universidad de león

Escuela de Ingenierías I. I.

Industrial, Informática y Aeroespacial

MÁSTER EN INGENIERÍA INFORMÁTICA

Trabajo de Fin de Máster

MERLIN2: Sistema cognitivo para ROS 2

MERLIN2: Cognitive System for ROS 2

Autor: Miguel Ángel González Santamarta

Tutor: Francisco Javier Rodríguez Lera

Cotutor: Camino Fernández Llamas

(Febrero, 2022)

UNIVERSIDAD DE LEÓN
Escuela de Ingenierías I. I.
MÁSTER EN INGENIERÍA INFORMÁTICA
Trabajo de Fin de Máster

ALUMNO: Miguel Ángel González Santamarta

TUTOR: Francisco Javier Rodríguez Lera

COTUTOR: Camino Fernández Llamas

TÍTULO: MERLIN2: Sistema cognitivo para ROS 2

CONVOCATORIA: Febrero, 2022

RESUMEN:

Desplegar robots sociales que realicen tareas en entornos reales conlleva desarrollar software complejo dada la cantidad de información involucrada en la toma de decisiones de un robot. Uno de los mecanismos habituales para gestionar y modelar el conocimiento para su inclusión en el sistema de toma de decisiones de un robot es el uso de arquitecturas cognitivas. Este Trabajo de Fin de Máster (TFM) presenta el desarrollo de una arquitectura cognitiva híbrida para robots de servicio llamada MERLIN2. MERLIN2 es capaz de gestionar las mecanismos habituales de razonamiento y planificación que un robot asistencial lleva a cabo en entornos dinámicos. Por un lado, se presenta el diseño multicapa de la arquitectura que hace uso de conceptos clásicos como son los sistemas reactivos, de planificación y de gestión del conocimiento. Por otro lado, se implementa dicha arquitectura centrándose en el uso de diversos patrones de diseño software y utilizando el framework ROS 2. Por último, se presenta su aplicación real mediante demos sobre los robots RB1 y TiaGo dónde se validará su funcionamiento en entornos reales y simulados.

Palabras clave: Robótica, Robot Operating System (ROS), ROS2, PDDL, Arquitecturas cognitivas, Planificación, Máquinas de estados, Inteligencia Artificial, Python

UNIVERSIDAD DE LEÓN
Escuela de Ingenierías I. I.
MÁSTER EN INGENIERÍA INFORMÁTICA
Trabajo de Fin de Máster

STUDENT: Miguel Ángel González Santamarta

TUTOR: Francisco Javier Rodríguez Lera

COTUTOR: Camino Fernández Llamas

TITLE: MERLIN2: Cognitive System for ROS 2

CALL: February, 2022

ABSTRACT:

Deploying social robots that perform tasks in real environments entails developing complex software given the amount of information involved in a robot's decision-making. One of the common mechanisms to manage and model knowledge for inclusion in a robot's decision-making system is the use of cognitive architectures. This work presents the development of a hybrid cognitive architecture for service robots called MERLIN2. MERLIN2 is capable of managing the usual reasoning and planning mechanisms that an assistive robot carries out in dynamic environments. On the one hand, the multilayer design of the architecture is presented. It makes use of classic concepts such as reactive, planning and knowledge management systems. On the other hand, this architecture is implemented focusing on the use of various software design patterns and using the ROS 2 framework. Finally, its real application is presented through demonstrations on the RB1 and TiaGo robots where their operation will be validated in real environments and simulated.

Keywords: Robotics, Robot Operating System (ROS), ROS2, PDDL, Cognitive Architectures, Planning, State Machine, Artificial Intelligence, Python

Índice general

Índice de figuras

Índice de tablas

Glosario de términos

Abstract Factory Pattern : Patrón de diseño software creacional que se basa en una clase abstracta con unas interfaces comunes para crear objetos de una determinada familia. Mediante la clase abstracta, se pueden crear factorías concretas que creen los objetos relacionados entre sí y pertenecientes a una misma familia.

Blackboard Pattern : Patrón de diseño software de comportamiento que se basa en la integración de varios módulos. Se tiene un componente que se encarga de controlar la ejecución de los módulos. Además, se utiliza una estructura global de memoria que contiene datos de que generan los módulos.

Chain of Responsibility Pattern : Patrón de diseño software de comportamiento que se basa en usar una cadena de manejadores que procesan información. La entrada de un manejador es la salida del manejador anterior.

Data Access Object Pattern (DAO) : Patrón de diseño software que se basa en separar la lógica de acceso a datos de los objetos de negocio. El DAO encapsula toda la lógica de acceso a los datos en clases que el resto de la aplicación puedan usar.

Data Transfer Object Pattern (DTO) : Patrón de diseño software que se basa en clases planas con atributos y métodos para acceder y editar los atributos.

Facade Pattern : Patrón de diseño software estructural basado en el uso de una clase que actúa como interfaz de un subsistema mucho más complejo. Puede que la clase fachada no tenga toda la funcionalidad del subsistema pero tendrá las características que los clientes necesitan.

Factoy Method Pattern : Patrón de diseño software creacional basado en el uso de una clase que se encarga de crear nuevos objetos sin especificar su clase. Mediante este patrón se consigue que el cliente que quiere crear esos objetos no tenga que gestionar la creación.

Framework : Marco que proporciona ciertas interfaces y bases para desarrollar aplicaciones software destinadas a una plataforma específica.

Gazebo : Herramienta de simulación 3D utilizada en la creación de entornos y robots. Está integrado con ROS 2. [3]

GitLab : Servicio de gestión de versiones basado en Git para el desarrollo de software. [4]

JSGF (*JSpeech Grammar Format*): Formato utilizado para definir gramáticas usadas en el reconocimiento del habla. Estas gramáticas están formadas por reglas que definen las posibles frases que se pueden dar. [5]

MongoDB : Base de datos no relacional, distribuida, basada en documentos y de uso general. [6]

mongoengine : DOM (Document-Object Mapper) para trabar con MongoDB desde Python. [7]

pytest : Framework para crear y ejecutar tests de Python. [8]

PDDL (*Planning Domain Definition Language*): Lenguaje para codificar las tareas en la planificación clásica. Los componentes de este lenguaje son: objetos, las cosas que hay en el mundo; predicados, las propiedades de los objetos que pueden ser verdaderas o falsas; acciones, las diferentes formas de alterar el mundo; y los objetivos, predicados que se quiere que sean verdaderos. [9]

Pylint : Herramienta software para verificar el código Python buscando errores y comprobando el estilo con el que está escrito. [10]

Python : Lenguaje de programación de propósito general. Es un lenguaje multiplataforma, se puede usar en diferentes sistemas informáticos. Además, es un lenguaje interpretado, es decir, el usuario no tiene que compilar el código para ejecutarlo. Finalmente, es multiparadigma ya que se puede usar en programación imperativa, orientada objetos y funcional. [11]

Robot : Sistema formado por sensores, actuadores y una unidad de control que se puede programar para que interactúe con el entorno.

