostraron que el robot “deliberó” para crear la secuencia de tareas, pero a su vez “reaccionó” a un obstáculo no previsto en el plan, por lo que el resultado de la prueba fue satisfactorio.

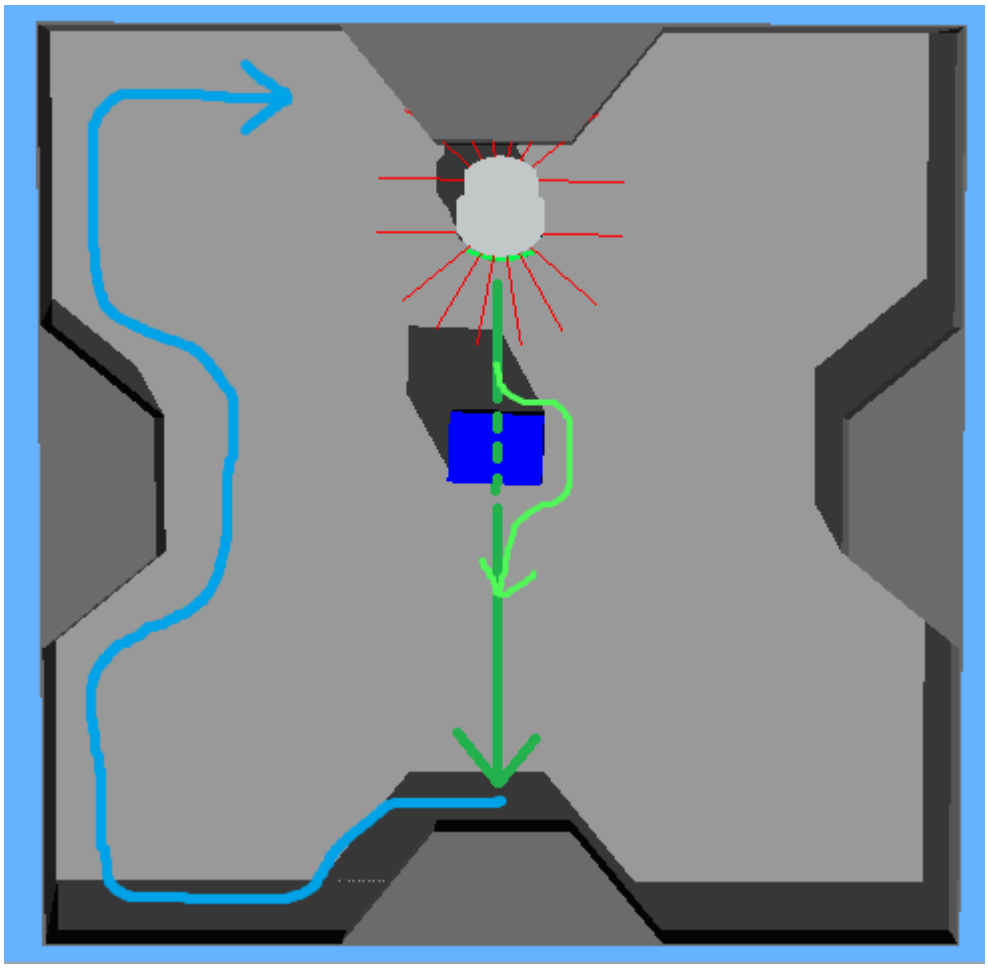


Figura 61: Prueba avanzada en simulador

Resultados en robot real: Para la prueba en entorno real, se utilizó el mismo entorno que hasta ahora. En el laboratorio se preparó un espacio vacío, en el que el robot tendría que mantener una trayectoria rectilínea hasta la pared, y luego seguir su contorno (ver figura 63). Tal y como se muestra en la figura 62, había una caja preparada para interponerla en la trayectoria del robot cuando éste se acercara a la mitad del camino. Los resultados obtenidos en esta prueba fueron similares a los obtenidos en el simulador. El robot avanzó en línea recta, siguiendo el camino marcado en el plan. En cierto momento, la caja obstaculizó el camino, el robot detectó el obstáculo y comenzó a rodearlo. Tras hacer esto y cuando ya era seguro reorientarse hacia el punto de destino, el robot realizó los giros necesarios para poder avanzar en línea recta y sin peligro hasta la pared. Una vez cubrió la distancia indicada, se orientó para poder seguir el contorno de la pared. En entorno real también se demostró el correcto funcionamiento de los

comportamientos deliberativos y reactivos combinados, por lo que la prueba obtuvo unos resultados satisfactorios.

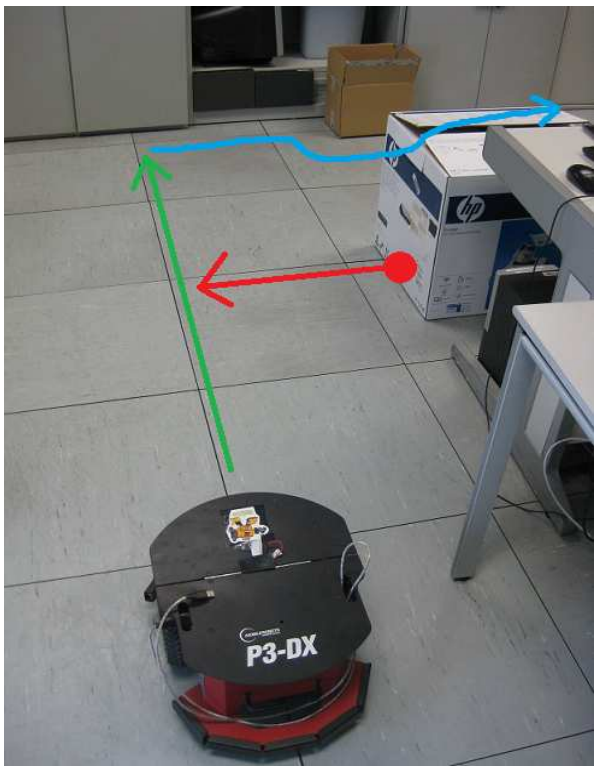


Figura 62: Prueba avanzada en entorno real (a)

El correcto funcionamiento de esta prueba avanzada aseguró la consecución de los objetivos marcados durante el desarrollo del proyecto. Esta prueba, aunque llevó a cabo un plan sencillo, demostró que el robot fue capaz de desenvolverse, tanto en simulador como en un entorno real, de manera deliberativa y reactiva conjuntamente. El robot fue capaz de elaborar una secuencia de tareas a partir de un plan establecido (cuya elaboración quedó fuera del alcance del proyecto). Mientras llevaba a cabo esas tareas, gracias a la capa inferior reactiva (el *Controlador*), el robot reaccionó a cambios en el entorno. Esto, unido a pruebas anteriores en las que se comprobó que también fue capaz de recuperarse de fallos en la ejecución del plan (colisiones), hace que se pueda decir que la arquitectura adquirió cierto grado de robustez.

Figura 63: Prueba avanzada en entorno real (b)

Sin embargo, tal y como dicta la experiencia, la robótica es un campo relativamente nuevo, en el que los algoritmos no son únicos e infalibles para todo tipo de problemas y en todo tipo de situaciones. En este caso particular, la ejecución en simuladores es una ejecución en un entorno controlado. Por otro lado, la ejecución en el robot real se lleva a cabo en un entorno no controlado, altamente dinámico y que depende de muchos factores que no siempre son controlables. Además, este proyecto asienta las bases y crea un marco para una arquitectura híbrida en tres capas, por lo que no se han controlado todos los factores que influyen en un entorno real, lo que en ocasiones dará lugar a fallos y comportamientos diferentes para mismos planes y secuencias de tareas. Esto es inevitable, pero también se han incluido mecanismos de recuperación que dotan al robot de cierto nivel de "inteligencia". Tomando esta implementación y estas pruebas como base, será sencillo incluir nuevas funcionalidades, nuevos comportamientos e implementar la capa superior de la

arquitectura, lo que hará que los controladores que se creen para el robot, tanto en simulación como en entorno real, sean mejores y puedan superar los problemas que en este momento puedan surgir.

4.2. Diferencias entre el Simulador y el Entorno Real

Aparte de los resultados obtenidos en las pruebas, mostradas en el apartado anterior, para comprobar el buen funcionamiento de todas las funcionalidades de la arquitectura, uno de los objetivos del proyecto es estudiar las diferencias encontradas entre la simulación y el robot ejecutando en un entorno real. Estas diferencias son generalizadas, es decir, aparecen por la naturaleza del problema general del control automático de robots, no por utilizar una paradigma o una arquitectura concreta.

Aunque ya se han comentado las diferencias apreciadas a la hora de probar la arquitectura, en este apartado se recogen y se exponen en detalle en qué varía el comportamiento de un robot en simulación frente a su utilización en el mundo real.

La forma de percibir: La primera diferencia fundamental entre un robot ejecutando en simulación y en un entorno real es la forma de percibir el mundo tal y como es. Esto es algo con lo que se debe lidiar desde el mismo momento en que se diseñan los algoritmos de control automático del robot. La diferencia no reside en utilizar un simulador o un robot real, sino en la plataforma, ya sea hardware o software, donde se ejecuta el algoritmo de control. La diversidad de plataformas y librerías genera grandes diferencias a la hora de percibir el mundo por el robot. El ejemplo más claro de esta diferencia son las mediciones de los sensores, como los ultrasonidos (sónar).

Han sido muchas las diferencias encontradas a lo largo del desarrollo del proyecto en este aspecto. La información del entorno que obtiene el robot se transforma en datos diferentes en función de la plataforma que esté ejecutando el mecanismo de control. Por ejemplo, en el simulador Webots, a mayor distancia de los objetos del entorno, menor valor dan los sónar. Se podría decir que en esta plataforma, los ultrasonidos indican el riesgo que existe de que el robot colisione con un obstáculo. Así, por ejemplo, el robot informa de un objeto detectado en la distancia con un dato cuyo valor puede ser 100, mientras que al acercarse mucho a dicho objeto, el robot informa

con un dato cuyo valor es cercano de 500. En cambio, al probar el mismo comportamiento en un entorno real con las librerías de Aria, el robot informa de un objeto relativamente lejano (unas decenas de centímetros) con un dato cuyo valor puede ser 2000, mientras que al acercarse a dicho objeto, el robot informa con un dato cuyo valor se va aproximando hasta 200 antes de chocar. En este caso, los ultrasonidos informan sobre la distancia al objeto: cuando más cerca, menor valor. Se trata de una forma de sentir completamente opuesta a la gestionada en el simulador, por lo que los algoritmos para la generación de habilidades y comportamientos cambian radicalmente y tienen que ser reprogramados en función de si se ejecutan en simulador o en entorno real (siempre buscando la reutilización de código y que sea transparente para la aplicación cliente).

Esta diferencia se ha encontrado entre el simulador y el entorno real, pero también pueden existir entre distintos simuladores, que utilicen distintas librerías; así como entre distintas plataformas hardware (distintos robots que hacen uso de los mismos sensores).

La forma de actuar: Al igual que el robot puede sentir de forma diferente en función de la plataforma, también es muy normal que actúe de forma diferente, entendiendo como forma de actuar no el comportamiento frente a un mismo estímulo o tarea, sino como la forma en que se dan las órdenes al robot. Diferentes plataformas software y también hardware ofrecen distintas librerías, que tal y como ocurre a la hora de tomar información del entorno, generan diferencias a la hora de realizar acciones sobre el entorno.

Realmente este caso es similar al expuesto anteriormente. Por ejemplo, cuando se dan órdenes al motor del robot para avanzar con una determinada velocidad en cada rueda. En este aspecto, también se ha comprobado que el simulador y el robot real mediante las librerías de Aria utilizan unidades de medida distintas. Mientras que en el simulador Webots se ha utilizado como velocidad normal para las ruedas un valor de 30 (para ambas, en el caso de un avance rectilíneo), lo que permite al robot moverse con cierta rapidez pero de forma segura, este valor es insuficiente al utilizarlo en entorno real, pues se traduce en una velocidad de 30 milímetros por segundo. En este caso, para realizar las pruebas sobre el robot en entorno real se ha utilizado un valor de 60, siendo aun así una velocidad muy baja para poder controlar en todo momento las acciones del robot y poder grabar unas pruebas que permitan una explicación exhaustiva. Una

velocidad normal, que se corresponda en cierta manera con la velocidad utilizada en el simulador, debería ser de 120: 4 veces más que la medida utilizada en el simulador.

Pero la diferencia en la forma de actuar no se limita únicamente a las medidas que utilizan las librerías. Éstas también proporcionan primitivas diferentes, lo que permite dar órdenes más o menos elaboradas al robot. Lo normal en el simulador ha sido utilizar la función que permite asignar cierta velocidad a cada rueda por separado, proporcionada por Aria para trabajar sobre el robot Pioneer 3-DX. Pero también ofrece otro tipo de funciones que permiten indicar una distancia para avanzar en línea recta, o dar bien una velocidad translacional, bien rotacional a las ruedas para efectuar movimientos rectilíneos o giros. Esta variedad de funciones puede influir también en la forma de implementar los comportamientos del robot, y es otra de las dificultades a la hora de realizar un diseño de la capa inferior de la arquitectura lo más independiente posible frente a las aplicaciones cliente que pueden hacer uso de ella.

El tiempo de respuesta: Dejando a un lado las diferencias derivadas de las librerías y características propias de las plataformas software o hardware, la diferencia más clara encontrada durante el desarrollo del proyecto es el tiempo de respuesta del robot. En simulación, el tiempo de respuesta del robot es inferior en todos los aspectos al tiempo de respuesta al ejecutar con el robot Pioneer 3-DX en entorno real. Tanto a la hora de tomar información de los sensores, dar órdenes a los actuadores, como a la hora de reaccionar y realizar operaciones de capas superiores, el simulador ejecuta en todo momento más rápido que el robot en entorno real. Si bien no es una diferencia de tiempo muy grande, sí es perceptible al ojo humano, por lo que se puede estar hablando de décimas de segundo, o incluso segundos.

Esta diferencia en el tiempo de respuesta es debida fundamentalmente a las comunicaciones del robot real con la arquitectura implementada. Mientras que al trabajar en el simulador, la arquitectura está corriendo sobre éste, al utilizar el robot real la arquitectura está ejecutando sobre un PC (en este caso un portátil colocado sobre el robot). Por otro lado, el robot está tomando la información de los sensores y enviando las órdenes a las ruedas, y tiene que comunicarse por cable, WiFi o cualquier otra tecnología con la arquitectura. Por tanto, ambas comunicaciones son distintas, y en el robot real tienen mucho más retardo que en el simulador.

Esta diferencia en el tiempo de respuesta forma parte también de la causa por la que existe una cuarta y última diferencia que se ha creído conveniente comentar en este

aparado, y que se explica a continuación.

El comportamiento del robot: A pesar de buscar en todo momento la independencia de la aplicación cliente, de la plataforma hardware o software, y crear habilidades o comportamientos que funcionen igual, este objetivo no es fácil de conseguir ni es algo trivial. Siempre van a existir diferencias de comportamiento en función de las aplicaciones cliente y las plataformas sobre las que se ejecuten los comportamientos. Estas diferencias pueden ser menores o mayores, pero están ahí, y en buena parte son efecto de las diferencias comentadas hasta el momento.

Como se ha visto, durante el desarrollo del proyecto se han tenido que implementar algoritmos con diferencias para las mismas habilidades debido a las diferentes formas de sentir o actuar en función de la plataforma software o hardware. Esto va a originar necesariamente diferentes formas de enfrentarse a los comportamientos por parte del robot: avanzar más rápido o más despacio, o detectar un obstáculo antes o después, por ejemplo. No sólo eso, el tiempo de respuesta es también fundamental a la hora de obtener un comportamiento por parte del robot más o menos eficaz y eficiente. Con algoritmos exactamente iguales que pudieran funcionar por igual en simulación y en entorno real, la diferencia en el tiempo de respuesta haría que los comportamientos obtenidos fueran diferentes, debido al retardo producido en la comunicación entre el robot y la arquitectura en el caso de un entorno real. El robot, por ejemplo, podría verse afectado por el retardo, resultando en una colisión con un obstáculo, que en simulador podría haberse evitado siempre.

Estas diferencias en el comportamiento del robot son las más difíciles de superar, y aparecen en la mayoría de los casos, por no decir el 100%, ya que son muchos los factores que influyen en el comportamiento del robot según esté ejecutando una tarea en simulador o en entorno real: plataformas, librerías, tiempos de respuesta, retardos, medidas, dinámica del entorno, etc.

Por todo esto, la creación de un marco común para el desarrollo de sistemas de control automático, cuya principal característica sea una independencia lo más alta posible frente a las plataformas hardware y software utilizadas, es una tarea ardua, complicada y que sigue siendo objetivo de multitud de proyectos de investigación.

Capítulo 5

Conclusiones y Líneas Futuras

En este capítulo se recogen las principales conclusiones obtenidas a través de los resultados del proyecto y la visión general del mismo, así como una serie de posibles trabajos futuros para mejorar e incluir nuevas funcionalidades en la arquitectura desarrollada.

Echando la vista atrás, en el primer capítulo de este documento se exponían una serie de objetivos que se pretendían alcanzar con el desarrollo del proyecto. El objetivo fundamental era la creación de un marco de trabajo, consistente en el diseño e implementación de una arquitectura híbrida, que permitiera a un robot móvil Pioneer 3-DX mostrar comportamientos inteligentes. Este objetivo ha sido el eje central del proyecto, y el resultado de varios meses de trabajo. Pero es importante mencionar que debido a la envergadura del proyecto y la importancia de los objetivos secundarios que se comentan a continuación, el alcance de la implementación de la arquitectura fue limitado a las dos capas inferiores de ésta, dejando la implementación de la capa deliberativa (el planificador) fuera del alcance del proyecto. Aun así, sí se implementó parte de esta capa para poder demostrar la funcionalidad completa de la capa intermedia, y presentar una arquitectura que funciona tanto a nivel deliberativo como reactivo.

Entre los objetivos secundarios, había ciertas características de la arquitectura que eran bastante deseables de cumplir. Todas ellas también se han tenido en cuenta, y han quedado reflejadas en los diversos capítulos de los que ha constado la presente memoria. También se verá que se ha cumplido con estos objetivos en las líneas futuras que se indican como posibles continuaciones de este proyecto. La arquitectura ha sido especialmente diseñada para poder ser extensible con la máxima sencillez posible (en el Manual de Usuario se explica al futuro desarrollador qué pasos debe seguir para incorporar nuevas funcionalidades a la arquitectura y al robot). Además, la decisión de diseñar una u otra arquitectura ha sido el resultado de un estudio de muchas arquitecturas híbridas utilizadas en la actualidad, de las cuales se ha estudiado la modularidad, otro de los objetivos planteados al inicio del proyecto.

Por otro lado, las pruebas realizadas en la fase final del proyecto han demostrado el correcto funcionamiento de la arquitectura en dos plataformas diferentes: una software (el simulador) y una hardware (el propio robot), lo que hace que la arquitectura

haya resultado lo suficientemente independiente de la aplicación cliente. Estas pruebas también han reflejado las habilidades con las que se ha dotado al robot móvil para conseguir comportamientos inteligentes de navegación básica.

Finalmente, un objetivo muy importante de este proyecto era documentar las diferencias existentes entre utilizar un sistema de control automático en simulación y en entorno real. El hecho de realizar pruebas en ambos entornos ha permitido sacar a la luz una serie de resultados, que se expusieron en el capítulo anterior de forma detallada y que es importante recoger aquí también. Utilizando un mismo sistema de control automático, el comportamiento observado en simulación y en el robot real es ligeramente diferente. Estas diferencias se aprecian en la forma en que el robot siente y actúa, en su tiempo de respuesta y, en general, en su comportamiento mostrado ante situaciones similares. Estas diferencias de comportamiento son intrínsecas al problema planteado, puesto que la ejecución en simulación se realiza en un entorno más controlado que el mundo real, y una simulación intenta recrear las características y condiciones de éste, algo que no es alcanzable al 100%.

En cuanto a las líneas futuras, la propia motivación del proyecto y su alcance ha impulsado un diseño e implementación de cara a trabajos futuros con el objetivo de mejorar el presente proyecto, incluir nuevos módulos, características y funcionalidades. Cada parte del diseño y la implementación se ha tenido en cuenta para dejar abierta una puerta a multitud de mejoras y nuevos añadidos.

A continuación se presenta una serie de posibles mejoras y trabajos para completar la arquitectura desarrollada en este proyecto, y que servirán para hacer de esta arquitectura una herramienta muy potente para el control automático de robots móviles.

- Implementación completa del planificador (capa superior deliberativa de la arquitectura).

- Sustitución del mecanismo de secuenciación basado en RAP por otro mecanismo diferente, como por ejemplo ESL (Execution Support Language).

- Implementación de distintos tipos de planificadores para la capa deliberativa, o sustitución de esta parte del sistema por un planificador ya existente.

- Inclusión de nuevos tipos de sensores y actuadores en la arquitectura, de los que puede hacer uso el robot para llevar a cabo otro tipo de tareas y dotarle de comportamientos inteligentes más elaborados.

- Inclusión de nuevos módulos de habilidades, como el módulo de visión, que puede hacer uso o no de otros tipos de sensores y actuadores, y su integración con módulos de habilidades ya implementados para dotar al robot de comportamientos inteligentes más elaborados.

- Inclusión de nuevas habilidades para módulos ya implementados, como el módulo de navegación, que permita al robot llevar a cabo más tareas diferentes dentro del mismo módulo.

- Extensión del modelo del mundo, incorporando nuevos tipos de mapas y nuevos elementos del entorno del robot a tener en cuenta, tales como baterías en el caso de elementos del propio robot, o cargadores y balizas en el caso de elementos del entorno.

- Utilización de la arquitectura en nuevas y diversas plataformas hardware y software, haciendo uso de las librerías adecuadas y de las operaciones proporcionadas por la arquitectura desarrollada en este proyecto.

- Utilización de la arquitectura para el control automático en un sistema multi-robot, incluyendo los módulos necesarios de habilidades, modelo del mundo y dominio del planificador y tareas.

Aparte de todas estas mejoras y nuevas funcionalidades que pueden incluirse en la arquitectura desarrollada, el objetivo intrínseco de la misma es su utilización para la resolución de diversos problemas o para la investigación. Utilizar la arquitectura como lo que es (un sistema de control automático) para la resolución de problemas y tareas por parte de un robot móvil, siendo éste un campo muy abierto y que abarca una gran

cantidad de ámbitos. Por citar algunos ejemplos, asistencia en el hogar, reconocimiento de entornos peligrosos, localización y creación de mapas u otros tipos de problemas que puedan resolverse con técnicas de navegación básica. Ampliando la arquitectura con nuevos módulos de habilidades y funcionalidades para el robot, el espacio de problemas a ser resueltos utilizando esta arquitectura se agranda enormemente.

Como se puede observar, hay un gran número de posibilidades para trabajar en un futuro con la arquitectura desarrollada en este proyecto. Es cuestión de las necesidades de los problemas concretos que se pretendan resolver con la arquitectura el seguir o no algunas de estas líneas de trabajo. En cualquier caso, con los objetivos perseguidos y alcanzados durante el desarrollo del proyecto, se asegura una buena base y un marco de trabajo sencillo para que futuros usuarios puedan llevar a cabo sus estudios, investigaciones o proyectos en este ámbito del control automático de robots móviles autónomos.

Para terminar, se puede consultar el Presupuesto del proyecto en el anexo B del presente documento.

Capítulo 6

Referencias

Alami et al., 2000 R. Alami, R. Chatila, S. Fleury, M. Herrb, F. Ingrand, M. Khatib, B. Morisset, P. Montarlier, and T. Simeon, (2000). Around the lab in 40 labs... In IEEE International Conference on Robotics and Automation, ICRA'O0, San Francisco.

Albus, 1991 Albus, J.S., (1991). "A Theory of Intelligent Machine Systems". IEE/RSJ International Workshop on Intelligent Robots and Systems. Osaka, Japan.

Albus and Barbera, 2005 J.S. Albus and A.J. Barbera, "RCS: A cognitive architecture for intelligent multi-agent systems," In Proceedings of the 5th IFAC/EURON Symposium on Intelligent Autonomous Vehicles (2004).

Arkin, 1990 Arkin, R.C., (1990). Integrating Behavioural, Perceptual and World Knowledge in Reactive Navigation. Robots and Autonomous Systems, 6, 105-122.

Bonasso, 1997 Bonasso, R.P., Firby, J., Gat, E., Kortenkamp, D., Miller, D., Slack, M., (1997). Experiences with an Architecture for Intelligent Reactive Agents. Journal of Experimental and Theoretical Artificial Intelligence, vol. 9, no. 2.

Braitenberg, 1984 Braitenberg, V., (1984). Vehicles: Experiments in Synthetic Psychology. MIT Press, Cambridge.

Brooks, 1986 Brooks, R.A., (1986). A robust layered control architecture for a mobile robot. IEEE Journal of Robotics and Automation, 2, (1), 14-23.

Echeverria et al., 2010 Echeverria, G., Lassabe, N., Degroote, A., Lemaignan, S., (2010). Modular Open Robots Simulation Engine: MORSE.

Firby, 1989 R. James Firby. Adaptive Execution in Complex Dynamic Worlds. Technical report YALEU/CSD/RR#672, Yale University, 1989.

Gat, 1990 Gat, E., M. Slack, D.P. Miller and R.J. Firby. 1990. Path Planning and Execution Monitoring for a Planetary rover. In Proc. of the IEEE Int. Conf. on Robotics and Automation. Vol. 1 Cincinatti, OH (US): pp. 20-25.

Gat, 1992 Gat, E., (1992). Integrating planning and reacting in a heterogeneous asynchronous architecture for controlling real world mobile robots In Proceedings of the National Conference on Artificial Intelligence. San Jose (CA): pp. 809-815.

Gerkey, 2003 B. P. Gerkey, R. T. Vaughan, and A. Howard. The player/stage project: Tools for multi-robot and distributed sensor systems. In In Proceedings of the 11th International Conference on Advanced Robotics, pages 317–323, 2003.

Hexmoor, 1993 H. Hexmoor, J. Lammens, and S. C. Shapiro. Embodiment in GLAIR: A grounded layered architecture with integrated reasoning for autonomous agents. In D. Dankel, editor, Proceedings of the Florida AI Research Symposium, pages 325--329, 1993.

IEEE, 2000 IEEE Std 1471-2000, IEEE Recommended Practice for Architectural Description of Software-Intensive Systems, Sponsor: Software Engineering Standards

Committee of the IEEE Computer Society, Approved 21 September 2000. (ISO/IEC 42010:2007)

Koenig, 2004 N. Koenig and A. Howard. Design and use paradigms for gazebo, an open-source multi-robot simulator. In In IEEE/RSJ International Conference on Intelligent Robots and Systems, pages 2149–2154, 2004.

Konolige, 1998 Konolige, K., Myers, K., (1998). The Saphira Architecture for Autonomous Mobile Robots. Artificial Intelligence and Mobile Robots. D. Kortenkamp, R. Bonasso, R. Murphy, editors, MIT Press, 1998.

Lammens, 1993 Lammens, J. M.; Hexmoor, H. H.; and Shapiro, S. C. 1995. Of elephants and men. In Steels, L., ed., The Biology and Technology of Intelligent Autonomous Agents. Berlin: Springer-Verlag, Berlin. 312—344

Lindström, 2000 Lindström, M., Orebäck, A., and Christensen, H. 2000. Berra: A research architecture for service robots. In Internacional Conference on Robotics and Automation.

Martinet and Patin, 2008 Martinet, P., Patin B. 2008. PROTEUS: A platform to organise transfer inside French robotic community. 3rd National Conference on "Control Architectures of Robots". Bourges, May29-30, 2008

Murphy, 2000 Murphy, R.R., (2000). Introduction to AI Robotics. MIT Press.

Patin, 2009 Partin B. and PROTEUS Consortium. 2009. PROTEUS Platform proposal to ANR 2009 ARPEGE call.

Poza and Posadas, 2009 Poza Luján, J.L., Posadas Yagüe, J.L. 2009. Revisión de las Arquitecturas del Control Distribuido

Simmons, 1994 R. Simmons. Structured control for autonomous robots. IEEE Transactions on Robotics and Automation, 10(1):34-43, Feb. 1994.

Carmen website http://carmen.sourceforge.net/

Cyberbotics website http://www.cyberbotics.com/

CycabTK website http://cycabtk.gforge.inria.fr/wiki/doku.php

GOSTAI website http://www.gostai.com/

ERSP website http://www.evolution.com/products/ersp/

MobileRobots website http://www.mobilerobots.com/Mobile_Robots.aspx

MS Robotics website http://www.microsoft.com/robotics/

Orocos website http://www.orocos.org/

Player website http://playerstage.sourceforge.net/

ROS website http://www.ros.org/wiki/

ROSETTA website http://www.fp7rosetta.org/

Skilligent website http://www.skilligent.com/

Anexos

Anexo A

Manual de Usuario

Arquitectura híbrida para robot Pioneer 3-DX

Manual de usuario

José Luis Díaz Cebrián

ÍNDICE

1. Introducción

Este documento es un manual para el usuario y futuro desarrollador de la arquitectura híbrida para el robot Pioneer 3-DX desarrollada como Proyecto Fin de Carrera entre enero y julio de 2011 para el departamento de informática de la Universidad Carlos III de Madrid.

El objetivo de este documento es facilitar y orientar al usuario de esta arquitectura en su ejecución en las diversas plataformas donde ha sido probada, así como guiarle en la tarea de incluir nuevas funcionalidades y módulos de la arquitectura. En este manual se recogen todos los pasos que debe dar el usuario para realizar estas tareas, así como todas las consideraciones que debe tener en cuenta, pues la arquitectura diseñada únicamente asienta las bases y no ha sido implementada en su totalidad.

El manual comienza con una descripción detallada de los elementos más importantes del robot Pioneer 3-DX, como son el panel de control y los sensores utilizados para la implementación de las habilidades de navegación básica de las que se ha dotado al robot.

A continuación se desarrollan los dos capítulos fundamentales de este manual. El primero de ellos explica cómo utilizar y ejecutar la arquitectura híbrida, tanto en una plataforma software (simulador Webots) como en una plataforma hardware (el propio robot Pioneer 3-DX). Este capítulo incluye también una guía detallada para que el usuario pueda crear sus propias aplicaciones cliente para utilizar en estas dos plataformas.

El siguiente capítulo expone las posibles mejoras y nuevas funcionalidades que pueden implementarse en la arquitectura, explicando al futuro desarrollador cómo realizar estos cambios y qué pasos debe dar para incluir nuevos módulos. Esta tarea será más sencilla de lo que parece a simple vista, debido al diseño específico que se ha realizado, buscando en todo momento la extensibilidad y modularidad de la arquitectura.

Por último, se presenta un breve resumen con los pasos principales descritos en el documento para crear un controlador genérico, sirviendo este capítulo final como una guía de referencia rápida para el usuario.

2. Descripción del robot

Los robots Pioneer pertenecen a una familia de robots móviles, de dos y cuatro ruedas, fabricados por Mobile Robots. Todos los robots pertenecientes a esta familia (Pioneer 1, Pioneer AT, Pioneer 2-DX, Pioneer 3-DX, etc) comparten una arquitectura común para la investigación y desarrollo de aplicaciones robóticas.

Figura 1: Aspecto lateral del robot Pioneer 3-DX

Este robot es uno de los más pequeños del mercado, pero tiene la capacidad de presentar gran cantidad de comportamientos inteligentes, gracias a las especificaciones técnicas de esta familia de robots. Apenas mide 22 cm. de alto y menos de 40 cm. de diámetro. La siguiente figura muestra las medidas de este pequeño robot.

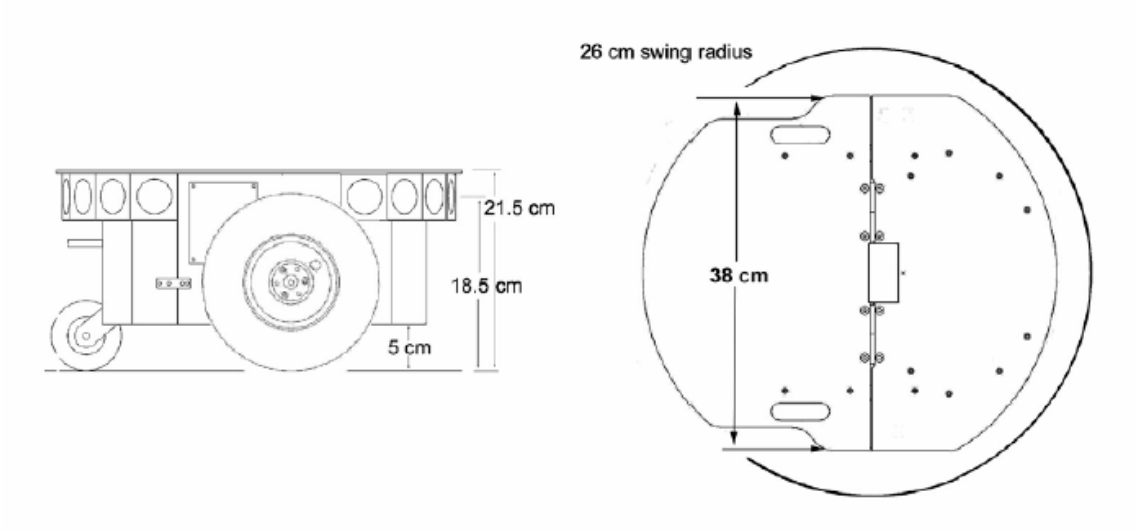


Figura 2: Dimensiones del robot Pioneer 3-DX

Su peso es de alrededor de 10 Kg. con una única batería, pero puede soportar hasta 23 Kg. de carga adicional, pudiendo incorporar elementos como portátiles, brazos robóticas, cámaras o láser encima de su "lomo".

A continuación se describen los elementos más importantes del Pioneer 3-DX, que han sido utilizados durante el desarrollo del proyecto, y a los que se hará mención en próximos capítulos del presente documento.

Panel de control: Consiste en un panel de botones e indicadores que permiten el control del robot y obtener información del microcontrolador incorporado en él. Además, posee un puerto serie. La siguiente figura muestra el panel de control del robot, y a continuación se explica brevemente cada elemento.

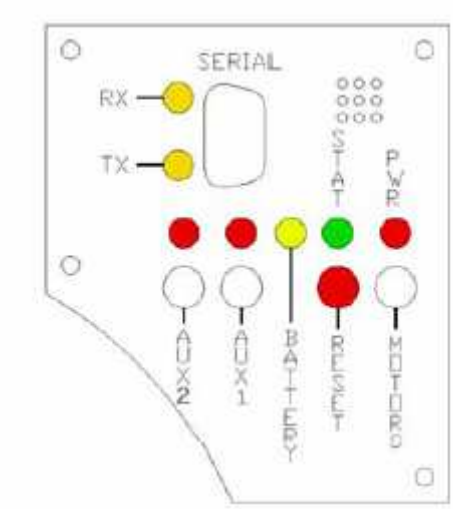


Figura 3: Panel de control del Pioneer 3-DX

- **Serial:** Puerto de conexión serie, a través del cual se realiza la conexión con el robot. Posee dos indicadores: RX (indicador de entrada de datos) y TX (indicador de salida de datos).
- **PWR:** Indicador de encendido.
- **Stat:** Indicador de estado del robot. Parpadea lentamente cuando está a la espera de una conexión con el cliente. Si conecta con el cliente parpadeará rápidamente.
- **Battery:** Indicador del nivel de carga de la batería del robot. Con carga completa, el LED se encuentra verde, y va cambiando de amarillo a rojo a medida que la carga disminuye.
- **Motors:** Botón que cambia el estado de los motores, habilitándolos o deshabilitándolos.
- **Reset:** Resetea el robot.
- **Aux1/Aux2:** Indicadores de conexiones auxiliares.

Ultrasonidos (sónar): El Pioneer 3-DX posee ocho sensores de ultrasonidos colocados en la parte delantera del robot. Dos de ellos se encuentran en las caras laterales, mientras que los otros seis se encuentran en la cara delantera, con una orientación de 20º de diferencia entre cada uno de ellos, lo que permite al robot detectar obstáculos que se encuentren en un ángulo de 180º con respecto al eje de dirección del robot. Para la detección de objetos, estos sensores emiten señales de ultrasonidos y miden el tiempo que tardan en recibir el eco de la señal para calcular la distancia a la

que se encuentran los objetos. Como se puede ver en la siguiente figura, cada uno de los ocho sensores de ultrasonidos se identifica por un número de 0 a 7, de derecha a izquierda, mirando de frente al robot.

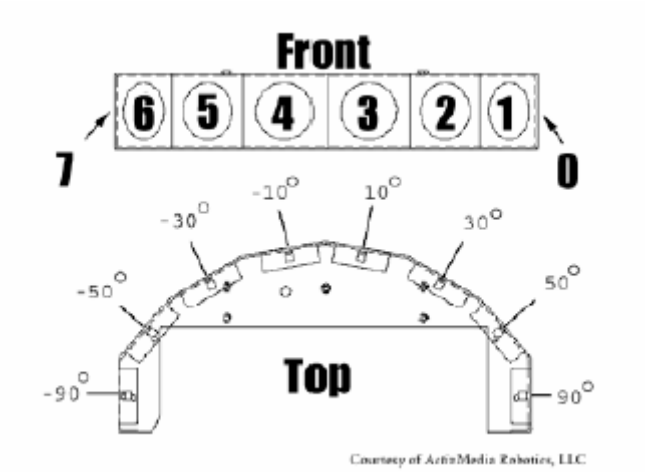


Figura 4: Disposición de los sensores de ultrasonidos

Parachoques (bumpers): Se trata de otro tipo de sensores que permiten al robot detectar colisiones con los objetos del entorno. El Pioneer 3-DX cuenta con diez parachoques: cinco delanteros y cinco traseros, colocados de tal manera que evitan los choques desde cualquier ángulo con el cuerpo del robot. Estos parachoques son sensores que envían una señal al microcontrolador sólo en caso de que se activen por una colisión. Mientras este hecho no suceda, los parachoques no emiten ningún tipo de señal. La siguiente figura muestra la disposición de los parachoques del robot.

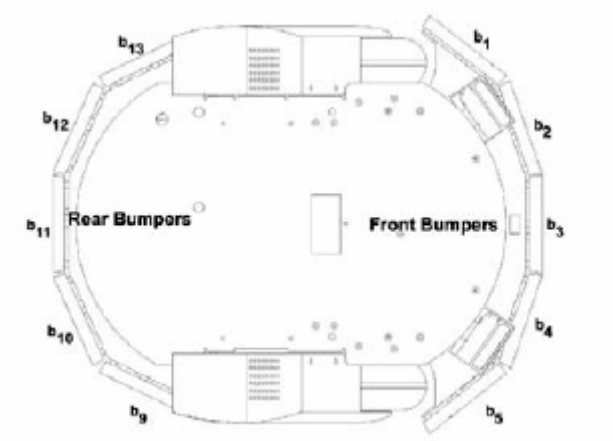


Figura 5: Disposición de los parachoques

Motores y ruedas: El sistema de dirección del robot cuenta con dos ruedas motrices, con sus respectivos motores, cada uno equipado con un "encoder" óptico de alta resolución que permite conocer la posición, giro y velocidad de las ruedas en todo momento. También posee una pequeña rueda posterior que aporta estabilidad al robot.

Por último, a continuación se explica cómo conectar el robot con un PC para la ejecución de controladores que utilicen la arquitectura híbrida. Aunque el robot soporta varios tipos de conexión, como TCP/IP, se explicará la conexión utilizada durante el desarrollo del proyecto. Esta conexión se ha realizado mediante el puerto serie del robot.

Para realizar esta operación, se debe conectar el cable por el puerto serie al robot y por el puerto USB al PC. Si el PC no tenía instalado previamente los *drivers* del cable, se instalarán automáticamente. A continuación se debe comprobar que el puerto serie virtual creado por el PC es *COM1*, ya que es el que se utiliza en el robot. Para ello, se deben seguir los siguientes pasos:

- Pulsar con el botón derecho del ratón sobre el icono de *MiPC*.
- Seleccionar *Propiedades*.
- Acceder a la pestaña *Hardware*.
- Seleccionar *Administrador de dispositivos*.
- Acceder a la sección *Otros dispositivos* y comprobar el nombre del puerto serie virtual.

Una vez hecho esto, el robot ya está conectado correctamente al PC en el que se ejecutará la arquitectura, permitiendo que el robot Pioneer 3-DX lleve a cabo los planes indicados en los controladores que se creen para el robot.

3. Utilización de la arquitectura

Junto con la arquitectura, se han implementado una serie de ejemplos que han servido como pruebas para comprobar el correcto funcionamiento de la arquitectura. Como las pruebas del sistema se diseñaron de forma incremental, algunos de estos ejemplos no tienen interés en su ejecución, pero otros más elaborados sí pueden servir al usuario para comprender mejor el funcionamiento de la arquitectura.

En este apartado se explicará cómo ejecutar algunos de estos ejemplos, pero también qué pasos debe dar el usuario para crear sus propios controladores ejecutables, tanto en el simulador como en entorno real.

3.1. Ejecución de ejemplos

Para explicar cómo ejecutar pruebas ya implementadas para la arquitectura, se tomará como ejemplo una de ellas, indicando qué pasos se deben dar para su correcto funcionamiento. La prueba de ejemplo que se utilizará será la llamada *pruebaAvanzada*. Esta prueba consiste en la ejecución de un plan sencillo consistente en dos acciones: cubrir una cierta distancia en línea recta y a continuación seguir el contorno que forma la pared del mundo (virtual o real).

A continuación se presentan los pasos que se deben seguir para ejecutar esta prueba, tanto en el simulador Webots como con en el robot Pioneer 3-DX.

3.1.1. Simulador

En primer lugar, es importante mencionar que cada una de las pruebas incluidas con el código de la arquitectura lleva un archivo de mundo virtual de Webots asociados, que contiene el entorno preparado para ejecutar la prueba. Se pueden consultar todas las pruebas, con la información de sus respectivos controladores y archivos de mundo virtual en la memoria del proyecto (capítulo 4.1)

En el caso de la prueba de ejemplo, el controlador para el simulador Webots se llama *pruebaAvanzada*, contenido en la siguiente ruta (siempre suponiendo que nos encontramos en la carpeta raíz de la arquitectura): *bin/controllers/pruebaAvanzada/pruebaAvanzada.class*. El código fuente de este

ejemplo se encuentra en *src/pruebaAvanzada.java*. El archivo de mundo virtual asociado a esta prueba se llama *mundoOdometria*, y se encuentra en la siguiente ruta: *bins/worlds/mundoOdometria.wbt*.

Para poder ejecutar este ejemplo, se deben seguir los siguientes pasos:

- Abrir el archivo del mundo .wbt con el simulador Webots. Esto se puede realizar haciendo doble clic sobre el archivo mundoOdometria.wbt, o bien ejecutando el simulador, pulsando en Abrir y seleccionando el archivo correspondiente.
- Comprobar que el robot tiene asignado el controlador adecuado. Para ello, acceder al árbol de la escena, desplegar el nodo del robot (*Pioneer*) y comprobar el nodo *controller*. En caso de que no aparezca en este campo el archivo del controlador (*pruebaAvanzada*), pulsar el botón que contiene 3 puntos suspensivos y seleccionar el controlador adecuado.
- Debido a que el planificador no está implementado, los parámetros de las habilidades no se asignan de forma automática (puesto que no se crean planes). Por este motivo es necesario acceder al código de la clase *Habilidades* (*src/controlador/gestorHabilidades/Habilidades.java*) e indicar en el constructor de la clase la distancia que se pretende cubrir en el ejemplo. Busque la línea que crea la instancia de la habilidad *CubrirDistancia* y compruebe que los parámetros son iguales que los siguientes:

```
this.habilidades.put("CubrirDistancia", new CubrirDistancia(robot, sim, 35));
```

- Compilar el código si ha sido necesario retocarlo.
- Pulsar el botón *Revert* del simulador.
- Ahora ya está todo preparado para ejecutar el controlador. Pulsar el botón *Play* del simulador para llevar a cabo la ejecución.
- Durante la ejecución de la prueba, puede moverse el obstáculo azul para que el robot reaccione a éste y lo esquive mientras está llevando a cabo las dos tareas del plan.

3.1.2. Entorno real

En el caso de las pruebas de la arquitectura en entorno real, es necesario realizarlas en un entorno más o menos controlado, colocando al robot en una posición

inicial desde la que pueda llevar a cabo el plan. En este caso, el robot debería colocarse mirando hacia una de las paredes del entorno y a una distancia de ésta aproximadamente igual a la distancia que se desea cubrir, y que es el parámetro (en centímetros) de la habilidad que le permitirá llevar a cabo esta tarea.

Para las pruebas en entorno real, los controladores tienen el mismo nombre que los usados en el simulador, pero incluyendo la palabra *Aria*, que hace referencia a la interfaz que permite comunicar la arquitectura con el robot. En este caso, el controlador se llama *pruebaAriaAvanzada*, contenido en la siguiente ruta: *bin/ /pruebaAriaAvanzada.class*. El código fuente de este ejemplo se encuentra en *src/pruebaAriaAvanzada.java*.

Para poder ejecutar este ejemplo, se deben seguir los siguientes pasos:

- Conectar el robot con el PC en el que se ejecutará la arquitectura (ver capítulo 2 de este manual).
- Debido a que el planificador no está implementado, los parámetros de las habilidades no se asignan de forma automática (puesto que no se crean planes). Por este motivo es necesario acceder al código de la clase *Habilidades* (*src/controlador/gestorHabilidades/Habilidades.java*) e indicar en el constructor de la clase la distancia que se pretende cubrir en el ejemplo. Busque la línea que crea la instancia de la habilidad *CubrirDistancia* y si el robot se encuentra a 2 metros de distancia, compruebe que el parámetro es correcto:

```
this.habilidades.put("CubrirDistancia", new CubrirDistancia(robot, sim, 200));
```

- Compilar el código si ha sido necesario retocarlo.
- Ejecutar el controlador *pruebaAriaAvanzada*, bien lanzando directamente el archivo .class ejecutable con Java (*java pruebaAriaAvanzada*), bien utilizando un entorno de desarrollo que permita la ejecución, como Eclipse o NetBeans.
- Durante la ejecución de la prueba, puede colocarse algún tipo de obstáculo para que el robot reaccione a éste y lo esquive mientras está llevando a cabo las dos tareas del plan.

NOTA: Como se ha explicado anteriormente, será necesario indicar los parámetros de las habilidades modificando el código de la clase Habilidades mientras que la capa superior de la arquitectura (planificador) no esté implementada. Una vez incluida esta capa, el planificador debería dar parámetros a ciertas acciones (como aquellas relacionadas con cubrir una cierta distancia o girar un determinado número de grados) para que las habilidades puedan instanciarse en tiempo de ejecución con los parámetros adecuados.

3.2. Creación de controladores

Además de ejecutar controladores ya implementados durante el desarrollo del proyecto, la arquitectura permite la creación de controladores nuevos para ejecutar otro tipo de planes por parte del robot. Una vez la arquitectura esté implementada por completo, bastaría con crear un único controlador genérico que se encargara de recibir los datos de configuración para que la propia arquitectura cree los planes y lleve a cabo la ejecución del mismo, haciendo uso de la información obtenida por los sensores del robot, y mandando órdenes a los actuadores.

En este apartado se expone la forma de implementar un controlador que funcione con la arquitectura híbrida. El controlador implementado difiere en pequeños bloques dependiendo de la plataforma cliente que se utilice, no siendo así todo el bloque relacionado con la arquitectura. Siguiendo los pasos que se describen a continuación, es muy sencillo crear un controlador para ejecutar planes con la arquitectura híbrida.