ROS (*Robot Operating System*): Framework formado por una colección de librerías destinadas al desarrollo de software para robots. [12]

ROS 2 Action (*Acción de ROS 2*): Sistema de comunicación de ROS 2 basado en el uso de un servidor y un cliente. Al igual que los servicios de ROS, el servidor está contenido en un nodo que realizará una tarea y el cliente se ejecutará en otro

nodo que demandará esa tarea. Además, el cliente puede cancelar la ejecución del servidor. La comunicación entre el servidor y el cliente se hace mediante mensajes de petición, respuesta y progreso del servidor. [12]

ROS 2 Message (Mensaje de ROS 2): Tipo de datos usado por ROS para realizar las comunicaciones entre nodos usando topics, servicios y acciones. [12]

ROS 2 Node (Nodo de ROS 2): Unidad de ejecución de ROS 2 organizado en paquetes. Los nodos pueden usar topics, servicios y acciones para comunicarse con otros nodos. [12]

ROS 2 Package (Paquete ROS 2): Unidad para organizar el código desarrollado con ROS 2. Un paquete puede contener nodos, ejecutables, scripts y otros tipos de archivos. [12]

ROS 2 Service (*Servicio de ROS 2*): Sistema de comunicación de ROS 2 basado en el uso de un servidor y un cliente. El servidor está contenido en un nodo que realizará una tarea y el cliente se ejecuta en otro nodo que demandará esa tarea. La comunicación entre el servidor y el cliente se hace mediante mensajes de petición y respuesta. [12]

ROS 2 Topic : Canal de comunicación de ROS 2 usado por los nodos para intercambiar mensajes. Un nodo puede publicar mensajes en un topic o suscribirse para leerlos. [12]

Speech-to-Text (STT) : Tecnología empleada para convertir audio en texto.

State Pattern : Patrón de diseño de comportamiento que permite a un objeto modificar su comportamiento cuando su estado cambia.

Strategy Pattern : Patrón de diseño software de comportamiento que permite definir una serie de algoritmos de la misma familia encapsulando cada uno de ellos en una clase intercambiable.

Text-to-Speech (TTS) : Tecnología empleada para convertir texto en audio.

unittest : Framework para crear tests de Python. [13]

YAML : Lenguaje de serialización de datos legible por humanos. [14]

Introducción

La robótica es una rama que abarca varias disciplinas de la ingeniería, como son la eléctrica, la mecánica, la electrónica o la informática entre otras. Es la encargada del desarrollo, mantenimiento y control de robots. Los robots son sistemas formados por sensores, actuadores y un sistema de control. De esta forma, el software encargado del control del comportamiento necesitará usar esos elementos para llevar a cabo unas tareas determinadas.

A lo largo de la historia se han diseñado y desarrollado varios enfoques en el campo de la gestión del comportamiento de los robots. Algunos de los paradigmas más populares han sido los sistemas deliberativos, las arquitecturas de tres capas o las arquitectas reactivas. El objetivo de todos ellos es implementar un framework para llevar a cabo la gestión y control de un robot.

Si se aplica la ingeniería del software a los paradigmas anteriormente mencionados, los componentes que los forman se convierten en nodos que utilizan mecanismos síncronos y asíncronos para comunicarse. Sin embargo, el problema se centra en el tratamiento de las condiciones cambiantes del entorno. De esta forma, para tratar este problema, se necesita una solución híbrida que combine varios paradigmas. Además, hay que tener en cuenta las características del robot.

Este proyecto presenta la creación de la arquitectura híbrida MERLIN 2 (MachinEd Ros 2 pLannINg) que se encarga de gestionar y controlar el funcionamiento de un robot. MERLIN2 consiste en la evolución y migración de MERLIN [15] a ROS 2, la nueva versión de ROS [12], y Python3 [11]. Para hacer esto, se han revisado los componentes que forman la arquitectura y se ha evaluado si se pueden migrar o si hay que crear nuevos componentes para reemplazarlos. Además, se quiere prestar especial atención a las principales herramientas utilizadas para crear MERLIN: ROSPlan [16], el framework más popular para realizar planificación en ROS, y SMACH [17], una librería de Python para desarrollar comportamientos mediante máquinas de estados.

Por último, el desarrollo de MERLIN2 se basa en la creación de un nuevo sistema de gestión del conocimiento basado en PDDL [9], el lenguaje más extendido en robótica para realizar planificación; y en el uso de patrones de diseño software. Por este motivo, es necesario reemplazar ROSPlan con un sistema que se encarguen de realizar la planificación de la arquitectura usando los nuevos componentes. Además, como SMACH no está disponible para ROS 2, se ha decidido crear una nueva librería para facilitar la creación de máquinas de estados para desarrollar comportamientos.

Planteamiento del problema

El problema que se quiere resolver es cómo modelar comportamientos en robots de servicio. De esta manera, se propone desarrollar una arquitectura software para robots que gestionará y controlará su comportamiento. Se construirá basándose en la arquitectura MERLIN, ROS 2 y Python3.

La arquitectura necesitará un sistema deliberativo que genere planes empleando conocimiento simbólico. Por este motivo, se pretende crear un nuevo sistema de gestión del conocimiento del robot que permita acceder, actualizar, crear y eliminar el conocimiento de un robot desde cualquier componente de la arquitectura.

Por último, se necesitarán comportamientos a corto plazo. Para ello se pretenden emplear máquinas de estados similares a las que se podían construir con SMACH.

Objetivos

El objetivo principal de este proyecto consiste en desarrollar MERLIN2, una arquitectura para gestionar y controlar la resolución de las misiones de un robot. Para ello, es necesario estudiar y comprender las tecnologías necesarias. De esta forma, los objetivos de este proyecto son los siguientes:

- **Objetivo 1** *Familiarizarse y comprender las tecnologías: para llevar a cabo el proyecto es necesario comprender ROS 2, sus nuevas funciones y mecanismos. Además, también es necesario revisar SMACH, la librería para desarrollar máquinas de estados. Por último, es necesario aprender a modelar documentos de MongoDB desde Python, especialmente los documentos que representarán los elementos PDDL.*

- **Objetivo 2** *Diseñar KANT, YASMIN y MERLIN2: el diseño de la arquitectura se centra en definir sus componentes. Además, el diseño se realizará a diferentes niveles siendo el de más alto nivel el diseño de los componentes generales de MERLIN2 y el de bajo nivel el diseño UML basado en diagramas de clases y en el uso de patrones de diseño software.*
- **Objetivo 3** *Desarrollar KANT (Knowledge mAnagemeNT): KANT es un nuevo sistema de gestión del conocimiento. Gracias a él, cualquier elemento de MERLIN2 podrá acceder al conocimiento del robot. Además, se crearán dos tipos de almacenamiento para el conocimiento: uno basado en MondoDB y otro basado en ROS 2.*
- **Objetivo 4** *Desarrollar YASMIN (Yet Another State MachINe): YASMIN consiste en una librería para Python3 que permite desarrollar comportamientos mediante la creación de máquinas de estados. Está basada en SMACH aunque busca ser una librería más simple. Además, permite integrarse fácilmente en ROS 2 gracias a la utilización de estados que encapsulan ciertos mecanismos de comunicación de ROS 2.*
- **Objetivo 5** *Desarrollo de MERLIN2 (Machined Ros2 pLanINg): este objetivo consiste en desarrollar la nueva arquitectura partiendo de la vieja versión MERLIN. Para ello, se desarrollarán las cuatro capas que la formaban: Mission layer, Planning layer, Executive layer y Reactive layer. En esta última capa se pretenden desarrollar los sistemas de reconocimiento del habla, de síntesis de voz y de navegación topológica.*
- **Objetivo 6** *Realizar pruebas: se quieren realizar pruebas en entornos reales y simulados. Además, se quiere repetir la prueba descrita en [15] con MERLIN2 para obtener métricas que permitan realizar comparaciones con MERLIN.*

Metodología

La metodología usada en este proyecto es Scrumban [18]. Es una metodología de desarrollo ágil que surge al combinar las metodologías Scrum y Kanban. De esta forma, se combinan la estructura de Scrum con el control y la visualización de los flujos de trabajo de Kanban. Las principales características de Scrumban son las siguientes:

- Reuniones: al igual que en Scrum, se realizan reuniones. Especialmente, se hacen reuniones de planificación bajo demanda en las que se deciden qué tareas están lista para ser empezadas.

- Iteraciones: al igual que en Scrum, se realizan iteraciones en las que se realiza el desarrollo del proyecto.
- Tablero: al igual que en Kanban y Scrum, se tiene un tablero. El tablero empleado se presenta en la figura 1. Para crear el tablero se ha utilizado la opción Board de GitLab [4], que permite crear un tablero con categorías personalizadas en las que se pueden añadir tareas. El tablero está formado por las siguientes categorías:
  - Backlog: en esta categoría se tienen las tareas que se está considerando realizar.
  - To Do: en esta categoría se tienen las tareas que se van a efectuar y que aún no se han comenzado.
  - Doing: en esta categoría se tienen las tareas que se están efectuando.
  - Testing: en esta categoría se tienen las tareas que se están probando.
  - Done: en esta categoría se tienen las tareas que se han terminado.

  Por último, al igual que en Kanban, se puede visualizar el estado actual del desarrollo y las tareas que se quieren llevar a cabo más adelante. Además, se puede limitar el trabajo en progreso (Work In Progress o WIP) consiguiendo mejorar el flujo para optimizar el tiempo medio para completar las tareas.

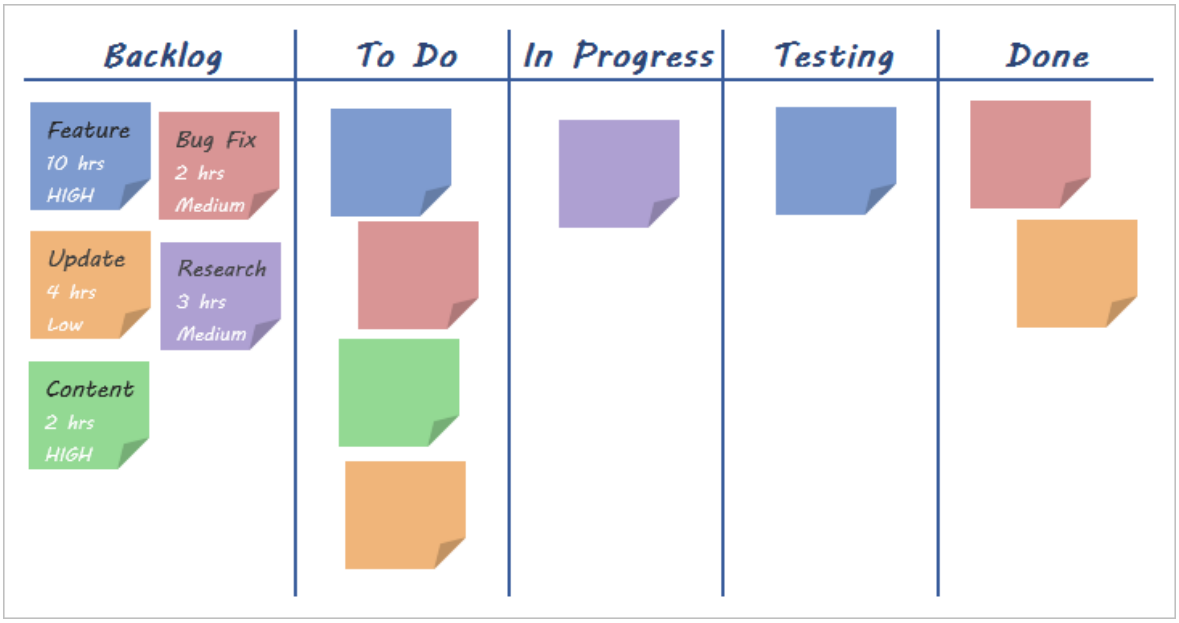


Figura 1: Tablero Scrumban
Fuente: "Kanban Board Template." vertex42.com.
`https://www.vertex42.com/ExcelTemplates/agile-kanban-board.html`.