3.2.1. Simulador

Para crear un controlador que será ejecutado en el simulador Webots, el primer paso es incluir las librerías necesarias para hacer uso de las primitivas del software de simulación, así como importar el subsistema de infraestructura, que es el encargado de comunicar al controlador con la arquitectura híbrida. Además, y de forma temporal mientras no se incorpore la capa superior de la arquitectura, será necesario acceder al gestor de planes para crear un plan predefinido en el controlador.

```
import com.cyberbotics.webots.*;
import infraestructura.*;
import planificador.gestorPlanes.*;
```

La clase que define el controlador debe heredar de la clase Controller, que forma parte de las librerías de Webots. Como atributos de la clase se recomienda utilizar una serie de arrays donde almacenar los valores leídos de los sensores y actuadores del robot. En el caso de ejemplo, se utilizan los sensores sónar y bumpers, así como los encoders del motor.

```
public class pruebaAvanzada extends Controller{

        static final int NUM_SENSORES=16;
        static int []ds = new int[NUM_SENSORES];

        static final int NUM_BUMPERS=3;
        static int []bs = new int[NUM_BUMPERS];

        static double []enc = new double[2];
```

A continuación es necesario definir una función de inicialización, que se encargará de habilitar en el mundo virtual del simulador todos los sensores que se pretendan utilizar para controlar el robot. Esta función de inicialización puede copiarse tal y como se presenta en la siguiente figura, haciendo coincidir los nombres de los atributos con los creados en el paso anterior.

```
public static void reset(){
        /* Inicialización de sensores de distancia */
        String text="ds";
        int i;
        for (i=0;i<NUM_SENSORES;i++) {
                text=text+String.valueOf(i);
                ds[i] = robot_get_device(text);
                text="ds";
        }
        for(i=0;i<NUM_SENSORES;i++) {
                distance_sensor_enable(ds[i],64);
        }

        /* Inicialización de sensores bumper */
        text="bump";
        for (i=0;i<NUM_BUMPERS;i++) {
                text=text+String.valueOf(i);
                bs[i] = robot_get_device(text);
                text="bump";
        }
        for(i=0;i<NUM_BUMPERS;i++) {
                touch_sensor_enable(bs[i],64);
        }

        /* Inicialización de los encoders de odometría */
        differential_wheels_enable_encoders(64);
}
```

El método main() del controlador, tras realizar una llamada a la función reset(), contiene todo el código del controlador. Lo primero que se debe hacer es crear una instancia del robot con la que trabajará la arquitectura. Para ello hay que definir e inicializar cada uno de los elementos de los que hará uso el robot, utilizando para ellos las operaciones que proporciona el subsistema de infraestructura. En el siguiente ejemplo se crea un robot que haría uso de los sensores sónar y bumpers, así como de un actuador motor. El estado del robot se inicializa con una tarea de inicio, que se dedicará a esperar a que la arquitectura le mande la primera tarea a llevar a cabo.A la hora de inicializar la instancia del robot, el último argumento indica si se ejecutará en simulador o en entorno real. Como en este caso la ejecución se llevará a cabo en simulador, habrá que ponerlo a “true”.

```
/* Inicialización de los elementos del robot */
Robot robot;
Sensor[] sensores_robot = new Sensor[2];
Actuador[] actuadores_robot = new Actuador[1];
Estado estado = new Estado(null, new Tarea("Inicio", null));
Sonar sonar = new Sonar(NUM_SENSORES);
sensores_robot[0] = sonar;
Bumpers bumpers = new Bumpers(NUM_BUMPERS, 0);
sensores_robot[2] = bumpers;
Motor motor = new Motor();
actuadores_robot[0] = motor;
robot = new Robot(estado, sensores_robot, actuadores_robot, true);
```

El siguiente paso (únicamente si no se ha implementado un planificador) será crear un plan directamente en el controlador. Para ello hay que definir en orden cada una de las acciones que formarán parte del plan, instanciarlo y mandar el plan al robot para que pueda trabajar con él.

```
/* Creación de un plan para pasar al secuenciador */
Plan plan;
Meta meta_global = new Meta("DetectarAlgo", null);
LinkedList<Accion> acciones = new LinkedList<Accion>();
acciones.add(new Accion("MantenerTrayectoria"));
acciones.add(new Accion("EncontrarAlgo"));
plan = new Plan("PLAN4", meta_global, acciones);
robot.getArquitectura().getSecuenciador().setPlan(plan);
```

Adicionalmente, antes de entrar en el bucle de ejecución que se repetirá hasta que el robot finalice la ejecución del plan, es recomendable hacer una lectura de todos

los sensores que se van a utilizar para evitar posteriores lecturas erróneas, debido que hasta que se inicializan por completo los sensores, es bastante normal tomar los primeros datos con información incorrecta, que puede hacer fallar al controlador.

```
for(int i=0; i<NUM_SENSORES; i++)
        robot.getSonar().setValorSonar(i, (double)distance_sensor_get_value(ds[i]));
for(int i=0; i<NUM_BUMPERS; i++)
        touch_sensor_get_value(bs[i]);
try {
        Thread.sleep(1000);
} catch(InterruptedException e){
        System.out.println("ERROR del hilo principal mientras dormía");
}
```

Una vez hecho esto, el bucle principal debe contener siempre las mismas instrucciones que se describen a continuación. La primera parte del bucle principal consiste en tomar los datos proporcionados por los sensores en el simulador, y almacenarlos en la instancia del robot que maneja la arquitectura. Los valores de algunas variables y comparaciones se han obtenido mediante la experimentación con el simulador, y se utilizan aquellos que han ofrecido mejores resultados en las pruebas.

```
while(robot.getEstado().getTarea().getNombre().compareTo("Fin")!=0){

        /* Lectura de encoders del motor */
        robot.getMotor().setEncoders(
                (double)differential_wheels_get_left_encoder(),
                (double)differential_wheels_get_right_encoder());

        /* Lectura de sónar */
        boolean cuidado = false;
        for(int i=0; i<NUM_SENSORES; i++){
                robot.getSonar().setValorSonar(i, (double)distance_sensor_get_value(ds[i]));
                if(i==0 || i==1 || i==2 || i==13 || i==14 || i==15)
                        if(robot.getSonar().getValorSonar(i)>150)
                                cuidado = true;
        }

        /* Lectura de bumpers */
        int[] bumps = new int[NUM_BUMPERS];
        for(int i=0; i<NUM_BUMPERS; i++){
                bumps[i] = touch_sensor_get_value(bs[i]);
                if(bumps[i]==1)
                        robot.getBumpers().setValor((i*100)+100);
        }
        if(bumps[0]==0 && bumps[1]==0 && bumps[2]==0 && !cuidado)
                robot.getBumpers().setValor(0);
```

Una vez leída la información de los sensores, se debe comprobar si se ha producido alguna colisión, pues es necesario informar de este hecho a la arquitectura para que realice las modificaciones necesarias en la agenda de tareas para continuar con la ejecución. Tras esta comprobación se manda al simulador la velocidad a asignar a los actuadores del robot, que ha sido calculada previamente por la arquitectura. La última función a la que se llama en el bucle principal es la que indica al simulador que dé un paso de la ejecución.

```
/* Si hay choque, informar para tratar de recuperarse */
if(robot.getBumpers().getValor()!=0)
        robot.getArquitectura().getControlador().informarChoque();

/* Asignación de velocidad a las ruedas y ejecución de un paso de la simulación */
differential_wheels_set_speed(robot.getMotor().getRuedaIzda(), robot.getMotor().getRuedaDcha());
robot_step(64);
```

Por último, y antes de cerrar el bucle de ejecución, se recomienda que se suspenda la ejecución del hilo principal durante 100ms para asegurar la ejecución de los hilos de la arquitectura entre cada iteración de este bucle. Con estas últimas sentencias, el controlador para el simulador está terminado.

```
                        try {
                                Thread.sleep(100);
                        } catch (InterruptedException e) {
                                System.out.println("ERROR del controlador mientras dormía");
                        }
                } //Fin del bucle
        }  //Fin del main
} //Fin de la clase
```

3.2.2. Entorno real

Para crear un controlador que será ejecutado en entorno real utilizando el robot Pioneer 3-DX, también es necesario incluir las librerías que permiten hacer uso de las primitivas de la interfaz que acompaña al robot. Y al igual que para el controlador usado en simulación, también es necesario importar algunas clases de la arquitectura.

```
import com.mobilerobots.Aria.*
;import infraestructura.*;
import planificador.gestorPlanes.*;
```

En este caso la clase que implementa el controlador no tiene que heredar de ninguna otra, los atributos se limitan al número de sensores que posee el robot, pero sí es necesario cargar las librerías de ARIA antes de programar el método main().

```
public class pruebaAriaAvanzada {

        static final int NUM_SENSORES = 8;
        static final int NUM_BUMPERS = 5;

       /* Carga de librerías de ARIA */
       static{
              try {
                     System.loadLibrary("AriaJava");
              } catch (UnsatisfiedLinkError e) {
                     System.err.println("Native code library libAriaJava failed to
load. Make sure that its directory is in your library path; See
javaExamples/README.txt and the chapter on Dynamic Linking Problems in the
SWIG Java documentation (http://www.swig.org) for help.\n" + e);
                      System.exit(1);
              }
       }
```

A continuación ya sí se implementa el método main(), cuyas primeras instrucciones se corresponden con la inicialización de la conexión con el robot, algo que realiza ARIA de forma transparente al usuario gracias a los métodos proporcionados por su interfaz. También incluye la inicialización de los dispositivos que utilizará el robot Pioneer 3-DX, lo que permite habilitarlos. En este caso, al tratarse de la misma prueba, se inicializan los sensores sónar, bumpers y el motor, así como los datos medidos por odometría.

Las funciones de las librerías de ARIA más comunes se pueden observar en la siguiente figura, que contiene toda la inicialización necesaria para llevar a cabo la conexión con el robot. En caso de necesitar inicializar otros sensores, las funciones pueden consultarse en la propia documentación de ARIA.

```
/* Inicialización del robot real y el servidor ARIA */
Aria.init();
ArRobot robot = new ArRobot();
ArSimpleConnector conn = new ArSimpleConnector(args);
ArSonarDevice sonar = new ArSonarDevice();
ArBumpers bumpers = new ArBumpers();
if(!Aria.parseArgs()){
        Aria.logOptions();
        Aria.shutdown();
        System.exit(1);
}

 if (!conn.connectRobot(robot)){
        System.err.println("Could not connect to robot, exiting.\n");
        System.exit(1);
}
robot.runAsync(true);
robot.addRangeDevice(sonar);
robot.addRangeDevice(bumpers);
robot.enableMotors();
robot.requestEncoderPackets();
robot.resetTripOdometer();
```

Los siguientes bloques de código a introducir son exactamente iguales que para los controladores que se ejecutan en simulación, ya que incluyen la inicialización de la instancia del robot que maneja la arquitectura, así como la creación del plan y una primera lectura de los sensores del robot para evitar posteriores lecturas erróneas. Si es importante menciona que, en este caso, como la ejecución se llevará a cabo en el robot real y no en simulador, habrá que poner el argumento correspondiente en la inicialización del robot a “false”, justo al contrario de lo que se hizo en la creación del controlador para su uso en el simulador.

```
/* Inicialización de los elementos del robot */
Robot robot;
Sensor[] sensores_robot = new Sensor[2];
Actuador[] actuadores_robot = new Actuador[1];
Estado estado = new Estado(null, new Tarea("Inicio", null));
Sonar sonar = new Sonar(NUM_SENSORES);
sensores_robot[0] = sonar;
Bumpers bumpers = new Bumpers(NUM_BUMPERS, 0);
sensores_robot[2] = bumpers;
Motor motor = new Motor();
actuadores_robot[0] = motor;
robot = new Robot(estado, sensores_robot, actuadores_robot, false);
```

La siguiente figura muestra la creación del plan.

```
/* Creación de un plan para pasar al secuenciador */
Plan plan;
Meta meta_global = new Meta("DetectarAlgo", null);
LinkedList<Accion> acciones = new LinkedList<Accion>();
acciones.add(new Accion("MantenerTrayectoria"));
acciones.add(new Accion("EncontrarAlgo"));
plan = new Plan("PLAN4", meta_global, acciones);
robot.getArquitectura().getSecuenciador().setPlan(plan);
```

Y como último paso, previo a la ejecución del bucle principal, la primera lectura de los sensores, que en este caso se realiza mediante las funciones proporcionadas por ARIA.

```
/* Lectura inicial para descartar lecturas erróneas */
for(int i=0; i<NUM_SENSORES; i++){
        ArSensorReading asr = robot.getSonarReading(i);
        long val = asr.getRange();
        rob.getSonar().setValorSonar(i, val);
        }
ArUtil.sleep(1000);
```

Una vez hecho esto, el bucle principal debe contener siempre las mismas instrucciones que se describen a continuación, al igual que para el controlador ejecutado en simulador. La primera parte del bucle principal consiste en tomar los datos proporcionados por los sensores del robot real y almacenarlos en la instancia del robot que maneja la arquitectura. También aquí los valores de algunas variables y comparaciones se han obtenido mediante la experimentación con el simulador, y son diferentes a los valores utilizados para el controlador en el simulador debido a algunas diferencias que aparecen por el hecho de utilizar distintas plataformas cliente. Estas diferencias pueden consultarse en el capítulo 4.2 de la memoria del proyecto.

```
while(rob.getEstado().getTarea().getNombre().compareTo("Fin")!=0){

        /* Lectura de datos de odometría */
        rob.getMotor().setDistancia(robot.getOdometerDistance());
        rob.getMotor().setGiro(robot.getTripOdometerDegrees());

        /* Lectura de sónar */
        boolean cuidado = false;
        for(int i=0; i<NUM_SENSORES; i++){
                ArSensorReading asr = robot.getSonarReading(i);
                long val = asr.getRange();
                rob.getSonar().setValorSonar(i, val);
                if(i>0 && i<7)
                        if(rob.getSonar().getValorSonar(i)<350)
                                cuidado = true;
                }

        /* Lectura de bumpers */
        int bumps = robot.getStallValue();
        if(bumps!=0)
                rob.getBumpers().setValor(bumps);
        if(bumps==0 && !cuidado)
                rob.getBumpers().setValor(0);
```

Una vez leída la información de los sensores, se debe comprobar si se ha producido alguna colisión, pues es necesario informar de este hecho a la arquitectura para que realice las modificaciones necesarias en la agenda de tareas para continuar con la ejecución. Tras esta comprobación se manda al robot la velocidad a asignar a los actuadores del robot, que ha sido calculada previamente por la arquitectura. Por último, se suspende también el hilo principal durante 100ms para asegurar la ejecución de los hilos de la arquitectura entre cada iteración de este bucle.

```
                        /* Si hay choque, informar para tratar de recuperarse */
                        if(rob.getBumpers().getValor()!=0)
                                rob.getArquitectura().getControlador().informarChoque();

                        /* Asignación de velocidad a las ruedas */
                         double ri = (double) rob.getMotor().getRuedaIzda();
                        double rd = (double) rob.getMotor().getRuedaDcha();
                        robot.setVel2(ri, rd);

                        ArUtil.sleep(100);
                } //Fin del bucle
        } //Fin del main
} //Fin de la clase
```

4. Extensibilidad de la arquitectura

Como se ha comentado en la introducción de este manual, la arquitectura híbrida desarrollada tiene como objetivos asentar las bases de un marco de trabajo extensible y modular, que permita incluir nuevas funcionalidades en el futuro. Por ello, no toda la funcionalidad posible de la arquitectura ha sido implementada, pero sí tenida en cuenta para facilitar el trabajo al futuro desarrollador.

En este apartado se explican las tareas más importantes que se deben llevar a cabo para extender la arquitectura en diversos ámbitos o módulos, tales como las habilidades, los elementos del robot o el modelo del mundo.

4.1. Cómo incluir nuevos elementos del robot

La arquitectura permite incluir nuevos elementos y dispositivos relacionados con el robot de manera sencilla y eficaz. Se han dejado implementados algunos ejemplos en el propio código de la arquitectura en forma de clases abstractas, como *Bateria* y *Laser*.

Si se desea incluir un nuevo tipo de sensor, por ejemplo, el primer paso es crear una clase que herede de Sensor, e implementar todos los métodos relacionados con este nuevo tipo de sensor. Algunos métodos clave que deberían implementarse son los métodos de acceso (getters y setters) de la información con la que trabaja el sensor, para permitir que la instancia del robot que maneja la arquitectura pueda almacenar la información que recibe del sensor del robot real o simulado. Una vez hecho esto, la clase del robot (*src/infraestructura/Robot.java*) deberá ser modificada para incluir un nuevo atributo del tipo del nuevo sensor, y reflejarlo en los métodos que permiten acceder a los sensores del robot (habría que incluir las líneas marcadas en rojo).

```
public void setSensor(String id, int indice, int valor){
        if(id.compareTo("sonar")==0)
                this.sonar.setValorSonar(indice, valor);
        else if(id.compareTo("laser")==0)
                this.laser.setInfo(valor);
        else if(id.compareTo("bumpers")==0)
                this.bumpers.setValor(valor);
        else if(id.compareTo("nuevoSensor")==0)
                this.nuevoSensor.setValor(valor);
        else
                System.out.println("ERROR: Tipo de sensor " + id + " desconocido.");
}
```

```
public Sensor getSensor(String id){
        Sensor sensor = null;
        if(id.compareTo(this.sonar.getId())==0)
                sensor = this.sonar;
        else if(id.compareTo(this.laser.getId())==0)
                sensor = this.laser;
        else if(id.compareTo(this.bumpers.getId())==0)
                sensor = this.bumpers;
        else if(id.compareTo(this.nuevoSensor.getId())==0)
                sensor = this.nuevoSensor;
        else
                System.out.println("ERROR: El identificador " + id + " no se corresponde
con ningún sensor.");
        return sensor;
}
```

Para incluir un nuevo actuador habría que seguir los mismos pasos, pero con las clases y métodos correspondientes a los actuadores del robot. Por otro lado, se puede incluir cualquier otro elemento o dispositivo del robot que se pretenda utilizar en la ejecución, sin necesidad de ser un sensor o actuador propiamente dichos. Un ejemplo de otro elemento cualquier es la batería. Tal y como se ha dejado indicado en el código de la arquitectura, para incluir este tipo de elementos, basta con crear la clase deseada e incluir un atributo de dicha clase en la clase *Robot*.

4.2. Cómo incluir nuevas habilidades

La inclusión de nuevas habilidades o módulos de habilidades también resulta muy sencilla gracias al diseño del subsistema controlador de la arquitectura. Este subsistema se compone de un gestor de habilidades y tantos módulos de habilidades como diferentes tipos de éstas se quieran utilizar.

Para incluir una nueva habilidad en un módulo ya definido, se va a utilizar como ejemplo la clase abstracta *IrA*, una nueva habilidad del módulo de navegación. Los pasos a seguir para incluir una nueva habilidad son los siguientes: crear una clase en el componente adecuado (en este caso *Navegacion*) que represente la habilidad. Esta clase deberá heredar de la clase *Habilidad*, por lo que obligatoriamente deberá implementar los métodos *ejecutar()* y *comprobarExito()*. Si es necesario, cada uno de estos métodos pueden dividirse en dos o más funciones diferentes, dependiendo de la plataforma donde se ejecutará la habilidad. Para ver un ejemplo de esto, la clase *EvitarObstaculo*

contiene ambos métodos divididos en otros dos (para simulador y entorno real). Una vez hecho esto, es necesario incluir la nueva habilidad en la librería de habilidades, que es inicializada en el constructor de la clase Habilidades (*src/controlador/gestorHabilidades/Habilidades.java*). Habrá que añadir una línea como la que se propone en el ejemplo:

```
public Habilidades(boolean todas, Robot robot, boolean sim){
        this.habilidades = new HashMap<String, Habilidad>();
        if(todas){
                this.habilidades.put("SeguirPared", new SeguirPared(true, robot, sim));
                this.habilidades.put("Avanzar", new Avanzar(robot, sim));
                this.habilidades.put("Girar", new Girar(robot, sim, false, 4, 90));
                ...
                this.habilidades.put("IrA", new IrA(robot, sim));
                this.habilidades.put("Esperar", new Esperar(robot, sim));
                this.habilidades.put("Finalizar", new Finalizar(robot, sim));
        }
}
```

Pero sólo esto no es suficiente para que la arquitectura pueda hacer uso de esta nueva habilidad. La inclusión de una nueva habilidad va asociada a la inclusión de nuevas funcionalidades del robot, con nuevas acciones y tareas que llevar a cabo. Es decir, siempre irá ligada a una extensión en el dominio del planificador y del secuenciador. Cuando se realice esta extensión, habrá que tener en cuenta las nuevas habilidades incorporadas para incluirlas en RAPs asociados, de forma que se incluyan en la librería de RAPs (*src/secuenciador/memoria/LibreriaRAPs.java*):

```
public LibreriaRAPs(Metas libreria){
        this.lista = new HashMap<String, RAP>();

        String hab1[] = {"Girar", "Avanzar"};
        this.lista.put("RAP1", new RAP("RAP1", libreria.getMeta("PuntoEnfrente"), hab1));

        String hab2[] = {"Avanzar", "Girar", "IrA", "SeguirPared"};
        this.lista.put("RAP2", new RAP("RAP2", libreria.getMeta("LlegarA"), hab2));

        ...

        String habN[] = {"NuevaHabilidad", "Avanzar", "OtraHabilidadAntigua"};
        this.lista.put("RAPN", new RAP("RAPN", libreria.getMeta("NuevaMeta"), habN));
}
```

Si lo que se desea es incluir un nuevo módulo completo de habilidades, como por ejemplo habilidades relacionadas con la visión, el proceso es similar al de incluir

una nueva habilidad. En primer lugar es necesario crear un nuevo componente en el subsistema controlador (como ejemplo se proporciona el componente Vision) e incluir en él una clase para cada una de las habilidades que formarán este módulo, tal y como se ha explicado para la inclusión de habilidades únicas. Una vez hecho esto, también es necesario incluir las nuevas habilidades en la librería que se inicializa en la clase *Habilidades*.

Una vez más, no tiene sentido incluir un nuevo módulo de habilidades si éstas no van asociadas a una nueva funcionalidad del robot, por lo que habrá que incluir nuevos elementos en los dominios del planificador y secuenciador (ver apartado 4.4), que estarán asociados a este nuevo módulo de habilidades, y que por tanto deberán verse éstas reflejadas en la librería de RAPs del sistema.

4.3. Cómo incluir nuevos elementos en el modelo del mundo

También es posible incluir nuevos elementos a tener en cuenta en el modelo del mundo, para poder crear problemas más elaborados que el robot pueda llegar a resolver. Estos nuevos elementos a incluir en el modelo del mundo pueden ser de dos tipos: elementos del robot que juegan un papel fundamental en el problema, o bien elementos propios del entorno en el que se mueve el robot.

Como ejemplo de los primeros se encuentra la batería del robot. Se trata de un elemento del robot, el cual ya se ha explicado anteriormente cómo incluirlo en la arquitectura, pero que además puede formar parte del modelo del mundo entre los elementos del entorno a tener en cuenta para resolver un problema. Para incluirlo también en este módulo, es necesario que el estado del robot refleje este elemento, bien manteniendo una copia de la instancia, bien almacenando cierta información, como puede ser el nivel de batería. En cualquier caso, se deberá modificar la clase *Estado* (*src/infraestructura/Estado.java*) para incluir los atributos y métodos de acceso necesarios para que otros elementos de la arquitectura lo tengan en cuenta. Adicionalmente, puede ser necesario que la clase *Mapa* (*src/planificador/modeloMundo/Mapa.java*) que se utilice en el planificador mantenga una referencia a la localización de algún elemento relacionado con este otro, como pueden ser las coordenadas de la base de carga de la batería.

En cuanto a incluir nuevos elementos del propio entorno, nos estamos refiriendo a la forma de trabajar con el entorno del robot, como por ejemplo su representación. Aunque no está implementada la capa superior de la arquitectura, sí se han incluido algunas clases y ejemplos de cómo hacerlo. Entre las aportaciones al modelo del mundo se presenta la capacidad de trabajar con distintos tipos de mapas del entorno. A partir de la superclase *Mapa*, se pueden definir tantas subclases como tipos de mapas se deseen utilizar para la localización y posicionamiento del robot y otros elementos del entorno. Como ejemplo se dan dos tipos de mapas: de coordenadas y de movimientos.