Estructura del trabajo

Este documento está dividido en 5 capítulos y 3 anexos. A continuación se da una descripción breve de cada uno:

- Capítulo 1: en este capítulo se presenta el contexto en el que se realiza el trabajo. Se hace una revisión de las tecnologías, plataformas, herramientas y trabajos previos.
- Capítulo 2: en este capítulo se presenta la gestión del proyecto. Se incluye la realización de presupuestos, definición de tareas y gestión de recursos.
- Capítulo 3: en este capítulo se presenta la solución que se desarrolla en el proyecto.
- Capítulo 4: en este capítulo se muestra la evaluación de la solución llevada a cabo en este proyecto.
- Capítulo 5: en este capítulo se presentan las conclusiones que se han obtenido de los resultados de este trabajo.
- Anexo A: este anexo contiene la información sobre el control de versiones que se ha utilizado para llevar a cabo el proyecto.
- Anexo B: este anexo trata el seguimiento del proyecto.
- Anexo C: este anexo contiene los diagramas UML de clases del proyecto software.

Capítulo 1

Estudio del problema

En este capítulo se presenta el contexto en el que se realiza el trabajo. Se hace una revisión de las tecnologías, plataformas, herramientas y trabajos previos.

1.1. El contexto del problema

Este proyecto se ha desarrollado en el Grupo de Robótica de la Universidad de León. En el grupo se dispone de diferentes plataformas, es decir, diferentes robots. Con ellos se puede trabajar y realizar pruebas en entornos reales. Las aplicaciones que se desarrollan en el grupo suelen tratar la interacción de los robots con humanos y el entorno y la generación de comportamientos autónomos. De esta forma, este proyecto se centra en el desarrollo de una arquitectura software llamada MERLIN2 para controlar el comportamiento de un robot. Este desarrollo se centra principalmente en el uso de ROS 2, la nueva versión de ROS [12].

1.1.1. ROS 2

El contexto software de este proyecto parte de ROS (Robot Operating System). ROS es un framework formado por una colección de librerías para desarrollar software para robots. De esta forma, proporciona librerías y herramientas que ayudan al desarrollo de aplicaciones de robots. Sus aportaciones son la abstracción del hardware, los controladores para el hardware, los visualizadores de información, la gestión de paquetes y mucho más. Además, hay dos lenguajes de programación con los que se pueden crear aplicaciones basadas en ROS, que son Python y C++.

Por otro lado, ROS 2 es la nueva versión de ROS. Conserva la gran mayoría de conceptos de ROS, sin embargo, los sistemas de comunicación han sufrido varios cambios que es necesario comprender para poder utilizarlos. El uso de ROS 2 produce un sistema distribuido formado por procesos llamados nodos que trabajan de forma asíncrona y se comunican mediante los sistemas de comunicación proporcionados. De esta manera, una aplicación de ROS 2 estaría formada por uno o más nodos comunicados entre sí. Los nodos se organizan en paquetes y los archivos que se usan para ejecutarlos de manera conjunta se llaman launch.

1.1.2. Comunicaciones de ROS 2

Las comunicaciones de ROS 2 se basan en el uso de Data Distribution Service (DDS) [19]. DDS proporciona un estándar para comunicar máquinas mediante el intercambio de datos usando el patrón de publicación-subscripción. De esta forma, ROS 2 tiene varios mecanismos para comunicar nodos:

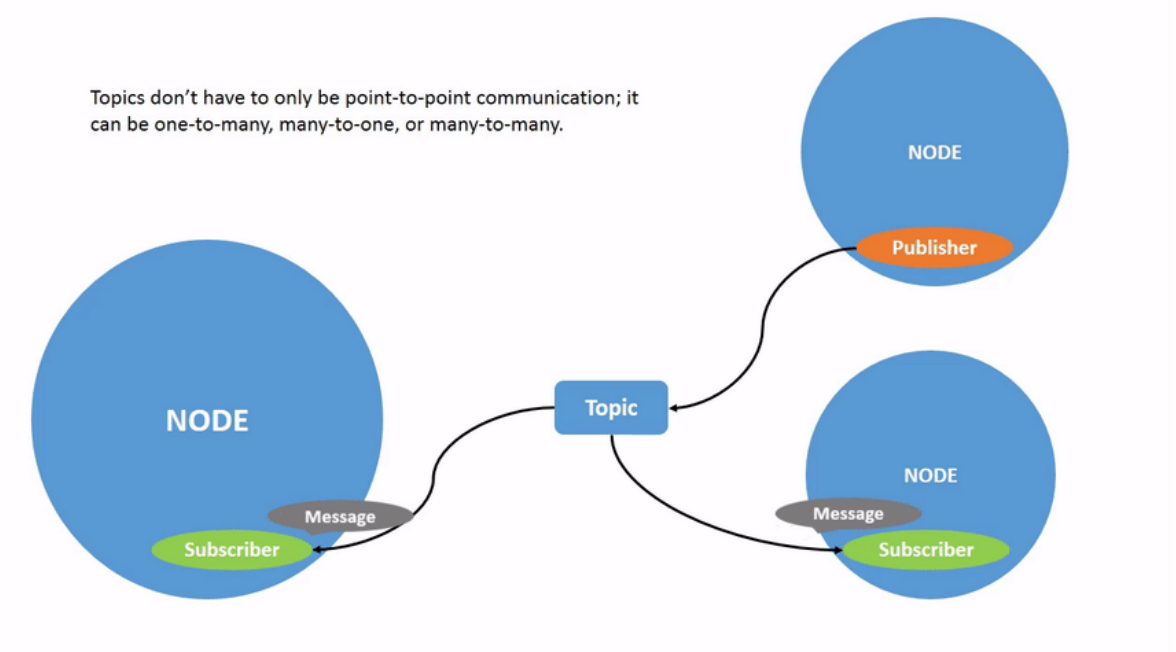


Figura 1.1: Ejemplo de topics de ROS 2
Fuente:
https://index.ros.org/doc/ros2/Tutorials/Topics/Understanding-ROS2-Topics/

- **Topics**: los topics son canales de comunicación que varios nodos pueden usar. El uso de topics se centra en leer o escribir información, sin prestar atención a qué nodos los están empleando. En un topic puede haber varios nodos que generen información (publishers) y varios nodos que lean información (subscribers). Los

nodos que leen información utilizan callbacks, funciones que se ejecutan cada vez que se reciben datos. En la figura 1.1 se presenta un ejemplo gráfico del uso de topics en ROS 2. Para utilizar un topic es necesario usar un tipo de mensaje que llevará la información. Este mensaje se puede crear o se puede elegir uno de los que ROS 2 tiene por defecto.

- **Services**: un servicio es un sistema de comunicación basado en la creación de servidores y clientes que se comunican mediante mensajes de petición y respuesta. Los servicios se usan para crear comunicaciones entre nodos parecidas a las llamadas a procedimientos remotos. En ROS 2, cuando un cliente envía una petición a un servidor, el servidor obtiene los datos de la petición, ejecuta su callback y envía la respuesta al cliente. El cliente puede esperar por la respuesta de forma síncrona, bloqueando la ejecución del hilo; o de forma asíncrona, continuando la ejecución del hilo y tratando la respuesta en un callback cuando se reciba. Para utilizar servicios hacen falta dos mensajes: la petición y la respuesta. Un ejemplo del uso de un servicio de ROS 2 se presenta en la figura 1.2.

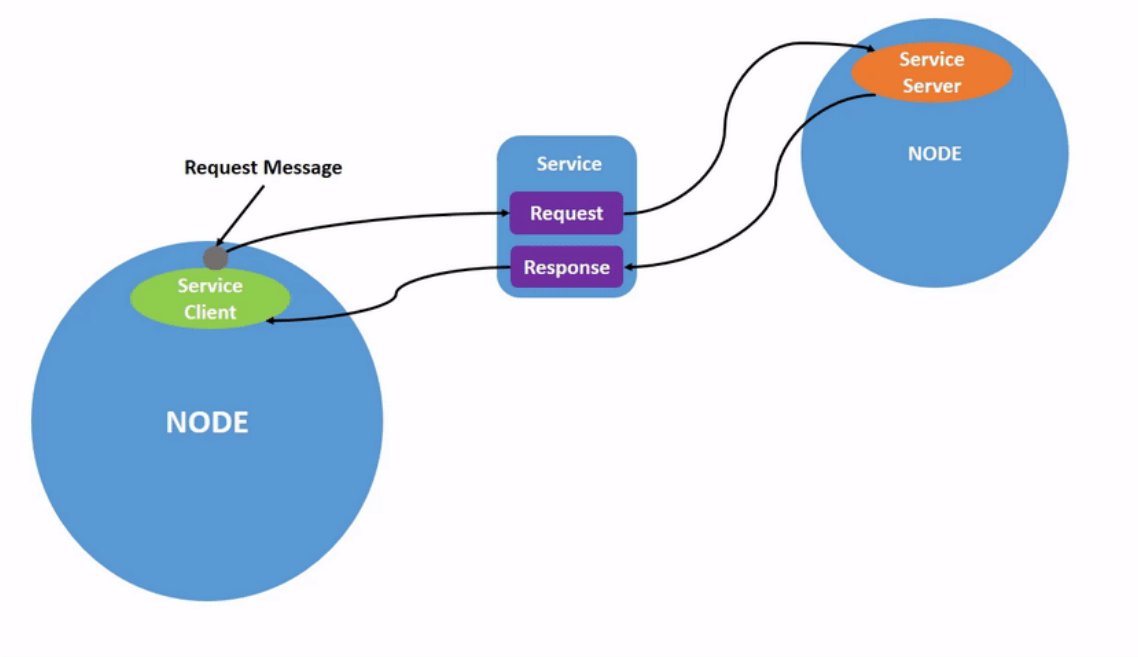


Figura 1.2: Ejemplo de servicios de ROS 2
Fuente:
https://index.ros.org/doc/ros2/Tutorials/Services/Understanding-ROS2-Services/

- **Action**: las acciones de ROS 2 se basan en usar servicios y topics. La principal diferencia entre las acciones y los servicios de ROS 2 es que se tiene un mayor control sobre la ejecución de las acciones. Esto se debe a que tienen la capacidad de cancelarse en cualquier momento. Para utilizar acciones hacen falta tres

mensajes: el objetivo o goal, el resultado y el progreso del servidor o feedback. Cuando el servidor de una acción recibe un goal de un cliente, activa un callback para decidir si aceptarlo o no. Este resultado es enviado al cliente. Después, si el servidor lo ha aceptado, se ejecuta su callback principal con los datos que el cliente envió en el goal. La aceptación y ejecución de los goals se realiza mediante servicios de ROS 2. Por otro lado, mientras se ejecuta el callback del servidor, se puede enviar feedback que será publicado en un topic. Por último, si todo salió bien, el servidor envía al cliente el resultado que ha obtenido. El esquema de comunicaciones entre el cliente y el servidor se presenta en la figura 1.3.