Cualquier tipo de mapa se recomienda que se inicialice en el sistema mediante un fichero de configuración que contenga toda la información del mapa (posiciones de elementos y vectores que describan el entorno en el caso de mapas de coordenadas; localizaciones y conexiones entre éstas en el caso de mapas de movimientos). Se aporta un ejemplo de fichero de configuración en la carpeta raíz de la arquitectura híbrida (*map.txt*) La forma de implementar todos estos elementos del modelo del mundo es totalmente libre, pero deben estar recogidos en el componente *ModeloMundo* y el planificador deberá trabajar con esta información para elaborar planes de alto nivel, que posteriormente permitan al secuenciador y controlador transformar en tareas que el robot pueda llevar a cabo en ese modelo del mundo que se haya definido.

4.4. Cómo sustituir e incluir nuevos módulos

Por último, comentar que tanto el planificador como el secuenciador se han diseñado a partir de conceptos estudiados durante la carrera o durante la documentación del presente proyecto. En el caso del planificador, se diseñó uno basado en la utilización de algún algoritmo de búsqueda clásico, que finalmente no se implementó, con sus correspondientes acciones, predicados y elementos de programación lógica. Por su parte, el secuenciador se diseñó basándose en una simplificación propia de los Reactive Action Package (RAP) desarrollados por otros autores.

Esta implementación puede ser modificada o sustituida si se desea. Este proceso es más complicado que las modificaciones que se han explicado hasta ahora en este apartado, pues es necesario diseñar la nueva forma de planificar o secuenciar, pero la

integración con la arquitectura no debería ser tan compleja. Simplemente se debe tener en cuenta que, para ambas capas (secuenciador y planificador), una buena parte del diseño no es necesario modificar. En el caso del secuenciador, el componente *Interprete* debería mantenerse tal y como está diseñado e implementado. La clase Secuenciacion es la encargada de transformar el plan elaborado previamente en una secuencia de tareas, y de proporcionar una solución al controlador mediante los RAPs. Al sustituir o incluir un nuevo método de secuenciación, simplemente habría que implementarlo aparte y que fuera esta clase (*Secuenciacion*) la que se comunicara con el nuevo módulo e incluyera las operaciones necesarias para crear la secuencia de tareas utilizando la funcionalidad de éste.

Lo mismo ocurre con el planificador, del cual no debería modificarse el componente *ModeloMundo*, y sólo parte del gestor de planes: aquella relacionada con el módulo de planificación diseñado. En este caso es la clase *Planificacion* la encargada de elaborar el plan haciendo uso de los elementos y funciones diseñados para utilizar un algoritmo de búsqueda clásico. Si se desea utilizar un planificador ya desarrollado, o implementar uno nuevo, lo único que hay que hacer es integrar esta clase con el planificador, para que puedan comunicarse y adaptar los resultados obtenidos por el planificador a los tipos de datos e información que utiliza la arquitectura híbrida.

Por otro lado, también entra en este apartado la modificación de los módulos del secuenciador y planificador que permiten dotar al robot de nuevas funcionalidades. Es decir, ampliar los dominios de planificación y secuenciación para incorporar nuevas acciones, tareas, metas y RAPs, que junto con nuevas habilidades, crearán planes más elaborados que permitirán al robot alcanzar nuevos objetivos y realizar mayor cantidad de tareas.

Para ampliar estos dominios simplemente es necesario diseñar bien cómo se va a trabajar con las nuevas tareas, metas y demás elementos. Es decir, definir correctamente qué nuevos objetivos puede alcanzar el robot (metas), qué nuevas acciones deberá llevar a cabo para conseguir estos objetivos (acciones del planificador, tareas del secuenciador) y cómo asociar estas nuevas tareas a las habilidades que posee el robot (RAPs). Se recomienda echar un vistazo a las tablas que muestran las librerías de todos estos elementos de la arquitectura en el documento principal de la memoria del proyecto

(capítulo 3.5) para ver un ejemplo de cómo definir estos dominios. Y a la hora de implementarlos, basta con incluir todos los nuevos elementos definidos en las librerías del sistema, de las que se muestra un ejemplo a continuación:

- Librería de acciones, predicados y elementos (planificador), que se encuentran en la clase *Planificacion* (*src/planificador/gestorPlanes/Planificacion.java*), creadas a partir de un fichero de configuración del dominio del planificador. Se aporta un ejemplo en el archivo raíz de la arquitectura híbrida (*domain.txt*).

- Librería de metas, que se encuentra en la clase *Metas* (*src/planificador/gestorPlanes/Metas.java*):

```
public Metas(){
        this.metas = new HashMap<String, Meta>();
        this.metas.put("LlegarA", new Meta("LlegarA", new Predicado()));
        this.metas.put("DetectarAlgo", new Meta("DetectarAlgo", new Predicado()));
        this.metas.put("PuntoEnfrente", new Meta("PuntoEnfrente", new Predicado()));
        ...
        this.metas.put("NuevaMeta", new Meta("NuevaMeta", new Predicado("NuevoPredicado")));
}
```

- Librería de tareas, que se encuentra en la clase *Tareas* (*src/secuenciador/interprete/Tareas.java*):

```
public Tareas(Metas libreria){
        this.tareas = new HashMap<Meta, Tarea>();
        this.tareas.put(libreria.getMeta("LlegarA"), new Tarea("MantenerDireccion",
                                libreria.getMeta("LlegarA")));
        this.tareas.put(libreria.getMeta("DetectarAlgo"), new Tarea("EncontrarAlgo",
                                libreria.getMeta("DetectarAlgo")));
        ...
        this.tareas.put(libreria.getMeta("NuevaMeta"), new Tarea("NuevaTarea",
                                libreria.getMeta("NuevaMeta")));
}
```

- Librería de RAPs, que se encuentra en la clase *LibreriaRAPs* (*src/secuenciador/memoria/LibreriaRAPs.java*):

```
public LibreriaRAPs(Metas libreria){
        this.lista = new HashMap<String, RAP>();

        String hab1[] = {"Girar", "Avanzar"};
        this.lista.put("RAP1", new RAP("RAP1", libreria.getMeta("PuntoEnfrente"), hab1));

        String hab2[] = {"Avanzar", "Girar", "IrA", "SeguirPared"};
        this.lista.put("RAP2", new RAP("RAP2", libreria.getMeta("LlegarA"), hab2));

        ...

        String habN[] = {"NuevaHabilidad", "Habilidad1", “HabilidadN”};
        this.lista.put("RAPN", new RAP("RAPN", libreria.getMeta("NuevaMeta"), habN));
}
```

5. Resumen

Para facilitar al usuario la consulta sobre la creación de un controlador para ejecutar la arquitectura híbrida en un simulador, o bien en un entorno real con el robot Pioneer 3-DX, se presenta esta guía de referencia rápida con un esquema de las operaciones a llevar a cabo para implementar el controlador. Para más detalle o consulta de las funciones concretas a utilizar, ya sean parte del subsistema de infraestructura de la propia arquitectura, de las librerías del simulador Webots, o de las librerías de la interfaz ARIA para el robot, se remite al capítulo 3.2 del presente documento.

Los pasos a seguir para implementar un nuevo controlador que haga uso de la arquitectura híbrida son los siguientes:

- Importar las librerías necesarias (Webots o ARIA) y los paquetes de la arquitectura (infraestructura obligatorio, gestor de planes opcional).

- Declarar los atributos del controlador, que serán el número de sensores de cada tipo que se usarán, y los arrays de almacenamiento de datos sólo en el caso del simulador.

- Cargar las librerías de ARIA (sólo en el caso del robot real).

- Inicializar los dispositivos de los que hará uso el robot.
    - En caso de utilizar el simulador, incluir la inicialización en una función llamada *reset*.
    - En caso de utilizar el robot real, incluir la inicialización al comienzo del método *main*.

- Inicializar la instancia del robot que maneja la arquitectura, junto con todos los elementos de los que hará uso.

- Crear un plan predefinido en el controlador y asignárselo a la instancia del robot (sólo en caso de no trabajar con un planificador implementado como capa superior de la arquitectura).

- Realizar una lectura inicial de la información ofrecida por los sensores para evitar posteriores lecturas erróneas.

- Implementar el bucle principal, que se ejecutará hasta que el robot haya llevado a cabo el plan completo y se indique mediante una tarea especial de gestión de la finalización de la ejecución. El bucle consiste en las siguientes operaciones:
    - Lectura de todos los tipos de sensores de los que se vaya a hacer uso.
    - Comprobación de una posible colisión del robot con algún obstáculo, y su correspondiente notificación.
    - Asignación de las velocidades calculadas por la arquitectura a los actuadores del robot.
    - Ejecución de un paso de la simulación (sólo en el caso del simulador).
    - Suspensión de la ejecución del controlador durante un tiempo no superior a 100ms.

Anexo B

Presupuesto

1.- Autor: José Luis Díaz Cebrián

2.- Departamento: Informática

3.- Descripción del Proyecto:

- Titulo	Implementación de una arquitectura de control híbrida para robots autónomos
- Duración (meses)	6
Tasa de costes Indirectos:	20%

4.- Presupuesto total del Proyecto (valores en Euros):

15.524,25 Euros

5.- Desglose presupuestario (costes directos)

PERSONAL

Apellidos y nombre	N.I.F. (no rellenar - solo a título informativo)	Categoría	Dedicación (hombres mes) [a]	Coste hombre mes	Coste (Euro)	Firma de conformidad
Díaz Cebrián, José Luis		Ingeniero	4,57	2.694,39	12.313,36	
		Ingeniero Senior		4.289,54	0,00	
		Ingeniero		2.694,39	0,00	
					0,00	
					0,00	
		Hombres mes	**4,57**	**Total**	12.313,36	

[a] 1 Hombre mes = 131,25 horas. Máximo anual de dedicación de 12 hombres mes (1575 horas)
Máximo anual para PDI de la Universidad Carlos III de Madrid de 8,8 hombres mes (1.155 horas)

EQUIPOS

Descripción	Coste (Euro)	% Uso dedicado proyecto	Dedicación (meses)	Periodo de depreciación	Coste imputable [d]
Portátil Acer Aspire TimeLineX	639,00	100	6	60	63,90
MobileRobots Pioneer P3-DX	2.651,58	100	6	60	265,16
Licencia USB simulador Webots	2.944,57	100	6	60	294,46
		100		60	0,00
		100		60	0,00
					0,00
				Total	623,52

[d] Fórmula de cálculo de la Amortización:

$$\frac{A}{B} x C x D$$

A = nº de meses desde la fecha de facturación en que el equipo es utilizado
B = periodo de depreciación (60 meses)
C = coste del equipo (sin IVA)
D = % del uso que se dedica al proyecto (habitualmente 100%)

SUBCONTRATACIÓN DE TAREAS

Descripción	Empresa	Coste imputable
	Total	0,00

OTROS COSTES DIRECTOS DEL PROYECTO[e]

Descripción	Empresa	Costes imputable
	Total	0,00

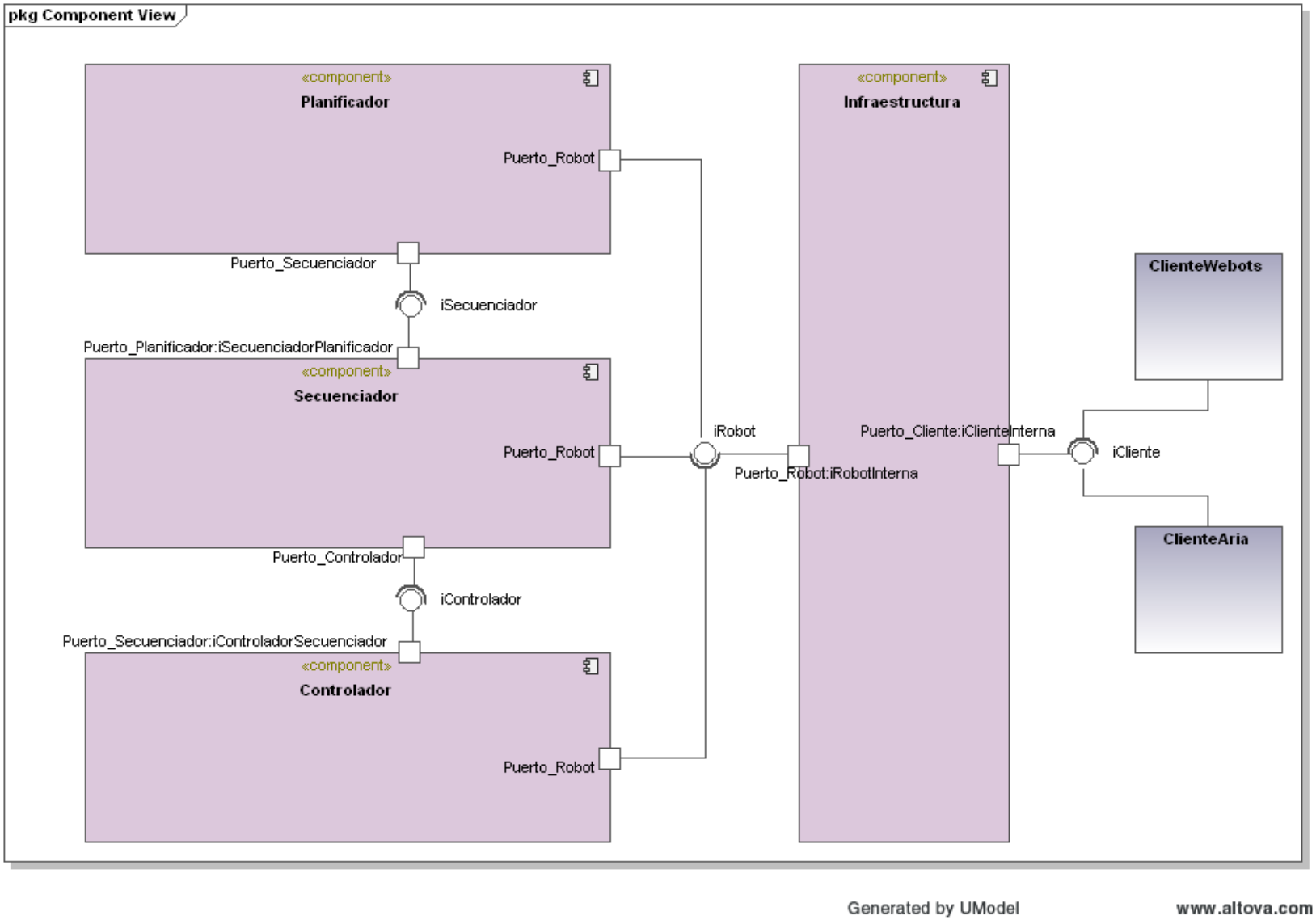
[e] Este capítulo de gastos incluye todos los gastos no contemplados en los conceptos anteriores, por ejemplo: fungible, viajes y

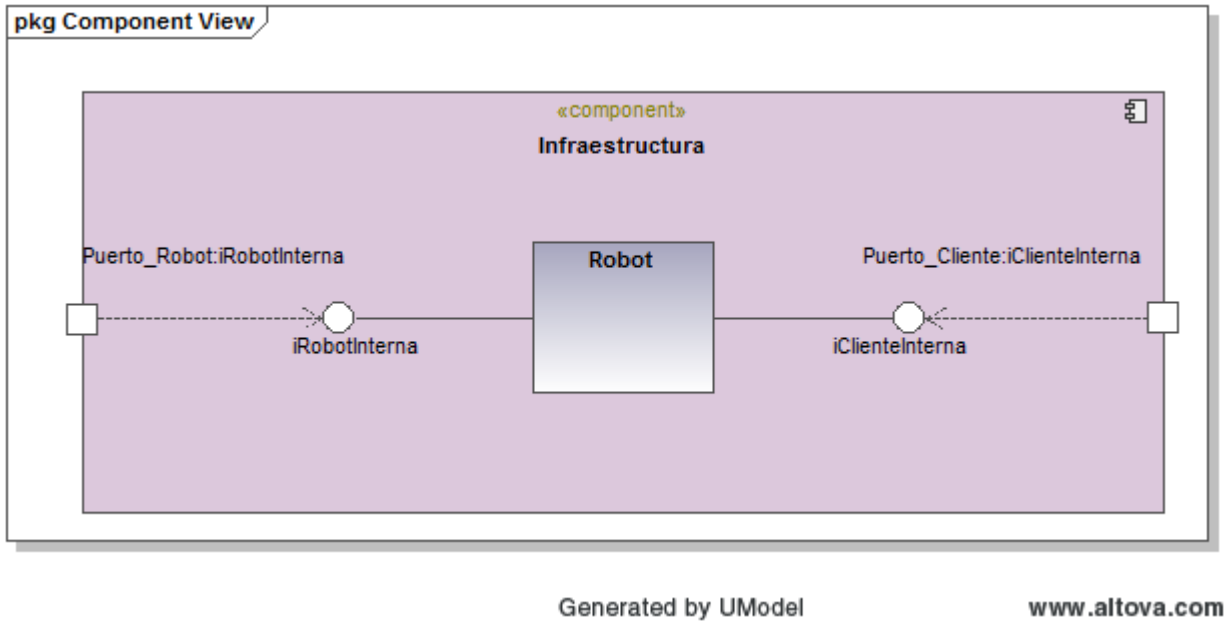
6.- Resumen de costes

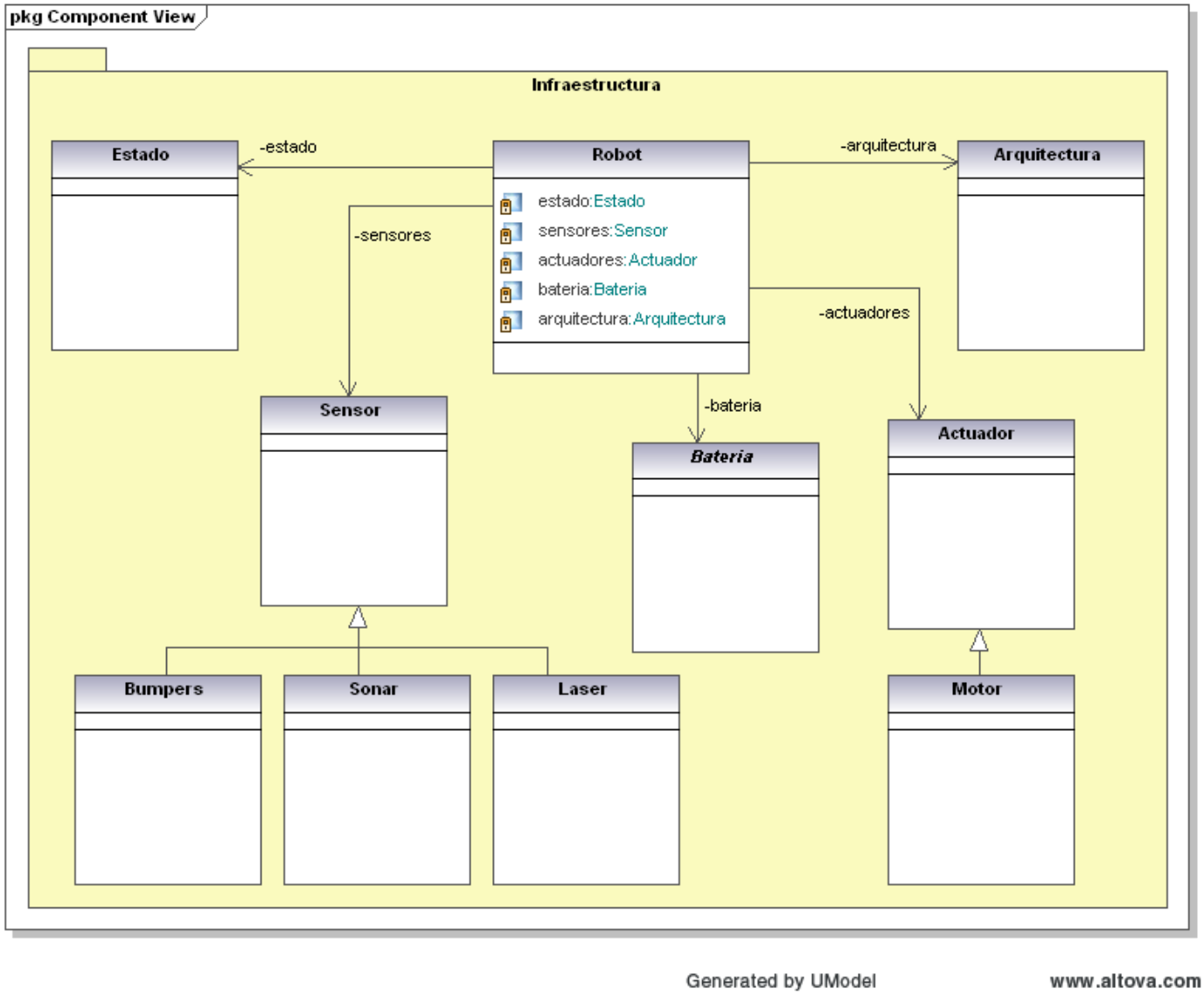
Presupuesto Costes Totales	**Presupuesto Costes Totales**
Personal	12.313
Amortización	624
Subcontratación de tareas	0
Costes de funcionamiento	0
Costes Indirectos	2.587
Total	15.524

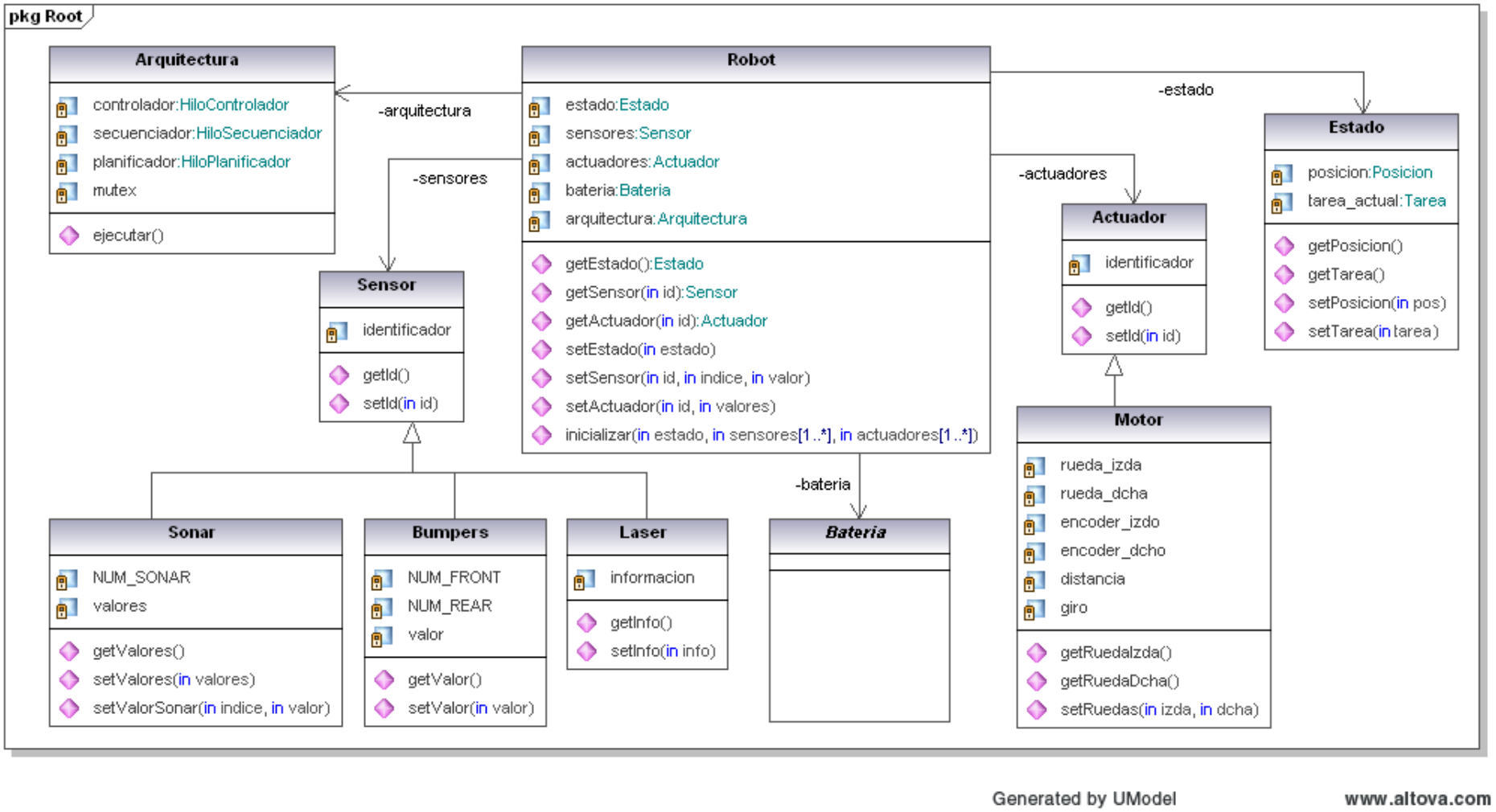
Anexo C

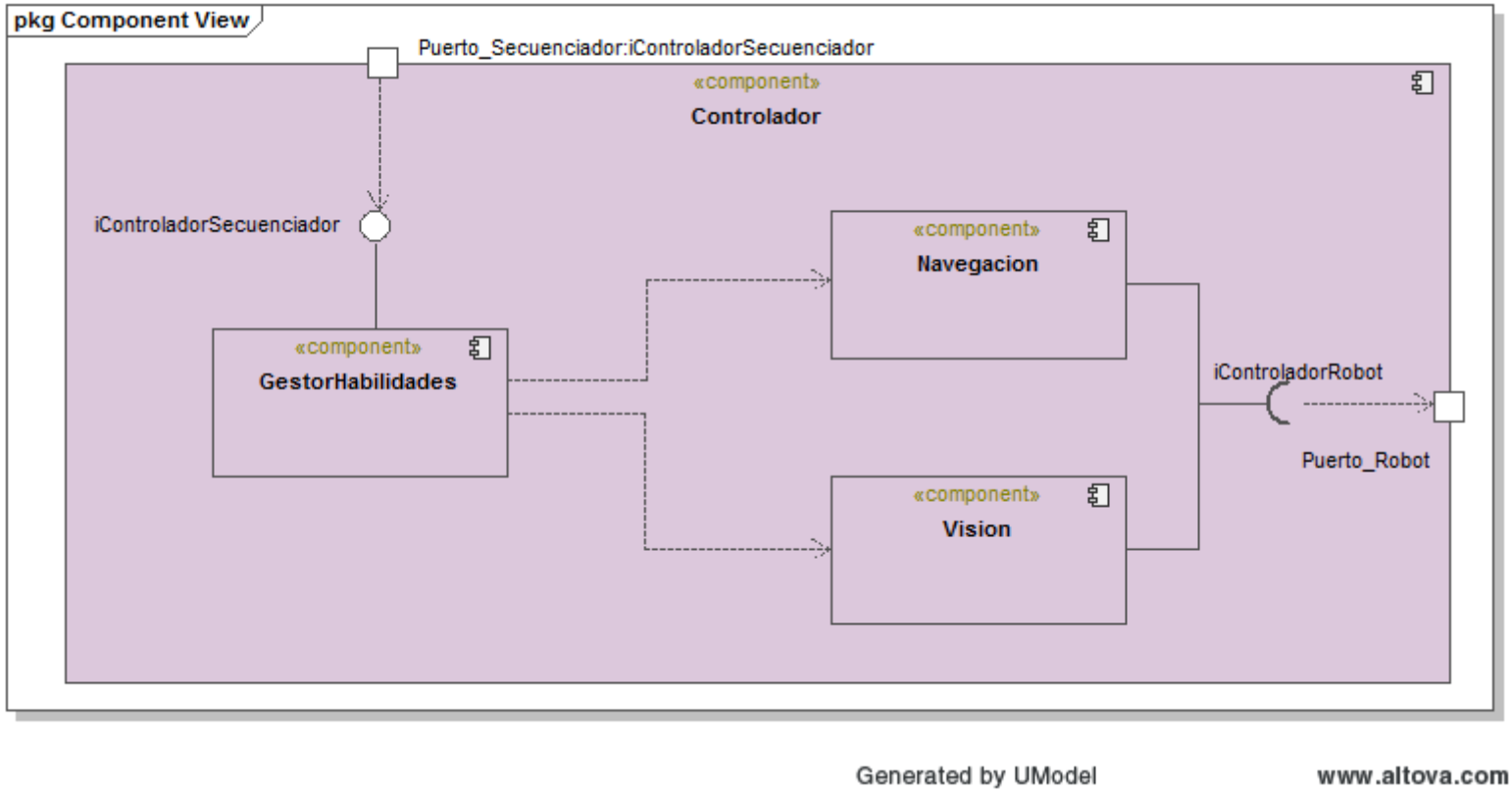
Diagramas de Diseño

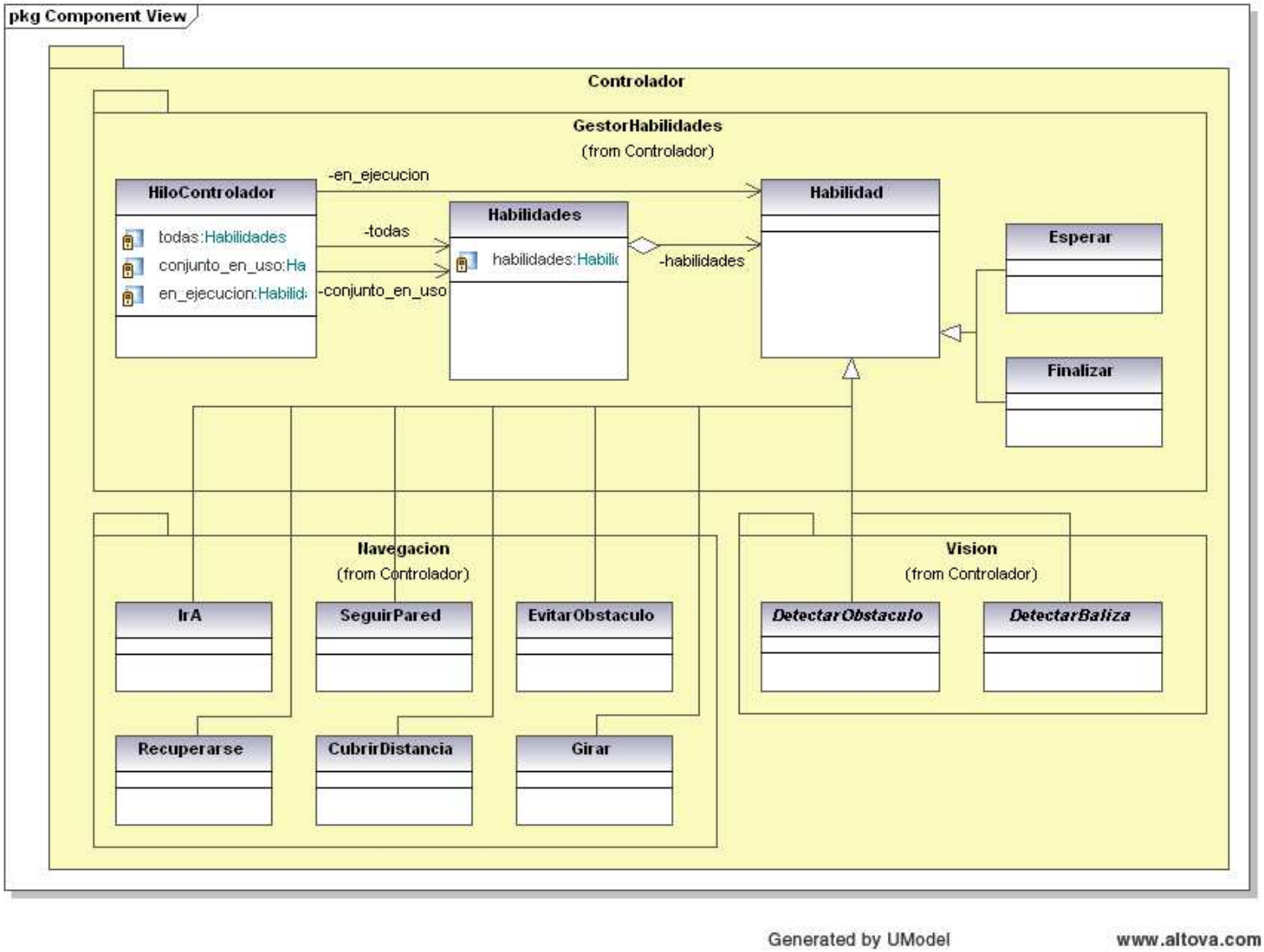

pkg Component View
«component»
Planificador
Puerto_Robot
Puerto_Secuenciador
iSecuenciador
Puerto_Planificador:iSecuenciadorPlanificador
«component»
Secuenciador
Puerto_Robot
Puerto_Controlador
iControlador
Puerto_Secuenciador:iControladorSecuenciador
«component»
Controlador
Puerto_Robot
iRobot
Puerto_Robot:iRobotInterna
«component»
Infraestructura
Puerto_Cliente:iClienteInterna
iCliente
ClienteWebots
ClienteAria
Generated by UModel
www.altova.com

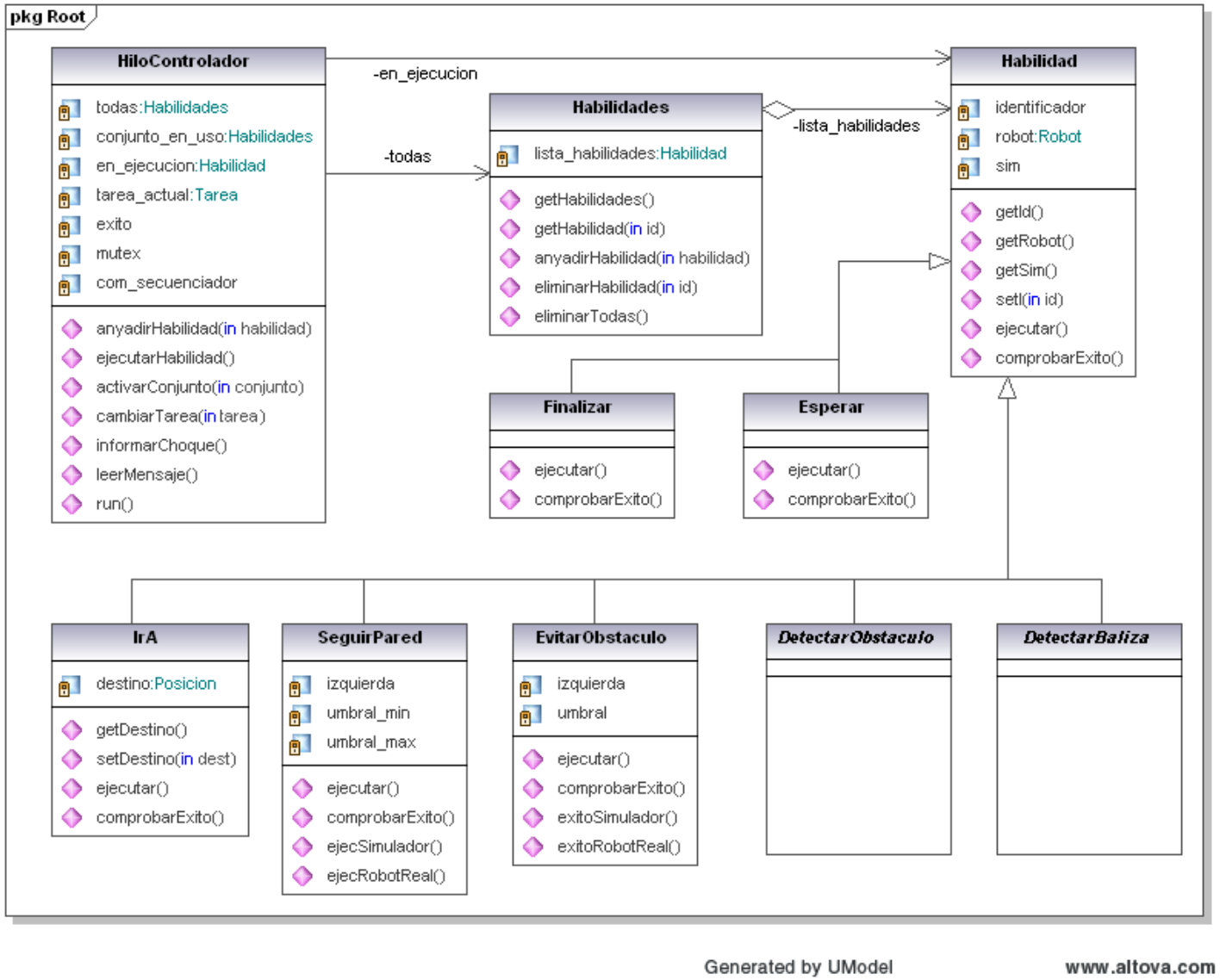

pkg Component View
«component»
Infraestructura
Puerto_Robot:iRobotInterna
Robot
Puerto_Cliente:iClienteInterna
iRobotInterna
iClienteInterna
Generated by UModel
www.altova.com

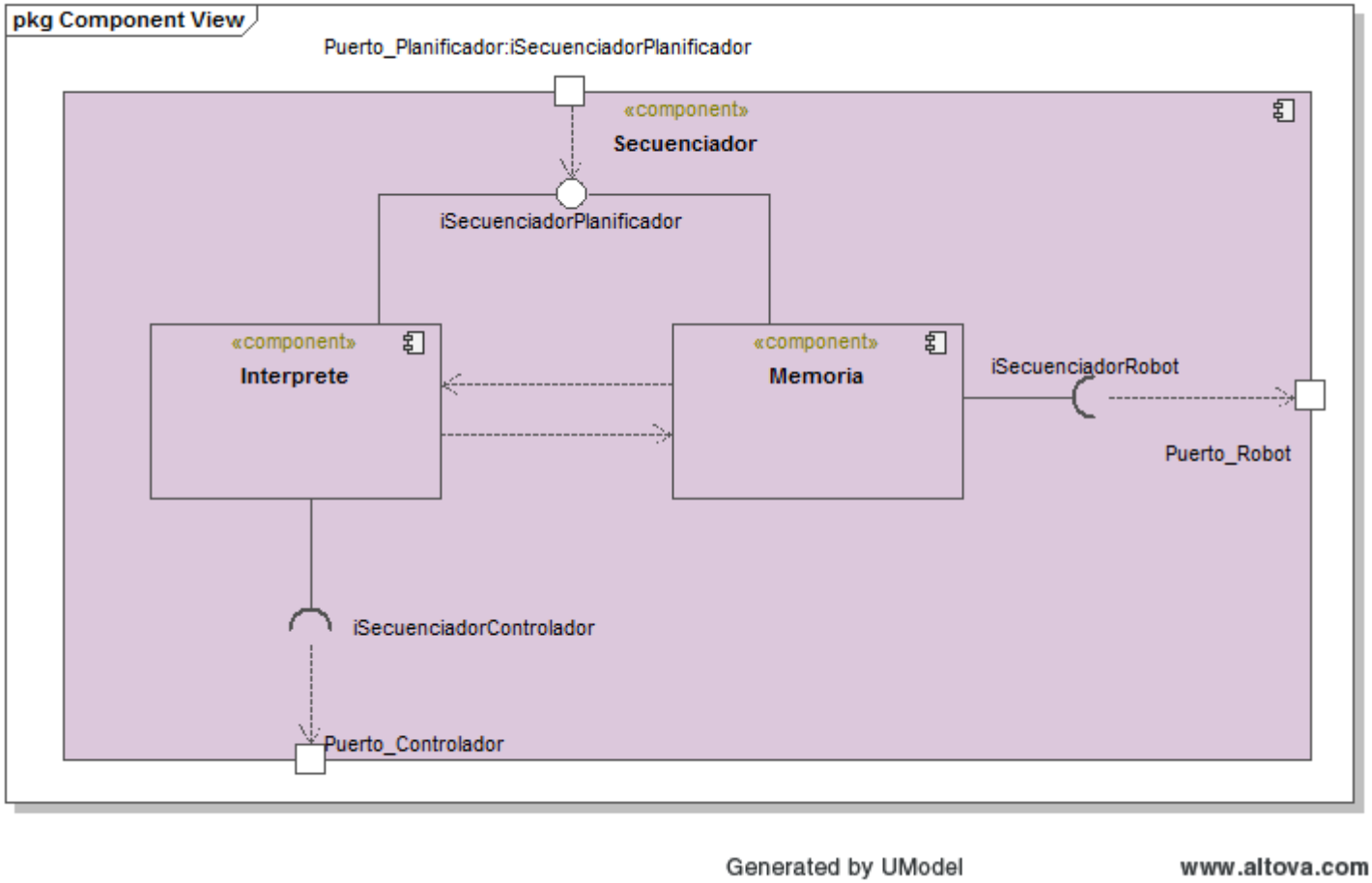

pkg Component View
Infraestructura
Estado
-estado
Robot
-arquitectura
Arquitectura
estado:Estado
sensores:Sensor
actuadores:Actuador
bateria:Bateria
arquitectura:Arquitectura
-sensores
-actuadores
-bateria
Sensor
Bateria
Actuador
Bumpers
Sonar
Laser
Motor
Generated by UModel
www.altova.com

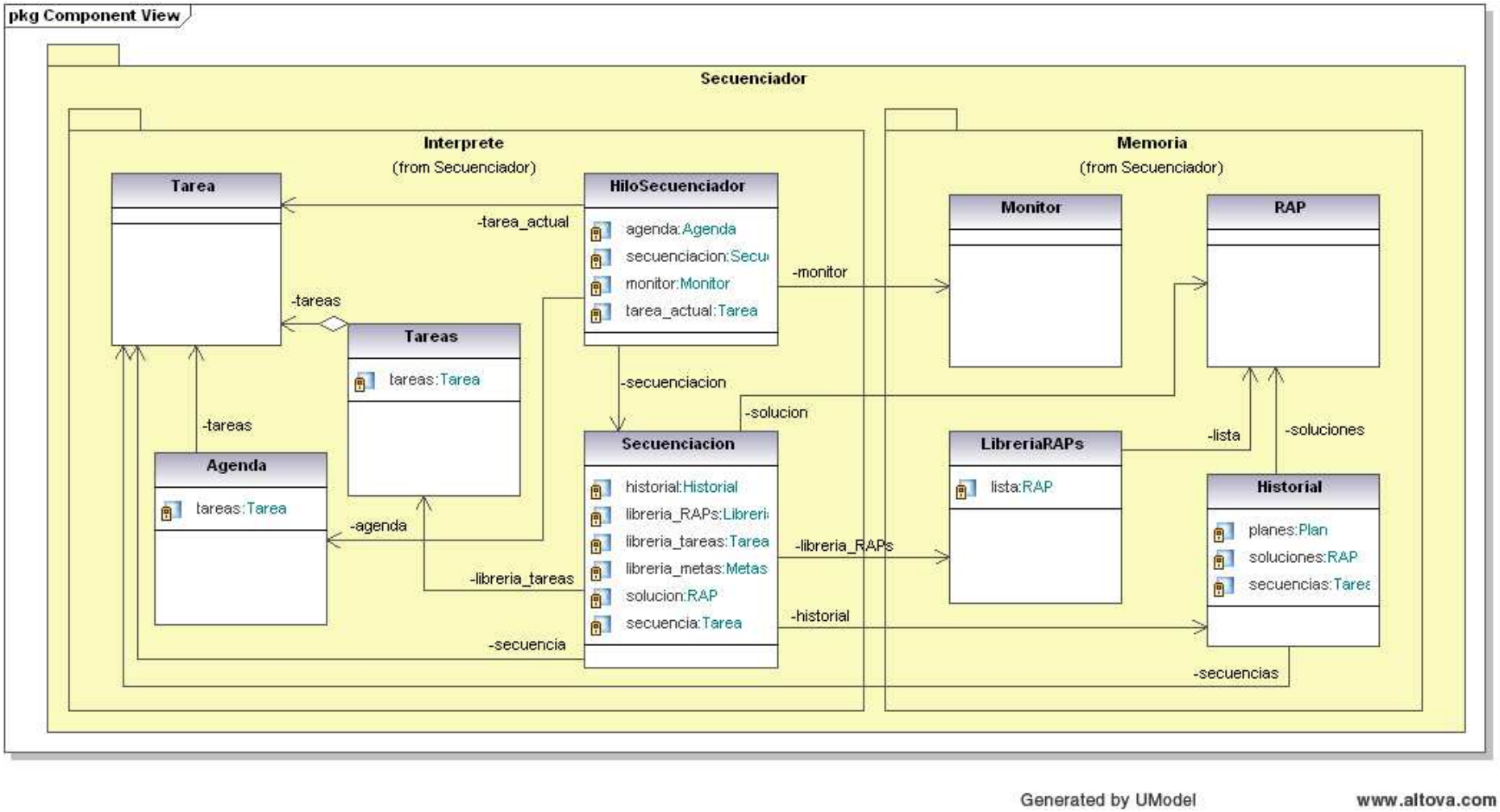

pkg Root
Arquitectura
controlador:HiloControlador
secuenciador:HiloSecuenciador
planificador:HiloPlanificador
mutex
ejecutar()
-arquitectura
-sensores
Robot
estado:Estado
sensores:Sensor
actuadores:Actuador
bateria:Bateria
arquitectura:Arquitectura
getEstado():Estado
getSensor(in id):Sensor
getActuador(in id):Actuador
setEstado(in estado)
setSensor(in id, in indice, in valor)
setActuador(in id, in valores)
inicializar(in estado, in sensores[1..*], in actuadores[1..*])
-estado
-actuadores
Estado
posicion:Posicion
tarea_actual:Tarea
getPosicion()
getTarea()
setPosicion(in pos)
setTarea(in tarea)
Actuador
identificador
getId()
setId(in id)
Sensor
identificador
getId()
setId(in id)
Motor
rueda_izda
rueda_dcha
encoder_izdo
encoder_dcho
distancia
giro
getRuedaIzda()
getRuedaDcha()
setRuedas(in izda, in dcha)
-bateria
Sonar
NUM_SONAR
valores
getValores()
setValores(in valores)
setValorSonar(in indice, in valor)
Bumpers
NUM_FRONT
NUM_REAR
valor
getValor()
setValor(in valor)
Laser
informacion
getInfo()
setInfo(in info)
Bateria
Generated by UModel
www.altova.com