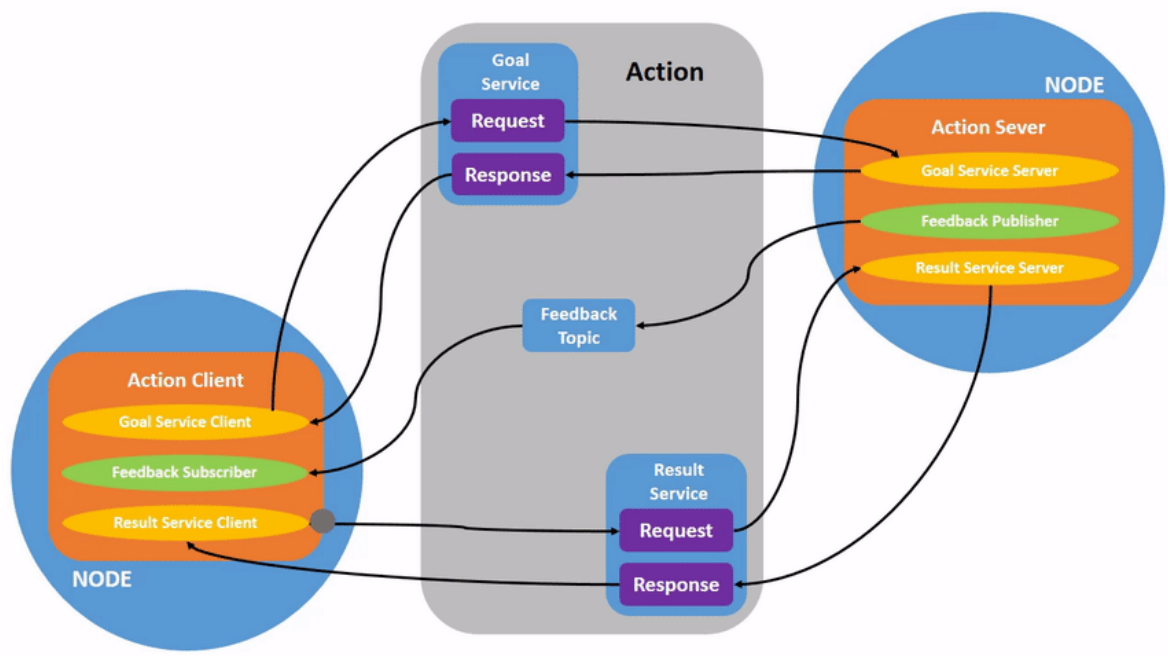


Figura 1.3: Ejemplo de Action de ROS 2
Fuente: *https://www.mathworks.com/help/ros/ug/rosaction_communication.png*

1.1.3. Gestión de tareas en ROS 2

En ROS 2 se generan tareas cada vez que un subscriber, un cliente o servidor de servicio o de acción reciben mensajes. Todas estas tareas se basan en los callbacks que van a tratar esos mensajes. De esta forma, cada vez que se genera una tarea, se encola la ejecución de su callback en la cola de callbacks.

Para decidir si la siguiente tarea se ejecuta, ROS 2 utiliza ejecutores (Executors). Los ejecutores son componentes software que controlan el sistema de threading usado para procesar callbacks. Existen dos tipos ejecutores en ROS 2:

- **SingleThreadedExecutor**: solo hay un thread que ejecuta todos los callbacks. Es el Executor por defecto en ROS 2.
- **MultiThreadedExecutor**: ejecuta los callbacks en un pool de threads.

Por otro lado, los callbacks se pueden agrupar por grupos (Callback Groups). Un grupo de callbacks controla cuándo se permite la ejecución de los callbacks. Se tienen dos tipos de grupos en ROS 2:

- **MutuallyExclusiveCallbackGroup**: solamente puede haber un callback ejecutándose a la vez. Es el grupo empleado por defecto en los nodos de ROS 2.
- **ReentrantCallbackGroup**: puede haber varios callbacks ejecutándose en paralelo.

Por último, los nodos están continuamente esperando por tareas. En ROS 2 a esto se le llama spin. Cuando llegan tareas, se utilizan estos ejecutores y grupos de callback para gestionarlas.

1.2. El estado de la cuestión

En esta sección se trata el estado de la cuestión dividido en dos partes: las arquitecturas precursoras y las arquitecturas modernas.

1.2.1. Arquitecturas precursoras

En este apartado se presentan las primeras arquitecturas para robots que surgieron. La mayoría se desarrollaron en el siglo XX, comenzando en los años setenta.

AURA (Ronald C. Arkin et al.)

El desarrollo y creación de arquitecturas para controlar y gestionar las tareas de un robot comenzó en el siglo pasado. En los años setenta, Ronal C. Arkin comenzó el desarrollo de la que se podría considerar la primera arquitectura software para robots llamada Aura (Autonomous Robot Architecture). Aura era una arquitectura híbrida cuyo principal objetivo era gestionar la navegación de un robot. Se dice que es híbrida porque usaba un sistema deliberativo, basado en técnicas tradicionales de inteligencia artificial; y un sistema reactivo, encargado del control de los sensores y actuadores.

En la figura 1.4 se presenta el esquema de la arquitectura Aura. Sus componentes principales son los siguientes:

- Mission Planner: se encarga de comunicar el robot con el humano.
- Spatial Reasoner: se encarga de generar el plan, es decir, las acciones que realizará el robot.
- Plan Sequencer: es el encargado de ejecutar ese plan.
- Schema Controller: gestiona los actuadores y sensores del robot.

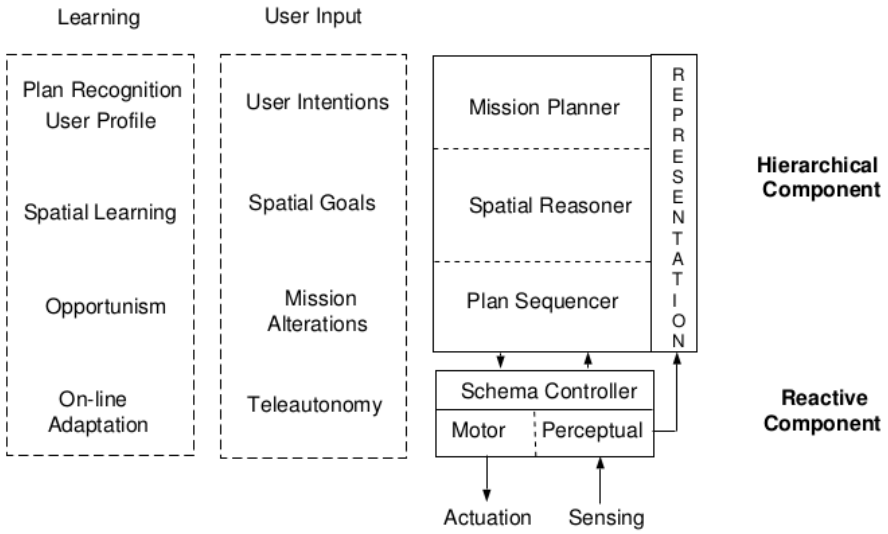


Figura 1.4: Aura (Autonomous Robot Arcuitecture)
Fuente: *[20]*

A pesar de que Aura solo se encargaba de gestionar la navegación de un robot, inició el desarrollo de arquitecturas híbridas y sirvió como base para crear las arquitecturas que surgieron después. Además, agrupaba un gran conjunto de campos de investigación como la planificación, el control reactivo y la inteligencia artificial.

Atlantis (Erann Gat)

El desarrollo de arquitecturas para controlar el comportamiento de robots se incrementó en los años noventa. En estos años destacó Erann Gat por el diseño y creación de las arquitecturas de tres capas [21]. Además, siguiendo este enfoque, desarrolló una arquitectura híbrida llamada Atlantis [22]. Atlantis era una arquitectura heterogénea y asíncrona orientada principalmente a robots móviles. De esta forma, en los estudios realizados por Gat, se concluyó que las arquitecturas software para robots tenían que

ser heterogéneas, es decir, se tienen que utilizar diferentes métodos de computación para realizar las tareas del robot; y asíncronas, para conseguir que las computaciones más lentas se ejecuten en paralelo evitando afectar al funcionamiento del resto de componentes. Por último, los componentes de Atlantis son:

- Controller: se encarga del control de las actividades primitivas que son los motores y sensores del robot.
- Sequencer: su misión es controlar la ejecución de las actividades primitivas y de la computación deliberativa.
- Deliberator: se encarga de la planificación y de mantener el conocimiento que tiene el robot del mundo.