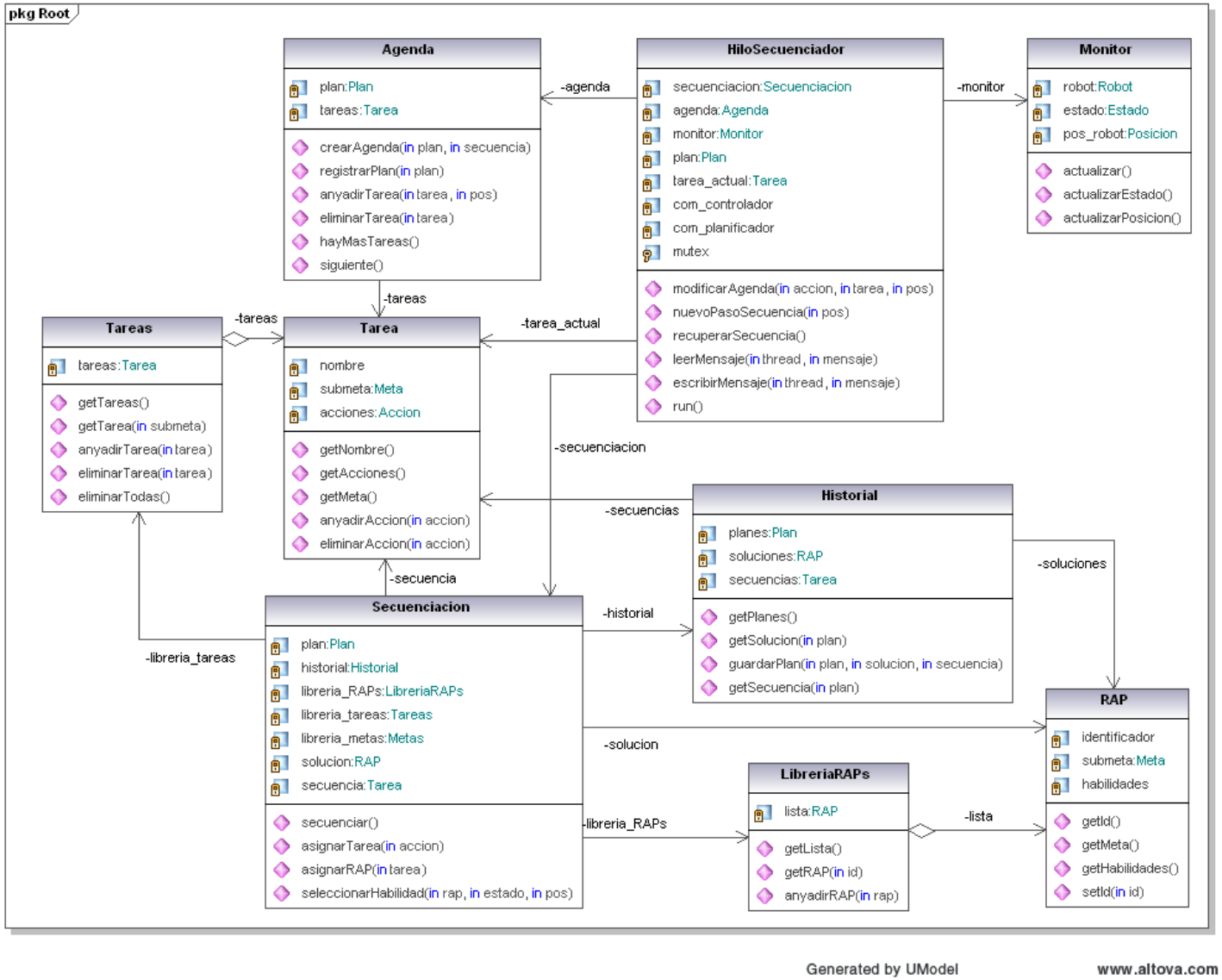

pkg Component View
Puerto_Secuenciador:iControladorSecuenciador
«component»
Controlador
iControladorSecuenciador
«component»
Navegacion
«component»
GestorHabilidades
iControladorRobot
Puerto_Robot
«component»
Vision
Generated by UModel
www.altova.com

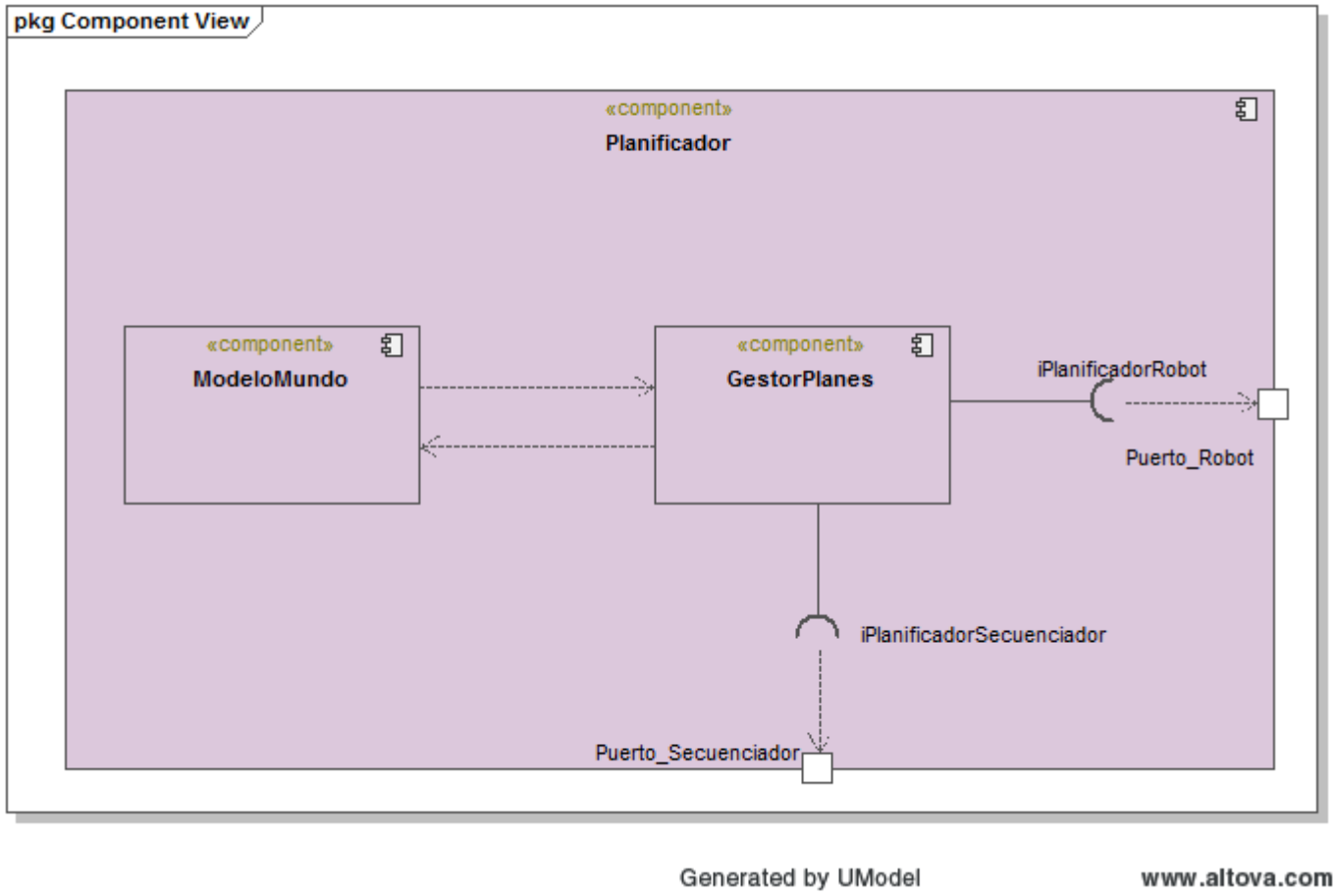

pkg Component View
Controlador
GestorHabilidades
(from Controlador)
HiloControlador
todas:Habilidades
conjunto_en_uso:Ha
en_ejecucion:Habilid
-en_ejecucion
-todas
-conjunto_en_uso
Habilidades
habilidades:Habili
-habilidades
Habilidad
Esperar
Finalizar
Navegacion
(from Controlador)
IrA
SeguirPared
EvitarObstaculo
Recuperarse
CubrirDistancia
Girar
Vision
(from Controlador)
DetectarObstaculo
DetectarBaliza
Generated by UModel
www.altova.com

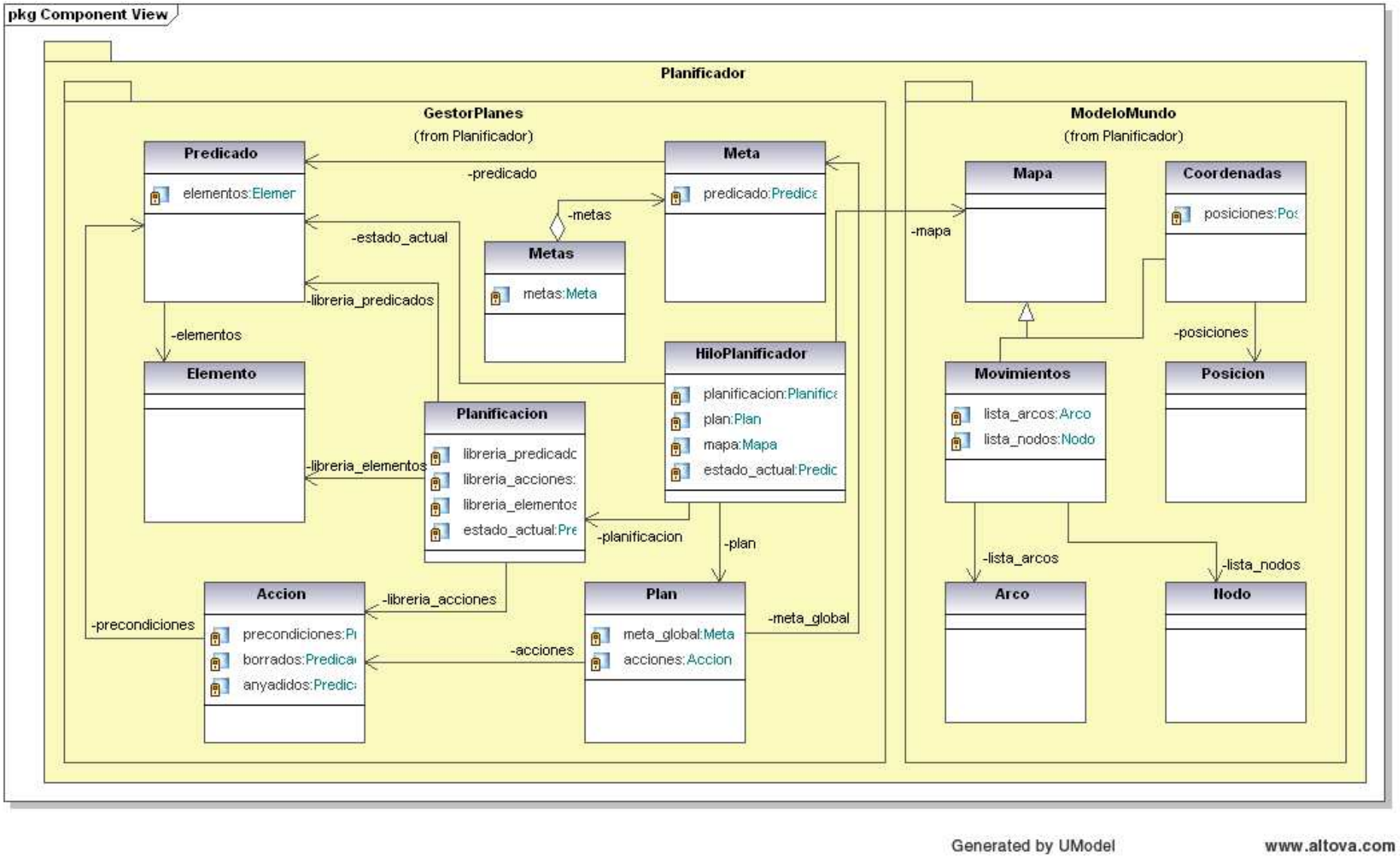

pkg Root
HiloControlador
todas:Habilidades
conjunto_en_uso:Habilidades
en_ejecucion:Habilidad
tarea_actual:Tarea
exito
mutex
com_secuenciador
anyadirHabilidad(in habilidad)
ejecutarHabilidad()
activarConjunto(in conjunto)
cambiarTarea(in tarea)
informarChoque()
leerMensaje()
run()
-en_ejecucion
-todas
Habilidades
lista_habilidades:Habilidad
getHabilidades()
getHabilidad(in id)
anyadirHabilidad(in habilidad)
eliminarHabilidad(in id)
eliminarTodas()
-lista_habilidades
Habilidad
identificador
robot:Robot
sim
getId()
getRobot()
getSim()
setI(in id)
ejecutar()
comprobarExito()
Finalizar
ejecutar()
comprobarExito()
Esperar
ejecutar()
comprobarExito()
IrA
destino:Posicion
getDestino()
setDestino(in dest)
ejecutar()
comprobarExito()
SeguirPared
izquierda
umbral_min
umbral_max
ejecutar()
comprobarExito()
ejecSimulador()
ejecRobotReal()
EvitarObstaculo
izquierda
umbral
ejecutar()
comprobarExito()
exitoSimulador()
exitoRobotReal()
DetectarObstaculo
DetectarBaliza
Generated by UModel
www.altova.com

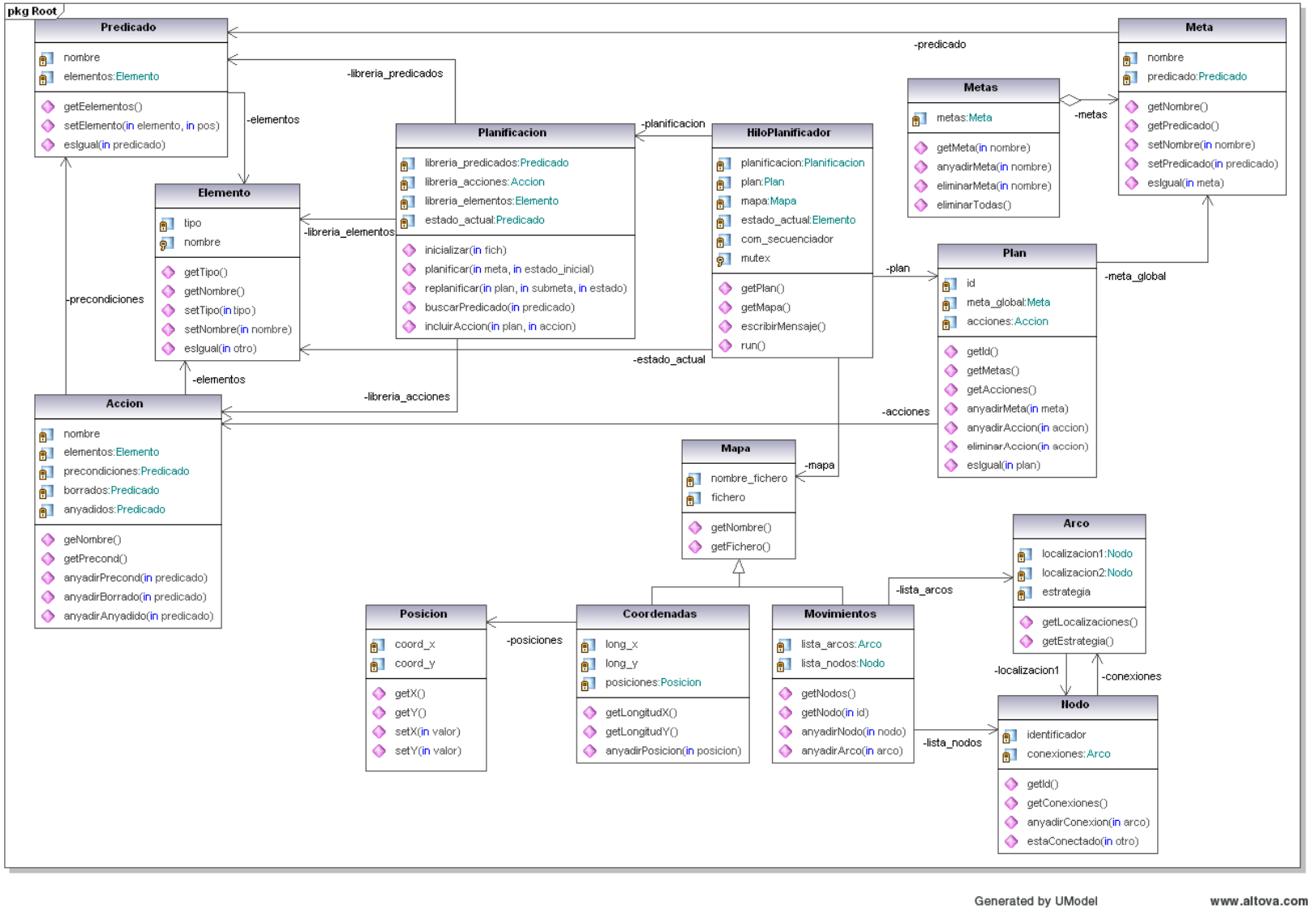

pkg Component View
Puerto_Planificador:iSecuenciadorPlanificador
«component»
Secuenciador
iSecuenciadorPlanificador
«component»
Interprete
«component»
Memoria
iSecuenciadorRobot
Puerto_Robot
iSecuenciadorControlador
Puerto_Controlador
Generated by UModel
www.altova.com


pkg Component View
Secuenciador
Interprete
(from Secuenciador)
Memoria
(from Secuenciador)
Tarea
HiloSecuenciador
agenda:Agenda
secuenciacion:Secu
monitor:Monitor
tarea_actual:Tarea
-tarea_actual
-monitor
Monitor
RAP
-tareas
Tareas
tareas:Tarea
-secuenciacion
-solucion
-tareas
Agenda
tareas:Tarea
Secuenciacion
historial:Historial
libreria_RAPs:Libreri
libreria_tareas:Tarea
libreria_metas:Metas
solucion:RAP
secuencia:Tarea
LibreriaRAPs
lista:RAP
-lista
-soluciones
Historial
planes:Plan
soluciones:RAP
secuencias:Tarea
-agenda
-libreria_RAPs
-libreria_tareas
-historial
-secuencia
-secuencias
Generated by UModel
www.altova.com


pkg Root
Agenda
plan:Plan
tareas:Tarea
crearAgenda(in plan, in secuencia)
registrarPlan(in plan)
anyadirTarea(in tarea, in pos)
eliminarTarea(in tarea)
hayMasTareas()
siguiente()
-agenda
HiloSecuenciador
secuenciacion:Secuenciacion
agenda:Agenda
monitor:Monitor
plan:Plan
tarea_actual:Tarea
com_controlador
com_planificador
mutex
modificarAgenda(in accion, in tarea, in pos)
nuevoPasoSecuencia(in pos)
recuperarSecuencia()
leerMensaje(in thread, in mensaje)
escribirMensaje(in thread, in mensaje)
run()
-monitor
Monitor
robot:Robot
estado:Estado
pos_robot:Posicion
actualizar()
actualizarEstado()
actualizarPosicion()
-tareas
Tareas
tareas:Tarea
getTareas()
getTarea(in submeta)
anyadirTarea(in tarea)
eliminarTarea(in tarea)
eliminarTodas()
-tareas
Tarea
nombre
submeta:Meta
acciones:Accion
getNombre()
getAcciones()
getMeta()
anyadirAccion(in accion)
eliminarAccion(in accion)
-tarea_actual
-secuenciacion
-secuencias
Historial
planes:Plan
soluciones:RAP
secuencias:Tarea
getPlanes()
getSolucion(in plan)
guardarPlan(in plan, in solucion, in secuencia)
getSecuencia(in plan)
-soluciones
-secuencia
Secuenciacion
plan:Plan
historial:Historial
libreria_RAPs:LibreriaRAPs
libreria_tareas:Tareas
libreria_metas:Metas
solucion:RAP
secuencia:Tarea
secuenciar()
asignarTarea(in accion)
asignarRAP(in tarea)
seleccionarHabilidad(in rap, in estado, in pos)
-historial
-libreria_tareas
-solucion
-libreria_RAPs
LibreriaRAPs
lista:RAP
getLista()
getRAP(in id)
anyadirRAP(in rap)
-lista
RAP
identificador
submeta:Meta
habilidades
getId()
getMeta()
getHabilidades()
setId(in id)
Generated by UModel
www.altova.com


pkg Component View
«component»
Planificador
«component»
ModeloMundo
«component»
GestorPlanes
iPlanificadorRobot
Puerto_Robot
iPlanificadorSecuenciador
Puerto_Secuenciador
Generated by UModel
www.altova.com


pkg Component View
Planificador
GestorPlanes
(from Planificador)
ModeloMundo
(from Planificador)
Predicado
elementos:Elemer
Meta
predicado:Predica
Metas
metas:Meta
Elemento
Planificacion
libreria_predicadc
libreria_acciones:
libreria_elementos
estado_actual:Pre
HiloPlanificador
planificacion:Planific
plan:Plan
mapa:Mapa
estado_actual:Predic
Accion
precondiciones:Pi
borrados:Predica
anyadidos:Predic
Plan
meta_global:Meta
acciones:Accion
Mapa
Coordenadas
posiciones:Pos
Movimientos
lista_arcos:Arco
lista_nodos:Nodo
Posicion
Arco
Nodo
-predicado
-metas
-estado_actual
-libreria_predicados
-elementos
-libreria_elementos
-planificacion
-plan
-libreria_acciones
-precondiciones
-acciones
-meta_global
-mapa
-posiciones
-lista_arcos
-lista_nodos
Generated by UModel
www.altova.com


pkg Root
Predicado
nombre
elementos:Elemento
getEelementos()
setElemento(in elemento, in pos)
esIgual(in predicado)
Meta
nombre
predicado:Predicado
getNombre()
getPredicado()
setNombre(in nombre)
setPredicado(in predicado)
esIgual(in meta)
-predicado
-libreria_predicados
-elementos
Metas
metas:Meta
getMeta(in nombre)
anyadirMeta(in nombre)
eliminarMeta(in nombre)
eliminarTodas()
-metas
Planificacion
libreria_predicados:Predicado
libreria_acciones:Accion
libreria_elementos:Elemento
estado_actual:Predicado
inicializar(in fich)
planificar(in meta, in estado_inicial)
replanificar(in plan, in submeta, in estado)
buscarPredicado(in predicado)
incluirAccion(in plan, in accion)
-planificacion
HiloPlanificador
planificacion:Planificacion
plan:Plan
mapa:Mapa
estado_actual:Elemento
com_secuenciador
mutex
getPlan()
getMapa()
escribirMensaje()
run()
Elemento
tipo
nombre
getTipo()
getNombre()
setTipo(in tipo)
setNombre(in nombre)
esIgual(in otro)
-libreria_elementos
-precondiciones
-elementos
-estado_actual
-plan
Plan
id
meta_global:Meta
acciones:Accion
getId()
getMetas()
getAcciones()
anyadirMeta(in meta)
anyadirAccion(in accion)
eliminarAccion(in accion)
esIgual(in plan)
-meta_global
Accion
nombre
elementos:Elemento
precondiciones:Predicado
borrados:Predicado
anyadidos:Predicado
geNombre()
getPrecond()
anyadirPrecond(in predicado)
anyadirBorrado(in predicado)
anyadirAnyadido(in predicado)
-libreria_acciones
-acciones
Mapa
nombre_fichero
fichero
getNombre()
getFichero()
-mapa
Arco
localizacion1:Nodo
localizacion2:Nodo
estrategia
getLocalizaciones()
getEstrategia()
-lista_arcos
Posicion
coord_x
coord_y
getX()
getY()
setX(in valor)
setY(in valor)
-posiciones
Coordenadas
long_x
long_y
posiciones:Posicion
getLongitudX()
getLongitudY()
anyadirPosicion(in posicion)
Movimientos
lista_arcos:Arco
lista_nodos:Nodo
getNodos()
getNodo(in id)
anyadirNodo(in nodo)
anyadirArco(in arco)
-lista_nodos
-localizacion1
-conexiones
Nodo
identificador
conexiones:Arco
getId()
getConexiones()
anyadirConexion(in arco)
estaConectado(in otro)
Generated by UModel
www.altova.com

Anexo D

Tablas de Componentes

Identificador	CO-01	**Nombre**	Infraestructura
Subsistema	No procede.		
Tipo	Componente.		
Propósito	Proporcionar a la arquitectura 3T el acceso al estado del robot y a los sensores y actuadores. Asimismo, permite al cliente hacer uso del sistema independientemente de la plataforma (simulador, robot real…)		
Dependencias	Ninguna.		
Interfaces	Proporciona las interfaces iCliente e iRobot.		
Proceso	El cliente inicializa el robot según sus necesidades. El robot mantendrá su estado coherente en todo momento, proporcionará la información obtenida por sus sensores y recibirá órdenes para sus actuadores.		
Datos	• Datos de configuración desde el usuario. • Información del mundo. • Órdenes mecánicas.		

Tabla 19: Componente CO-01

Identificador	CO-02	**Nombre**	Controlador
Subsistema	No procede.		
Tipo	Componente.		
Propósito	Implementar la capa reactiva de la arquitectura 3T, teniendo el control sobre las habilidades del robot y proporcionando al secuenciador el acceso a estas.		
Dependencias	Requiere la interfaz iRobot.		
Interfaces	Proporciona las interfaces iControlador.		
Proceso	El controlador genera un hilo de ejecución para las habilidades reactivas del robot. Lee la información de los sensores y actúa en consecuencia. Además, recibe órdenes sobre la secuenciación de las habilidades del componente Secuenciador.		
Datos	• Información de los sensores. • Grupos de habilidades a activar/desactivar. • Órdenes a los actuadores. • Estado del robot.		

Tabla 20: Componente CO-02

Identificador	CO-03	Nombre	GestorHabilidades
Subsistema	Controlador.		
Tipo	Componente.		
Propósito	Proporcionar al secuenciador las operaciones necesarias para activar y desactivar conjuntos de habilidades. Crea el hilo de ejecución de las habilidades y gestiona sus comunicaciones.		
Dependencias	Depende de los componentes Navegacion y Vision.		
Interfaces	Proporciona la interfaz iControlador.		
Proceso	Genera el hilo de ejecución de habilidades y gestiona sus comunicaciones con el resto de hilos de ejecución. El gestor recibe órdenes del secuenciador y procede a activar o desactivar los diferentes conjuntos de habilidades que se están ejecutando en su hilo.		
Datos	• Conjuntos de habilidades. • Mensajes de los hilos de ejecución.		

Tabla 21: Componente CO-03

Identificador	CO-04	Nombre	Navegacion
Subsistema	Controlador		
Tipo	Componente.		
Propósito	Proporcionar las habilidades relacionadas con la navegación del robot y los eventos asociados a ellas, que indican el éxito o no de estas habilidades.		
Dependencias	Requiere la interfaz iRobot.		
Interfaces	Ninguna.		
Proceso	Cada una de las habilidades de este conjunto obtendrá la información de los sensores y se ejecutarán las acciones correspondientes para conseguir el objetivo de la habilidad, dando órdenes a los actuadores del robot. En todo momento se comprueban los eventos que indican la finalización con éxito o no de la habilidad.		
Datos	• Información de los sensores. • Información del mundo. • Órdenes a los actuadores. • Estado del robot.		

Tabla 22: Componente CO-04

Identificador	CO-05	**Nombre**	Vision
Subsistema	Controlador		
Tipo	Componente.		
Propósito	Proporcionar las habilidades relacionadas con la visión del robot y los eventos asociados a ellas, que indican el éxito o no de estas habilidades.		
Dependencias	Requiere la interfaz iRobot.		
Interfaces	Ninguna.		
Proceso	Cada una de las habilidades de este conjunto obtendrá la información de los sensores y se ejecutarán las acciones correspondientes para conseguir el objetivo de la habilidad, dando órdenes a los actuadores del robot. En todo momento se comprueban los eventos que indican la finalización con éxito o no de la habilidad.		
Datos	• Información de los sensores. • Información del mundo. • Órdenes a los actuadores. • Estado del robot.		

Tabla 23: Componente CO-05

Identificador	CO-06	**Nombre**	Secuenciador
Subsistema	No procede.		
Tipo	Componente.		
Propósito	Implementar la capa intermedia que permite la interacción entre el comportamiento deliberativo y el reactivo de la arquitectura 3T, interpretando los planes para alcanzar las metas y activando o desactivando conjuntos de habilidades.		
Dependencias	Requiere las interfaces iControlador e iRobot.		
Interfaces	Proporciona la interfaz iSecuenciador.		
Proceso	El secuenciador genera un hilo de ejecución para el control de la capa intermedia de la arquitectura. Obtiene los planes para el robot y los interpreta, elaborando una secuencia de habilidades a ejecutar para cada tarea del plan. Esto lo hace activando o desactivando conjuntos de habilidades ofrecidos por el controlador.		
Datos	• Agenda de planes (metas, tareas y subtareas). • Conjunto de habilidades. • Estado del robot.		

Tabla 24: Componente CO-06

Identificador	CO-07	**Nombre**	Interprete
Subsistema	Secuenciador.		
Tipo	Componente.		
Propósito	Interpretar los planes elaborados por el planificador y diseñar una secuencia de habilidades a ejecutar para cada tarea y subtarea.		
Dependencias	Depende del componente Memoria. Requiere la interfaz iControlador.		
Interfaces	Proporciona parte de la interfaz iSecuenciador.		
Proceso	El intérprete obtiene los planes a ejecutar y accede a la memoria del sistema para diseñar una secuencia de habilidades idónea para alcanzar la meta del plan. Una vez hecho esto, activa y desactiva conjuntos de habilidades del controlador según el gestor de recursos lo requiera. Si la memoria capta un cambio significante en el modelo del mundo, el intérprete realiza una pequeña replanificación del plan.		
Datos	• Agenda de planes (metas, tareas y subtareas). • Conjunto de habilidades. • Estado del robot.		

Tabla 25: Componente CO-07

Identificador	CO-08	**Nombre**	Memoria
Subsistema	Secuenciador.		
Tipo	Componente.		
Propósito	Permite al subsistema tener una biblioteca o historial de planes pasados y su resolución con distintas secuencias de habilidades. Además, almacena los pasos realizados para el plan actual y el estado del robot en cada momento.		
Dependencias	Depende del componente Interprete. Requiere la interfaz iRobot.		
Interfaces	Proporciona parte de la interfaz iSecuenciador.		
Proceso	La memoria del sistema recibe el plan a ejecutar y se compara con planes antiguos para facilitar el diseño de la secuencia de habilidades. Además, hace uso del historial de acciones y secuencias llevadas a cabo para el plan actual, así como del estado del robot, para ayudar al intérprete en el rediseño de la secuencia en caso de ser necesario.		
Datos	• Agenda de planes (metas, tareas y subtareas). • Conjunto de habilidades. • Estado del robot.		

Tabla 26: Componente CO-08

Identificador	CO-09	**Nombre**	Planificador
Subsistema	No procede.		
Tipo	Componente.		
Propósito	Implementar la capa deliberativa de la arquitectura 3T, elaborando los planes para alcanzar las metas y permitiendo la localización del robot y lectura de mapas que representan el entorno.		
Dependencias	Requiere las interfaces iSecuenciador e iRobot.		
Interfaces	Ninguna.		
Proceso	El planificador genera un hilo de ejecución para la planificación deliberativa del robot. Lee el mapa del entorno, localiza al robot dentro de éste y elabora un plan, consistente en metas, tareas y subtareas. Proporciona esta información al secuenciador para su ejecución.		
Datos	• Localización del robot. • Mapa del entorno. • Estado del robot.		

Tabla 27: Componente CO-09

Identificador	CO-10	**Nombre**	GestorPlanes
Subsistema	Planificador.		
Tipo	Componente.		
Propósito	Crea el hilo de ejecución de la planificación y gestiona sus comunicaciones. Monitoriza la ejecución del plan, actualizando la localización del robot dentro del mapa del entorno para identificar tareas ya realizadas o modificar el plan en caso de ser necesario.		
Dependencias	Depende del componente ModeloMundo. Requiere las interfaces iSecuenciador e iRobot.		
Interfaces	Ninguna.		
Proceso	Genera el hilo de ejecución de la planificación y gestiona sus comunicaciones con el resto de hilos de ejecución. El gestor recibe información del robot y del entorno, la compara con el mapa y así localiza al robot dentro de éste. A partir de esta información, elabora un plan con un algoritmo de búsqueda para alcanzar las metas y le proporciona este plan al secuenciador.		
Datos	• Localización del robot. • Mapa del entorno. • Estado del robot.		

Tabla 28: Componente CO-10

Identificador	CO-11	**Nombre**	ModeloMundo
Subsistema	Planificador.		
Tipo	Componente.		
Propósito	Proporcionar a la arquitectura 3T una representación del entorno del robot mediante mapas. La información contenida en estos mapas es proporcionada al gestor de planes para la elaboración de un plan.		
Dependencias	Depende del componente GestorPlanes.		
Interfaces	Ninguna		
Proceso	Este componente representa el entorno del robot y proporciona al gestor de planes el acceso e interpretación de los mapas para elaborar una secuencia de tareas y subtareas que permitan alcanzar las metas.		
Datos	• Mapa del entorno.		

Tabla 29: Componente CO-11

Anexo E

Tablas de Clases

Identificador	CL-01	**Nombre**	Robot
Componente al que pertenece	CO-01		
Tipo	Clase.		
Propósito / Función	Sirve como enlace único entre la arquitectura del sistema y el cliente software. Representa el robot físico, con todas sus características, y proporciona el acceso a éstas y al estado del propio robot.		
Dependencias	Depende de las clases Sensor, Actuador, Estado y Arquitectura.		
Métodos	• inicializar(): Crea las instancias necesarias de sensores y actuadores según el robot físico que se utilizará en la ejecución del sistema. Asimismo, inicializa el estado del robot y otros posibles elementos.		

Tabla 30: Clase CL-01

Identificador	CL-02	**Nombre**	Estado
Componente al que pertenece	CO-01		
Tipo	Clase.		
Propósito / Función	Representa el estado del robot dentro del modelo del mundo: su posición en el mapa, la tarea actual encomendada, el nivel de batería y demás elementos significativos definidos para el problema.		
Dependencias	Depende de las clases Posicion y Tarea.		
Métodos	No relevantes.		

Tabla 31: Clase CL-02

Identificador	CL-03	**Nombre**	Sensor
Componente al que pertenece	CO-01		
Tipo	Clase.		
Propósito / Función	Representa un elemento hardware genérico que permite al robot obtener información externa del mundo. Es una generalización de los diferentes tipos de sensores de los que puede hacer uso el robot.		
Dependencias	Ninguna.		
Métodos	No relevantes.		

Tabla 32: Clase CL-03

Identificador	CL-04	**Nombre**	Sonar
Componente al que pertenece	CO-01		
Tipo	Clase.		
Propósito / Función	Define las características propias de este tipo de sensor e implementa la funcionalidad relacionada con ellas.		
Dependencias	Hereda de la clase Sensor.		
Métodos	No relevantes.		

Tabla 33: Clase CL-04

Identificador	CL-05	**Nombre**	Bumpers
Componente al que pertenece	CO-01		
Tipo	Clase.		
Propósito / Función	Define las características propias de este tipo de sensor e implementa la funcionalidad relacionada con ellas.		
Dependencias	Hereda de la clase Sensor.		
Métodos	No relevantes.		

Tabla 34: Clase CL-05

Identificador	CL-06	**Nombre**	Laser
Componente al que pertenece	CO-01		
Tipo	Clase.		
Propósito / Función	Sirve como ejemplo de extensibilidad de la representación del robot a la hora de añadir nuevos elementos en el modelo del mundo.		
Dependencias	Hereda de la clase Sensor.		
Métodos	No relevantes.		

Tabla 35: Clase CL-06

Identificador	CL-07	**Nombre**	Actuador
Componente al que pertenece	CO-01		
Tipo	Clase.		
Propósito / Función	Representa un elemento hardware genérico que permite al robot interactuar con el mundo. Es una generalización de los diferentes tipos de actuadores de los que puede hacer uso el robot.		
Dependencias	Ninguna.		
Métodos	No relevantes		

Tabla 36: Clase CL-07

Identificador	CL-08	**Nombre**	Motor
Componente al que pertenece	CO-01		
Tipo	Clase.		
Propósito / Función	Define las características propias de este tipo de actuador e implementa la funcionalidad relacionada con ellas.		
Dependencias	Hereda de la clase Actuador.		
Métodos	No relevantes.		

Tabla 37: Clase CL-08

Identificador	CL-09	**Nombre**	Batería
Componente al que pertenece	CO-01		
Tipo	Clase abstracta.		
Propósito / Función	Sirve como ejemplo de extensibilidad de la representación del robot a la hora de añadir nuevos elementos en el modelo del mundo.		
Dependencias	No aplicable.		
Métodos	No aplicable.		

Tabla 38: Clase CL-09

Identificador	CL-10	**Nombre**	Arquitectura
Componente al que pertenece	CO-01		
Tipo	Clase.		
Propósito / Función	Gestiona la creación e inicialización única de los tres hilos de ejecución de la arquitectura del robot.		
Dependencias	Depende de las clases HiloControlador, HiloSecuenciador e HiloPlanificador.		
Métodos	• ejecutar(): Lanza la ejecución de los tres hilos de la arquitectura.		

Tabla 39: Clase CL-10

Identificador	CL-11	**Nombre**	HiloControlador
Componente al que pertenece	CO-02, CO-03		
Tipo	Clase <Thread>.		
Propósito / Función	Mantiene la ejecución paralela del subsistema Controlador, ejecutando la parte reactiva de la arquitectura (habilidades). Gestiona las comunicaciones con el resto de hilos de ejecución.		
Dependencias	Depende de las clases Habilidades, Habilidad y Tarea.		
Métodos	• anyadirHabilidad(): Añade una habilidad a la lista de habilidades que soporta el robot. • ejecutarHabilidad(): Pone en ejecución la habilidad concreta con la que se puede llevar a cabo con éxito la tarea actual. • activarConjunto(): Activa un conjunto concreto de habilidades, según las órdenes del secuenciador. Las habilidades activadas podrán ser ejecutadas para llevar a cabo la tarea actual. • cambiarTarea(): Sustituye la tarea actual que se está llevando a cabo por una tarea nueva, independientemente del éxito o no de la anterior. • informarChoque(): Desencadena la secuencia de operaciones necesaria para recuperarse de un fallo en la ejecución de una habilidad, informando al secuenciador. • leerMensaje(): Recibe e interpreta un mensaje del flujo de comunicación con otros hilos de ejecución. • run(): Método heredado de la clase Thread para la ejecución de hilos.		

Tabla 40: Clase CL-11

Identificador	CL-12	**Nombre**	Habilidades
Componente al que pertenece	CO-02, CO-03		
Tipo	Clase.		
Propósito / Función	Agregado de habilidades, se trata de la lista que incluye todas las habilidades implementadas que puede ejecutar el robot. Esta clase las engloba y proporciona el acceso a ellas.		
Dependencias	Depende de la clase Habilidad.		
Métodos	No relevantes.		

Tabla 41: Clase CL-12

Identificador	CL-13	**Nombre**	Habilidad
Componente al que pertenece	CO-02, CO-03		
Tipo	Clase.		
Propósito / Función	Representa un método de ejecución que el robot puede llevar a cabo para resolver tareas. Es una generalización de los diferentes tipos de habilidades que puede ejecutar el robot.		
Dependencias	Depende de la clase Robot.		
Métodos	• ejecutar(): Pone en ejecución el código de la habilidad. Éste es un bucle infinito que realiza una acción concreta hasta que se produce un evento o hasta que el hilo controlador cancela la ejecución de la habilidad. • comprobarExito(): Comprueba la condición que indica que se ha dado el evento que finaliza la ejecución de la habilidad.		

Tabla 42: Clase CL-13

Identificador	CL-14	**Nombre**	Esperar
Componente al que pertenece	CO-02, CO-03		
Tipo	Clase.		
Propósito / Función	Define las características propias de esta habilidad especial de control del flujo de ejecución.		
Dependencias	Hereda de la clase Habilidad.		
Métodos	• ejecutar(): Ver clase Habilidad (Cl-13). • comprobarExito(): Ver clase Habilidad (CL-13).		

Tabla 43: Clase CL-14

Identificador	CL-15	**Nombre**	Finalizar
Componente al que pertenece	CO-02, CO-03		
Tipo	Clase.		
Propósito / Función	Define las características propias de esta habilidad especial de control del flujo de ejecución.		
Dependencias	Hereda de la clase Habilidad.		
Métodos	• ejecutar(): Ver clase Habilidad (Cl-13). • comprobarExito(): Ver clase Habilidad (CL-13).		

Tabla 44: Clase CL-15

Identificador	CL-16	**Nombre**	IrA
Componente al que pertenece	CO-02, CO-04		
Tipo	Clase.		
Propósito / Función	Define las características propias de esta habilidad e implementa su bucle de ejecución y la condición del evento de finalización.		
Dependencias	Hereda de la clase Habilidad.		
Métodos	• ejecutar(): Ver clase Habilidad (Cl-13). • comprobarExito(): Ver clase Habilidad (CL-13).		

Tabla 45: Clase CL-16

Identificador	CL-17	**Nombre**	SeguirPared
Componente al que pertenece	CO-02, CO-04		
Tipo	Clase.		
Propósito / Función	Define las características propias de esta habilidad e implementa su bucle de ejecución y la condición del evento de finalización.		
Dependencias	Hereda de la clase Habilidad.		
Métodos	• ejecutar(): Ver clase Habilidad (Cl-13). • comprobarExito(): Ver clase Habilidad (CL-13).		

Tabla 46: Clase CL-17

Identificador	CL-18	**Nombre**	EvitarObstaculo
Componente al que pertenece	CO-02, CO-04		
Tipo	Clase.		
Propósito / Función	Define las características propias de esta habilidad e implementa su bucle de ejecución y la condición del evento de finalización.		
Dependencias	Hereda de la clase Habilidad.		
Métodos	• ejecutar(): Ver clase Habilidad (Cl-13). • comprobarExito(): Ver clase Habilidad (CL-13).		

Tabla 47: Clase CL-18

Identificador	CL-19	**Nombre**	Recuperarse
Componente al que pertenece	CO-02, CO-04		
Tipo	Clase.		
Propósito / Función	Define las características propias de esta habilidad e implementa su bucle de ejecución y la condición del evento de finalización.		
Dependencias	Hereda de la clase Habilidad.		
Métodos	• ejecutar(): Ver clase Habilidad (Cl-13). • comprobarExito(): Ver clase Habilidad (CL-13).		

Tabla 48: Clase CL-19

Identificador	CL-20	**Nombre**	Avanzar
Componente al que pertenece	CO-02, CO-04		
Tipo	Clase.		
Propósito / Función	Define las características propias de esta habilidad e implementa su bucle de ejecución y la condición del evento de finalización.		
Dependencias	Hereda de la clase Habilidad.		
Métodos	• ejecutar(): Ver clase Habilidad (Cl-13). • comprobarExito(): Ver clase Habilidad (CL-13).		

Tabla 49: Clase CL-20

Identificador	CL-21	**Nombre**	Girar
Componente al que pertenece	CO-02, CO-04		
Tipo	Clase.		
Propósito / Función	Define las características propias de esta habilidad e implementa su bucle de ejecución y la condición del evento de finalización.		
Dependencias	Hereda de la clase Habilidad.		
Métodos	• ejecutar(): Ver clase Habilidad (Cl-13). • comprobarExito(): Ver clase Habilidad (CL-13).		

Tabla 50: Clase CL-21

Identificador	CL-22	**Nombre**	CubrirDistancia
Componente al que pertenece	CO-02, CO-04		
Tipo	Clase.		
Propósito / Función	Define las características propias de esta habilidad e implementa su bucle de ejecución y la condición del evento de finalización.		
Dependencias	Hereda de la clase Habilidad.		
Métodos	• ejecutar(): Ver clase Habilidad (Cl-13). • comprobarExito(): Ver clase Habilidad (CL-13).		

Tabla 51: Clase CL-22

Identificador	CL-23	**Nombre**	DetectarObstaculo
Componente al que pertenece	CO-02, CO-05		
Tipo	Clase abstracta.		
Propósito / Función	Sirve como ejemplo de extensibilidad de la funcionalidad del robot a la hora de añadir nuevas habilidades que puede ejecutar.		
Dependencias	No aplicable.		
Métodos	No aplicable.		

Tabla 52: Clase CL-23

Identificador	CL-24	**Nombre**	DetectarBaliza
Componente al que pertenece	CO-02, CO-05		
Tipo	Clase abstracta.		
Propósito / Función	Sirve como ejemplo de extensibilidad de la funcionalidad del robot a la hora de añadir nuevas habilidades que puede ejecutar.		
Dependencias	No aplicable.		
Métodos	No aplicable.		

Tabla 53: Clase CL-24

Identificador	CL-25	**Nombre**	HiloSecuenciador
Componente al que pertenece	CO-06, CO-07		
Tipo	Clase <Thread>.		
Propósito / Función	Mantiene la ejecución paralela del subsistema Secuenciador, ejecutando la parte capa intermedia de la arquitectura (secuenciación). Gestiona las comunicaciones con el resto de hilos de ejecución.		
Dependencias	Depende de las clases Agenda, Secuenciacion, Monitor, Plan y Tarea.		
Métodos	• modificarAgenda(): Añade, elimina o modifica el orden de una tarea en la agenda de tareas. • nuevoPasoSecuencia(): Obtiene de la agenda una nueva tara a poner en ejecución cuando la anterior se ha llevado a cabo con éxito. • recuperarSecuencia(): Modifica la agenda tras un fallo en la ejecución de la secuencia de tareas. • leerMensaje():Recibe e interpreta un mensaje del flujo de comunicación con otros hilos de ejecución. • escribirMensaje(): Envía un mensaje por el flujo de comunicación con otros hilos de ejecución. • run():Método heredado de la clase Thread para la ejecución de hilos.		

Tabla 54: Clase CL-25

Identificador	CL-26	**Nombre**	Agenda
Componente al que pertenece	CO-06, CO-07		
Tipo	Clase.		
Propósito / Función	Trata la secuencia de tareas que forma parte de la solución de un plan, dándoles un orden. Las tareas pendientes se almacenan en una lista y se van sacando según sea necesario.		
Dependencias	Depende de las clases Plan y Tarea.		
Métodos	• registrarPlan(): Almacena el plan elaborado por la capa superior, haciendo corresponder a cada acción del plan una tarea en la agenda. • modificarPlan(): Modifica el plan almacenado si se elabora una replanificación en la capa superior. • anyadirTarea(): Incluye una nueva tarea dentro de la agenda. • eliminarTarea(): Elimina una tarea de la agenda, bien por orden de otros elementos del sistema, bien porque se extrae de la agenda para ser ejecutada. • hayMasTareas(): Indica si existen tareas pendientes en la agenda. • siguiente(): Extrae la primera tarea de la agenda para su ejecución.		

Tabla 55: Clase CL-26

Identificador	CL-27	**Nombre**	Tareas
Componente al que pertenece	CO-06, CO-07		
Tipo	Clase.		
Propósito / Función	Agregado de tareas, se trata de la lista que incluye todas las tareas implementadas que puede llevar a cabo el robot. Esta clase las engloba y proporciona el acceso a ellas.		
Dependencias	Depende de la clase Tarea.		
Métodos	No relevantes		

Tabla 56: Clase CL-27

Identificador	CL-28	**Nombre**	Tarea
Componente al que pertenece	CO-06, CO-07		
Tipo	Clase.		
Propósito / Función	Representa el conjunto de acciones a llevar a cabo para cumplir una submeta dentro del plan establecido. Es el elemento mínimo de secuenciación, que permite seleccionar las habilidades a ejecutar para alcanzar dicha submeta.		
Dependencias	Depende de las clases Meta y Accion.		
Métodos	• anyadirAccion(): Añade una nueva acción a la tarea. • eliminarAccion(): Elimina una acción de la tarea.		

Tabla 57: Clase CL-28

Identificador	CL-29	**Nombre**	Secuenciacion
Componente al que pertenece	CO-06, CO-07		
Tipo	Clase.		
Propósito / Función	Realiza la secuenciación de tareas, asignando un RAP a cada una de ellas y seleccionando posteriormente una habilidad concreta de ese RAP. Se encarga de la activación y desactivación de conjuntos de habilidades y de mandar órdenes al controlador.		
Dependencias	Depende de las clases Plan, Historial, LibreriaRAPs, Tareas, Metas, RAP y Tarea.		
Métodos	• secuenciar(): Inicia el proceso de secuenciación. A partir de las tareas identificadas para el plan, establece un orden de ejecución de habilidades para llevarlo a cabo. • asignarTarea(): Método de apoyo para secuenciar. Dada una acción del plan, la transforma en una acción de entre la librería de tareas del sistema. • asignarRAP(): Método de apoyo para secuenciar. Dada una tarea, asigna un RAP en función de la submeta de dicha tarea. • seleccionarHabilidad(): Método de apoyo para secuenciar. Dada un RAP, selecciona una de las habilidades que lo definen en función del estado del robot y el modelo del mundo.		

Tabla 58: Clase CL-29

Identificador	CL-30	**Nombre**	Monitor
Componente al que pertenece	CO-06, CO-08		
Tipo	Clase.		
Propósito / Función	Monitoriza el estado del robot y el modelo del mundo, registrando los cambios que se producen en ambos elementos. Se encarga también de notificar al hilo de ejecución del secuenciador si se requiere de una replanificación debido a alguno de esos cambios.		
Dependencias	Depende de las clases Robot, Estado y Posicion.		
Métodos	• actualizarEstado(): Pregunta al robot por su estado y actualiza su copia en función de la respuesta obtenida. • actualizarPosicion(): Pregunta al modelo del mundo por las diferentes posiciones relevantes y actualiza sus copias en función de la respuesta obtenida. • actualizar(): Engloba las diferentes actualizaciones de los atributos del monitor, para realizarlas en una única llamada.		