DAMN (Julio K. Rosenblatt)

Otro enfoque que se desarrolló en los años noventa consiste en arquitecturas distribuidas. Un ejemplo es DAMN [23], que es una arquitectura distribuida para gestionar la navegación de robots. En la figura 1.5 se puede ver el esquema de DAMN. Los componentes de esta arquitectura son:

- Árbitro: genera las acciones del robot.
- Comportamientos: votan las acciones para decidir qué acciones se ejecutan.
- Manager: se encarga de definir qué votos tienen más peso dando más importancia a ciertos comportamientos.
- Controladores: se encargan de gestionar los motores.

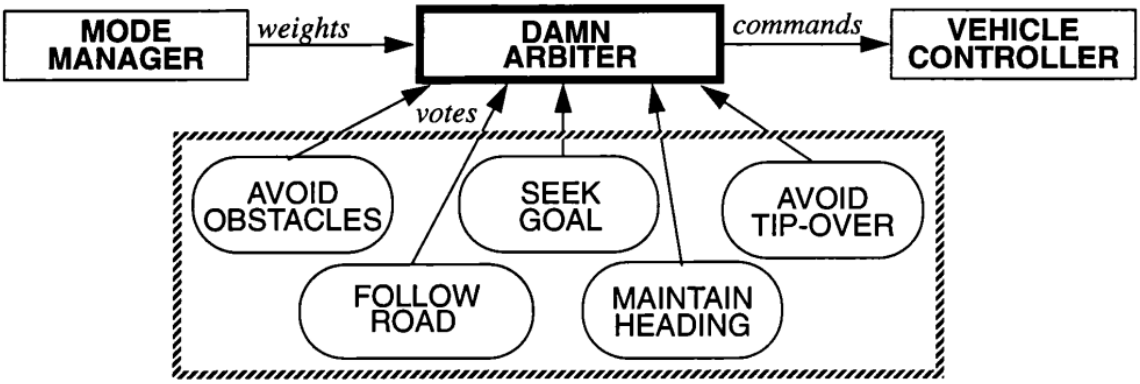


Figura 1.5: DAMN (Distributed Architecture for Mobile Navigation)
Fuente: *[23]*

Alliance (Lynne E. Parker)

En los años noventa también surgen las arquitecturas basadas en el uso de motivaciones para controlar el comportamiento de los robots. Un ejemplo de esto es la arquitectura Alliance [24]. Es una arquitectura distribuida que usa motivaciones para coordinar las actividades de varios grupos de robots. La secuencia de tareas que un robot realiza depende del comportamiento motivacional. Esto significa que las motivaciones activan los comportamientos que posteriormente votarán las acciones que se ejecutarán. De esta forma, Alliance está formada por un conjunto de motivaciones, un conjunto de comportamientos y un conjunto de acciones.

Robotic Systems Architectures and Programming (D.Kortenkamp and R. Simmons)

El modelo de arquitectura de tres capas es uno de los enfoques que más han impactado en los desarrollos posteriores. En estudios y libros recientes que tratan las arquitecturas software para robots, se usa el modelo de tres capas para explicar su desarrollo y funcionamiento. De esta forma, en 2008 David Kortenkamp y Reid Simmons usan este enfoque para explicar las arquitecturas [25]. En su capítulo sobre arquitecturas y programación, se estudian diferentes arquitecturas, entre ellas Aura, Atlantis y DAMN. Además, presentan los componentes que tendría una arquitectura, centrándose en el modelo de tres capas. Los componentes que proponen son los siguientes:

- Behavioral control: este componente se encarga del control de los sensores y actuadores del robot. Estaría formado por una serie de elementos llamados habilidades. Cada una de estas habilidades controlaría un sensor, un actuador o un conjunto de estos.
- Executive: este componente se encarga de traducir los planes de alto nivel en planes de bajo nivel.
- Planning: es el componente que se encarga de generar los planes de alto nivel. También se encarga de realizar la replanificación.

1.2.2. Arquitecturas modernas

En este apartado se presentan arquitecturas desarrolladas en el siglo XXI. Se muestran nuevos enfoques que antes no se habían presentado.

BICA (CARLOS R. Agüero et al.)

Una de las primeras arquitectas modernas es Behavior-based Iterative Component Architecture (BICA) [26]. Se centra en el desarrollo basado en componentes. De esta forma, el elemento principal de BICA es el componente, que es una máquina finita de estados. Un componente puede estar activo o inactivo, consiguiendo evitar el consumo de recursos de ciertos componentes. Además, un componente puede activar o desactivar otros componentes. Con todo esto, una aplicación hecha con BICA y destinada a ejecutarse en un robot será un conjunto de componentes que dependen unos de otros.

Integrating Classical Planning and Real Robots (Oscar Lima et al.)

En el paper de 2018 escrito por Oscar Lima y Rodrigo Ventura [1] se describe una arquitectura en la que se integra la planificación clásica. Para desarrollar esta arquitectura se usan varios componentes de ROSPlan [16], que es uno de los framework para realizar planificación en ROS más populares. Los componentes de esta arquitectura son:

- Base de conocimiento: se encarga de gestionar el conocimiento que estará escrito en PDDL [9]. Es un elemento de ROSPlan.
- Generador de problemas PDDL: se encarga de crear el problema en formato PDDL. Es un elemento de ROSPlan.
- Analizador de la base de conocimiento: se encarga de comprobar si hay objetivos por cumplir.
- Planificador: se encarga de generar el plan para alcanzar los objetivos.
- Validación del plan: se encarga de verificar si el plan resuelve los objetivos del problema.
- Traductor del plan: se encarga de traducir el plan producido para que la capa de ejecución pueda usarlo.
- Capa de ejecución: se encarga de ejecutar las acciones del plan.

Por último, como se muestra en la figura 1.6, la ejecución de esta arquitectura se presenta como una máquina de estados finita. Cada estado de la máquina se encarga de gestionar uno de los componentes de la arquitectura.