Tabla 59: Clase CL-30

Identificador	CL-31	**Nombre**	Historial
Componente al que pertenece	CO-06, CO-08		
Tipo	Clase.		
Propósito / Función	Almacena el historial de planes pasados y sus soluciones para facilitar la secuenciación del plan actual, y proporciona acceso a éstos.		
Dependencias	Depende de las clases Plan, RAP y Tarea.		
Métodos	• guardarPlan(): Almacena el plan actual una vez es satisfecho para posteriores consultas.		

Tabla 60: Clase CL-31

Identificador	CL-32	Nombre	LibreriaRAPs
Componente al que pertenece	CO-06, CO-08		
Tipo	Clase.		
Propósito / Función	Agregado de RAPs, se trata de la lista que incluye todos los RAPs definidos en el sistema para la secuenciación de tareas. Esta clase los engloba y proporciona el acceso a ellos.		
Dependencias	Depende de la clase RAP.		
Métodos	No relevantes.		

Tabla 61: Clase CL-32

Identificador	CL-27	Nombre	RAP
Componente al que pertenece	CO-06, CO-08		
Tipo	Clase.		
Propósito / Función	Representa la simplificación implementada de un RAP (Reactive Action Package). Define el conjunto de habilidades idóneas para llevar a cabo una tarea en función de la submeta que se busque lograr.		
Dependencias	Depende de la clase Meta.		
Métodos	No relevantes.		

Tabla 62: Clase CL-27

Identificador	CL-34	**Nombre**	HiloPlanificador
Componente al que pertenece	CO-09, CO-10		
Tipo	Clase <Thread>.		
Propósito / Función	Mantiene la ejecución paralela del subsistema Planificador, ejecutando la parte capa deliberativa de la arquitectura (planificación). Gestiona las comunicaciones con el resto de hilos de ejecución.		
Dependencias	Depende de las clases Planificacion, Plan, Mapa y Elemento.		
Métodos	• escribirMensaje(): Envía un mensaje por el flujo de comunicación con otros hilos de ejecución. • run():Método heredado de la clase Thread para la ejecución de hilos.		

Tabla 63: Clase CL-34

Identificador	CL-35	**Nombre**	Planificacion
Componente al que pertenece	CO-09, CO-10		
Tipo	Clase.		
Propósito / Función	Se encarga de realizar la planificación, en función del objetivo final del robot. Utiliza un algoritmo de búsqueda para elaborar el plan, haciendo uso para ello de los elementos clásicos de planificación: acciones y predicados.		
Dependencias	Depende de las clases Predicado, Accion y Elemento.		
Métodos	• inicializar(): Crea las librerías de predicados, elementos y acciones a partir del fichero de configuración del planificador. • planificar(): Inicia el proceso de planificación. A partir de la meta final del robot y el estado inicial, elabora un árbol de acciones a llevar a cabo para alcanzar dicha meta. • replanificar(): Realiza un proceso de planificación sobre el plan elaborado previamente, a partir de una submeta no alcanzada con éxito. Esta replanificación sólo se realiza si el secuenciador no ha sido capaz de resolver el problema. • buscarPredicado(): Método de apoyo para planificar. A partir de un predicado objetivo, se busca la forma de llegar a él mediante la inclusión de nuevas acciones en el plan. • incluirAccion(): Método de apoyo para planificar. Añade una acción concreta al plan que está siendo elaborado.		

Tabla 64: Clase CL-35

Identificador	CL-36	**Nombre**	Elemento
Componente al que pertenece	CO-09, CO-10		
Tipo	Clase.		
Propósito / Función	Representa cualquier elemento del modelo del mundo con relevancia en la planificación. Forma parte de los predicados para definir el estado del modelo del mundo en un momento concreto.		
Dependencias	Ninguna.		
Métodos	• esIgual(): Compara dos elementos e indica si son iguales o no.		

Tabla 65: Clase CL-36

Identificador	CL-37	**Nombre**	Predicado
Componente al que pertenece	CO-09, CO-10		
Tipo	Clase.		
Propósito / Función	Representa la parte mínima del estado del modelo del mundo del concepto clásico de planificación. Define una verdad en el modelo del mundo en la que se ven involucrados uno o varios elementos del modelo.		
Dependencias	Depende de la clase Elemento.		
Métodos	• esIgual(): Compara dos predicados e indica si son iguales o no.		

Tabla 66: Clase CL-37

Identificador	CL-38	**Nombre**	Accion
Componente al que pertenece	CO-09, CO-10		
Tipo	Clase.		
Propósito / Función	Representa una operación del concepto clásico de planificación. A partir de unos predicados que indican las precondiciones, ejecutando esta acción se llega a cambiar el modelo del mundo con otros predicados que indican las postcondiciones.		
Dependencias	Depende de las clases Predicado y Elemento.		
Métodos	No relevantes.		

Tabla 67: Clase CL-38

Identificador	CL-39	**Nombre**	Plan
Componente al que pertenece	CO-09, CO-10		
Tipo	Clase.		
Propósito / Función	Contiene la meta final del robot para un problema concreto y la secuencia ordenada de acciones que se deben llevar a cabo para alcanzar dicha meta. A partir de este plan, las capas inferiores trabajan para que el robot lo ejecute con éxito.		
Dependencias	Depende de las clases Meta y Accion.		
Métodos	• anyadirMeta(): Permite inicializar el plan con el objetivo global del robot para su posterior elaboración. • anyadirAccion(): Añade una nueva acción al plan. • eliminarAccion(): Elimina del plan una acción. • esIgual(): Compara dos planes e indica si son iguales o no.		

Tabla 68: Clase CL-39

Identificador	CL-40	**Nombre**	Metas
Componente al que pertenece	CO-09, CO-10		
Tipo	Clase.		
Propósito / Función	Agregado de meta, se trata de la lista que incluye todas las metas definidos en el sistema para la elaboración de planes. Esta clase las engloba y proporciona el acceso a ellas.		
Dependencias	Depende de la clase Meta.		
Métodos	No relevantes.		

Tabla 69: Clase CL-40

Identificador	CL-41	**Nombre**	Meta
Componente al que pertenece	CO-09, CO-10		
Tipo	Clase.		
Propósito / Función	Representa un objetivo del robot, que debe cumplirse para considerar el éxito de una tarea o del plan completo. Sirve como índice o identificador para los diferentes RAPs que forman parte del sistema.		
Dependencias	Depende de la clase Predicado.		
Métodos	• esIgual(): Compara dos metas e indica si son iguales o no.		

Tabla 70: Clase CL-41

Identificador	CL-42	**Nombre**	Mapa
Componente al que pertenece	CO-09, CO-11		
Tipo	Clase.		
Propósito / Función	Representa el mapa del modelo mundo, por el que el robot se mueve. Sirve como apoyo a la planificación para la identificación de submetas en busca de la meta global del robot. Contiene la información sobre los elementos relevantes identificados para el problema, tales como la posición del robot, de la meta o de la posible base de recarga de la batería. Es una generalización de los diferentes tipos de habilidades que puede ejecutar el robot.		
Dependencias	Ninguna.		
Métodos	No relevantes.		

Tabla 71: Clase CL-42

Identificador	CL-43	**Nombre**	Coordenadas
Componente al que pertenece	CO-09, CO-11		
Tipo	Clase.		
Propósito / Función	Define las características propias de este tipo de mapa, consistente en una localización clásica de coordenadas y vectores.		
Dependencias	Depende de la clase Posicion. Hereda de la clase Mapa.		
Métodos	No relevantes.		

Tabla 72: Clase CL-43

Identificador	CL-44	Nombre	Posicion
Componente al que pertenece	CO-09, CO-11		
Tipo	Clase.		
Propósito / Función	Representa una localización concreta dentro de un mapa de coordenadas, definida por sus valores respecto a los ejes del mismo.		
Dependencias	Ninguna.		
Métodos	No relevantes.		

Tabla 73: Clase CL-44

Identificador	CL-45	Nombre	Movimientos
Componente al que pertenece	CO-09, CO-11		
Tipo	Clase.		
Propósito / Función	Define las características propias de este tipo de mapa, consistente en una localización mediante grafos y estrategias de movimiento entre las localizaciones.		
Dependencias	Depende de las clases Nodo y Arco.		
Métodos	No relevantes.		

Tabla 74: Clase CL-45

Identificador	CL-46	Nombre	Nodo
Componente al que pertenece	CO-09, CO-11		
Tipo	Clase.		
Propósito / Función	Representa una localización concreta dentro de un mapa de movimientos.		
Dependencias	Depende de la clase Arco.		
Métodos	• estaConectado(): Indica si desde el nodo se puede acceder a otro mediante alguna de sus conexiones directas.		

Tabla 75: Clase CL-46

Identificador	CL-47	**Nombre**	Arco
Componente al que pertenece	CO-09, CO-11		
Tipo	Clase.		
Propósito / Función	Representa una conexión entre dos nodos en un mapa de movimientos, lo que significa que se puede acceder desde uno de ellos al otro (y viceversa) ejecutando una estrategia de movimiento.		
Dependencias	Depende de la clase Nodo.		
Métodos	No relevantes.		

Tabla 76: Clase CL-47

NOTA: En las tablas anteriores algunos campos no se han rellenado. En su lugar, se han indicado dos tipos de valores para dichos campos, cuyo significado se explica a continuación:

- Ninguna: La clase/interfaz no tiene dependencias con otras.
- No relevantes: Los métodos de la clase se limitan a los métodos de acceso u otros métodos auxiliares no relevantes para entender el funcionamiento de la arquitectura.
- No aplicable: Los campos no se han rellenado debido a que no se ha implementado por completo la clase, como en el caso de las clases abstractas incluidas como ejemplo de extensibilidad de la arquitectura.

Anexo F

Librerías de Elementos de la Arquitectura

En este anexo se presentan de forma esquemática las librería de habilidades, tareas, metas y RAP que se han implementado en la arquitectura para proveer al robot de comportamientos inteligentes y realizar una serie de pruebas del sistema que aseguren el buen funcionamiento del mismo.

Asimismo, se presenta un ejemplo de lo que podría ser el dominio del planificador para llevar a cabo las tareas y habilidades implementadas. Se presenta un posible conjunto de acciones, predicados y elementos con los que podría tratar el planificador para elaborar planes como los que se han desarrollado en las pruebas.

Librería de habilidades de navegación	
Avanzar	**Descripción:** El robot avanza en línea recta con una velocidad determinada.
	Condiciones de éxito: • Detectar un obstáculo. • Detectar una pared. • Llegar a un punto concreto del mapa.
Girar	**Descripción:** El robot realiza un giro a la izquierda o a la derecha con una velocidad determinada.
	Condiciones de éxito: • Alcanzar una orientación concreta. • Indefinida.
Recuperarse	**Descripción:** Si el robot ha chocado con un obstáculo, retrocede hasta una distancia prudencial y realiza un giro en la dirección adecuada.
	Condiciones de éxito: • Dejar de detectar el obstáculo.
IrA	**Descripción:** El robot avanza en línea recta, teniendo en cuenta los posibles obstáculos encontrados y esquivándolos, hasta alcanzar una posición determinada del mapa.
	Condiciones de éxito: • Llegar a un punto concreto del mapa.
EvitarObstaculo	**Descripción:** El robot esquiva un obstáculo detectado previamente, realizando los giros y avances pertinentes para superar el obstáculo y encontrarse sin ningún peligro delante de él.
	Condiciones de éxito: • Dejar de detectar el obstáculo.

SeguirPared	**Descripción:** El robot avanza siguiendo una guía marcada por la pared, acercándose a ésta si se aleja demasiado, y alejándose en caso contrario. Además, esquiva cualquier obstáculo, esquina o cambio en la forma de la pared que se detecte.
	Condiciones de éxito: • Llegar a un punto concreto del mapa. • Indefinida.
CubrirDistancia	**Descripción:** El robot avanza en línea recta contando la distancia recorrida hasta alcanzar una distancia deseada.
	Condiciones de éxito: • Llegar a avanzar cierta distancia.

Tabla 77: Habilidades de navegación

Librería de habilidades de visión	
DetectarBaliza	**Descripción:** El robot trata la imagen tomada de la cámara e identifica un objeto de un color concreto.
	Condiciones de éxito: • La baliza ocupa un porcentaje concreto de la imagen.
DetectarObstaculo	**Descripción:** El robot trata la imagen tomada de la cámara e identifica un obstáculo sólido a una distancia considerada peligrosa de cara a una posible colisión.
	Condiciones de éxito: • El obstáculo ocupa un porcentaje concreto de la imagen.

Tabla 78: Habilidades de visión

Librería de RAPs	
RAP 1	**Submeta:** PuntoEnfrente
	Habilidades: • Girar • Avanzar
RAP 2	**Submeta:** LlegarA
	Habilidades: • Avanzar • Girar • IrA • SeguirPared
RAP 3	**Submeta:** DetectarAlgo
	Habilidades: • SeguirPared • Avanzar
RAP 4	**Submeta:** SuperarChoque
	Habilidades: • Recuperarse • EvitarObstaculo
RAP 5	**Submeta:** MismaDistancia
	Habilidades: • Avanzar • Girar • EvitarObstaculo
RAP 6	**Submeta:** EvitarChoque
	Habilidades: • EvitarObstaculo • Girar • Avanzar
RAP 7	**Submeta:** DistanciaRecorrida
	Habilidades: • CubrirDistancia

Tabla 79: Librería de RAPs

Librería de tareas	
Inicio	**Descripción:** Tarea especial de gestión del sistema que es asignada mientras dura la planificación.
	Submeta: No aplicable
Fin	**Descripción:** Tarea especial de gestión del sistema que es asignada tras la ejecución del plan. Permite salir del sistema y la desconexión del robot de forma segura.
	Submeta: No aplicable.
MantenerDireccion	**Descripción:** El robot debe avanzar en un movimiento rectilíneo hacia delante.
	Submeta: LlegarA
EncontrarAlgo	**Descripción:** El robot debe avanzar hacia delante o siguiendo una ruta preestablecida hasta toparse con algún elemento del mapa, ya sea un obstáculo o una pared, por ejemplo
	Submeta: DetectarAlgo
Orientarse	**Descripción:** El robot debe girar hasta conseguir alinearse con algún elemento del mapa, al que poder alcanzar posteriormente con un avance rectilíneo o mediante otra habilidad.
	Submeta: PuntoEnfrente
Retroceder	**Descripción:** El robot debe volver sobre sus pasos y realizar una rectificación en su movimiento hacia delante con el fin de evitar o recuperarse de un peligro.
	Submeta: SuperarChoque
SituarseMedio	**Descripción:** El robot debe orientarse en el entorno y colocarse en el medio de éste, situándose a la misma distancia de las dos paredes con menor distancia entre ellas.
	Submeta: MismaDistancia
Rodear	**Descripción:** El robot debe esquivar un obstáculo frente a él, preferiblemente rodeándolo, para poder continuar con su camino, evitando un choque con dicho obstáculo.
	Submeta: EvitarChoque
MantenerTrayectoria	**Descripción:** El robot debe avanzar en línea recta hacia adelante hasta cubrir una distancia concreta, esquivando los posibles obstáculos que se pueda encontrar y redirigiendo su trayectoria en dicho caso.
	Submeta: DistanciaRecorrida

Tabla 80: Librería de tareas

Librería de metas	
LlegarA	**Descripción:** El robot ha alcanzado un punto determinado del mapa.
	Predicado: Esta_En (Movil, Lugar)
DetectarAlgo	**Descripción:** El robot se encuentra frente a un elemento del entorno a una distancia mínima pero segura.
	Predicado: Detectado (Movil, Obstaculo)
PuntoEnfrente	**Descripción:** El robot se ha situado frente a un punto determinado del mapa y un avance rectilíneo le permite llegar a él sin realizar otra acción.
	Predicado: Esta_En (Movil, Lugar) y Conectado (Lugar, Lugar2)
SuperarChoque	**Descripción:** El robot ha fallado previamente a la hora de alcanzar otra meta y ahora está en disposición de intentarlo de nuevo.
	Predicado: Colision (Movil, Obstaculo)
MismaDistancia	**Descripción:** El robot se ha situado en el centro de un pasillo o zona del entorno con la misma distancia a cada una de las paredes que delimitan dicha zona.
	Predicado: En_Medio (Movil, Lugar, Obstaculo1, Obstaculo2)
EvitarChoque	**Descripción:** El robot ha dejado atrás un obstáculo en su camino con el que podría haber chocado, pudiendo ahora continuar con su trayectoria.
	Predicado: Libre (Movil)
DistanciaRecorrida	**Descripción:** El robot ha avanzado en línea recta hasta alcanzar un número de metros determinado.
	Predicado: ---

Tabla 81: Librería de metas

Librería de elementos	
Lugar	**Ejemplos:** "Robot", "Robot1", "Robot2"...
Movil	**Ejemplos:** "Nodo1", "Nodo2", "Habitación1", "Pasillo1"...
Obstaculo	**Ejemplos:** "Pared1", "Pared2", "Caja1", "Obstaculo1"

Tabla 82: Librería de elementos del planificador

Librería de predicados	
Esta_En	**Sintaxis:** Esta_En (Movil, Lugar)
	Descipción: El objeto móvil *Movil* está situado en el lugar *Lugar*.
Se_Encuentra	**Sintaxis:** Se_Encuentra (Obstaculo, Lugar)
	Descipción: El objeto *Obstaculo* se encuentra en el lugar *Lugar*.
En_Medio	**Sintaxis:** En_Medio (Movil, Lugar, Obstaculo1, Obstaculo2)
	Descipción: El objeto móvil *Movil* se encuentra en el lugar *Lugar*, a la misma distancia del objeto *Obstaculo1* y el objeto *Obstaculo2*.
Conectado	**Sintaxis:** Conectado (Lugar1, Lugar2)
	Descipción: Existe un arco en el mapa entre los lugares *Lugar1* y *Lugar2*.
Detectado	**Sintaxis:** Detectado (Movil, Obstaculo)
	Descipción: El objeto móvil *Movil* está detectando enfrente a una distancia significativa el objeto *Obstaculo*.
Libre	**Sintaxis:** Libre (Movil)
	Descipción: El objeto móvil *Movil* no está detectando frente a sí ningún tipo de objeto.
Orientado	**Sintaxis:** Orientado (Movil, Lugar)
	Descipción: El objeto móvil *Movil* se encuentra orientado hacia el lugar *Lugar*, pudiendo llegar a él con un movimiento rectilíneo.
Colision	**Sintaxis:** Colision (Movil, Obstaculo)
	Descipción: El objeto móvil *Movil* ha chocado con el obstáculo Obstaculo (produciéndose el contacto).

Tabla 83: Librería de predicados del planificador

Librería de acciones	
MantenerDireccion	**Sintaxis:** MantenerDireccion (Movil, Lugar1, Lugar2)
	Precondiciones: • Esta_En (Movil, Lugar1) • Conectado (Lugar1, Lugar2) • Orientado (Movil, Lugar2) • Libre (Movil)
	Postcondiciones: • - Esta_En (Movil, Lugar1) • - Orientado (Movil, Lugar2) • + Esta_En (Movil, Lugar2)

Orientarse	**Sintaxis:** Orientarse (Movil, Lugar1, Lugar2)
	Precondiciones: • Esta_En (Movil, Lugar1) • Conectado (Lugar1, Lugar2)
	Postcondiciones: • + Orientado (Movil, Lugar2)
EncontrarAlgo	**Sintaxis:** EncontrarAlgo (Movil, Lugar, Obstaculo)
	Precondiciones: • Esta_En (Movil, Lugar) • Se_Encuentra (Obstaculo, Lugar) • Libre (Movil)
	Postcondiciones: • - Libre (Movil) • + Detectado (Movil, Obstaculo)
Retroceder	**Sintaxis:** Retroceder (Movil, Lugar, Obstaculo)
	Precondiciones: • Esta_En (Movil, Lugar) • Se_Encuentra (Obstaculo, Lugar) • Detectado (Movil, Obstaculo) • Colision(Movil, Obstaculo)
	Postcondiciones: • - Detectado (Movil, Obstaculo) • - Colision (Movil, Obstaculo) • + Libre (Movil)
SituarseMedio	**Sintaxis:** SituarseMedio (Movil, Lugar, Obstaculo1, Obstaculo2)
	Precondiciones: • Esta_En (Movil, Lugar) • Se_Encuentra (Obstaculo1, Lugar) • Se_Encuentra (Obstaculo2, Lugar)
	Postcondiciones: • + En_Medio (Movil, Lugar, Obstaculo1, Obstaculo2)
Rodear	**Sintaxis:** Rodear(Movil, Lugar, Obstaculo)
	Precondiciones: • Esta_En (Movil, Lugar) • Se_Encuentra (Obstaculo, Lugar) • Detectado (Movil, Obstaculo)
	Postcondiciones: • - Detectado (Movil, Obstaculo) • + Libre (Movil)

Tabla 84: Librería de acciones del planificador

Anexo G

Proyecto ROSETTA

ROSETTA es un proyecto llevado a cabo por la Unión Europea para la creación de tecnología por robots controlados por una ejecución de tareas en interacción natural con humanos, basado en autonomía, conocimiento acumulativo y aprendizaje [ROSETTA website]. El desarrollo de esta tecnología va enfocada a robots industriales que no se limiten simplemente a parecerse a humanos, sino que también cooperen con trabajadores humanos en un entorno seguro y lo más natural posible. Estos robots son programados de una forma eficaz e intuitiva, haciéndolos así fácilmente adaptables a nuevas tareas cuando la línea de producción se cambia para un nuevo producto.

Los cuatro objetivos principales del proyecto ROSETTA son los siguientes:

- Permitir que los robots puedan ser utilizados en tareas complejas con alta flexibilidad y robustez.
- Facilitar el esfuerzo del despliegue de distribución para permitir un cambio rápido y eficiente entre la producción de dos productos.
- Facilitar el uso del sistema programable para acceder a la funcionalidad del robot ROSETTA sin la necesidad de grandes habilidades en programación de robots.
- Elaborar nuevos métodos sensoriales, de control y de toma de decisiones para una interacción robot-humano segura.

El proyecto ROSETTA se centra en desarrollar métodos de ingeniería y sistemas de programación robóticos en términos intuitivos y más relacionados con la tarea, en lugar de ser más específicos con la instalación. Esto requiere que los robots sean capaces de ejecutar tareas de forma más autónoma, sin la necesidad de una descripción detallada de cada paso que deben dar, lo que origina una notable reducción del esfuerzo de programación. Una vez programados, los robots hacen uso de un aprendizaje basado en sus propios sensores para mejorar sus habilidades de forma autónoma que les permita llevar a cabo sus tareas rápidamente, casi tanto como lo hace un trabajador humano. Cuando el robot consigue llegar a este nivel de optimización, comparte con otros robots su conocimiento adquirido sobre la mejor forma de llevar a cabo sus tareas, mandando sus parámetros a un servidor central mediante una red de comunicaciones. Otros robots hacen lo mismo, lo que va creando una base de conocimiento acumulativa.

Todo este proyecto pretende dar origen a un nuevo paradigma de la robótica, basado en la representación robótica de tareas y habilidades. Los programadores podrán ser capaces de construir una aplicación robótica utilizando instrucciones a nivel de tarea, incrustando conocimiento sobre sensores, planificación, control y toma de decisiones apropiados para la tarea. Estos resultados serán posteriormente implementados en un controlador y probados en un entorno de robots humanoides, utilizando un control adaptativo híbrido para su mejora automática y lograr una interacción en lenguaje natural.

Por último, ROSETTA también pretende alcanzar nuevos métodos sensoriales, de control y de toma de decisiones para una interacción física robot-humano segura. Esto incluye nuevas funcionalidades, que no se encuentran actualmente en muchos robots, las cuales están "*centradas en humanos*": el controlador tiene conocimiento sobre cómo puede dañar al humano y sobre cómo evitarlo, y está equipado con sensores y capacidades de procesamiento de dichos sensores y razonamiento sobre esa información para encontrar un equilibrio óptimo entre el progreso eficiente en sus tareas y una interacción con los humanos segura y psicológicamente aceptable. Parte del esfuerzo de ROSETTA es estandarizar este tipo de funcionalidades e interacción.