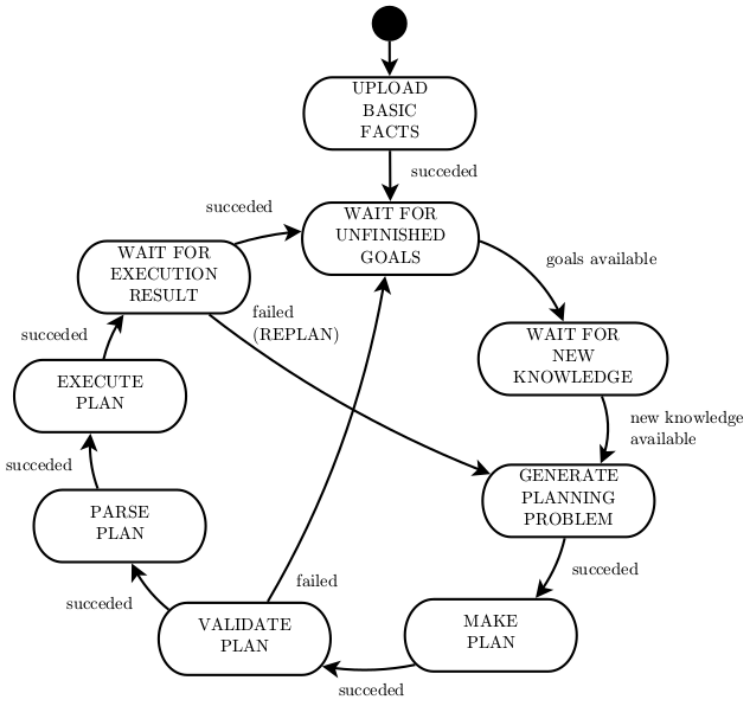


Figura 1.6: Máquina de estados de la arquitectura [1]
Fuente: *[1]*

HiMoP (Francisco J. Rodrígues-Lera et al.)

En 2018 se presentó HiMoP: A three-component architecture to create more human-acceptable social-assistive robots [27]. Es una arquitectura basada en motivaciones. Las motivaciones se usan para generar comportamientos más naturales para los humanos. Esto se consigue haciendo que el robot responda de forma dinámica a los cambios tanto de su entorno externo como interno. En la figura 1.7 se presenta el diagrama de esta arquitectura. Sus elementos son los siguientes:

- Jerarquía de necesidades: es un conjunto de necesidades con diferentes prioridades que se encargarán de activar las motivaciones. Además, hay un conjunto de roles para agrupar las diferentes necesidades.
- Motivaciones: las motivaciones se encargarían de iniciar y gestionar los comportamientos.
- Conjunto de máquinas de estados: las máquinas de estados se utilizan para originar los diferentes sistemas de la arquitectura. Se tienen los siguientes sistemas:
  - Sistema deliberativo: se encarga de la creación de los planes y de controlar la secuencia de acciones.

- Sistema reactivo: se encarga del control de los sensores y actuadores. Está preparado para adaptarse a los cambios del entorno.
- Sistema motivacional: es el sistema que gestiona las motivaciones descritas anteriormente.

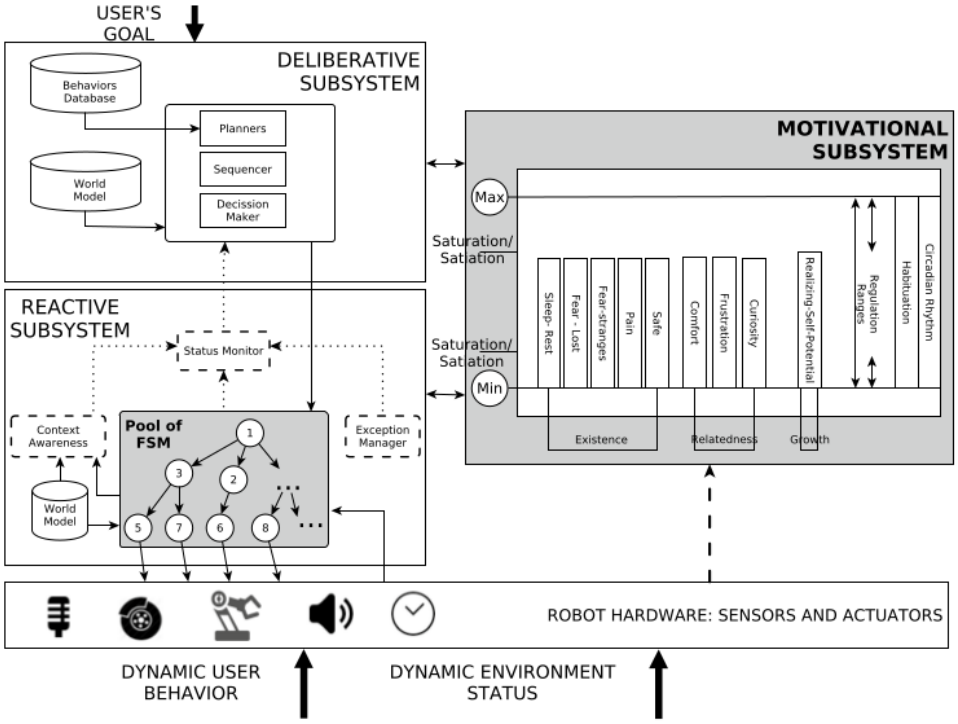


Figura 1.7: HiMoP
Fuente: *[27]*

Evolution of a Cognitive Architecture for Social Robots (Francisco Martín et al.)

En 2020 se presentó la evolución de la arquitectura BICA [28], una arquitecta basada en componentes. Esta nueva versión de BICA se presenta como una arquitectura compuesta por varias capas en la que se ha añadido un sistema de planificación. El cambio más significativo es el uso de máquinas de estados y árboles de comportamiento para crear los componentes de la arquitectura. Además, se ha desarrollado un sistema de memoria de trabajo en formato de grafo.

En la figura 1.8 se presentan las capas que forman esta arquitectura. En total hay cinco capas, que son las siguientes:

- Tier 1: se encarga de definir las misiones del robot a nivel simbólico usando PDDL.

- Tier 2: se encarga de realizar la planificación de la arquitectura.
- Tier 3: está formada por las acciones que tendrá el robot.
- Tier 4: está formada por los controladores de los sensores y actuadores.
- Tier 5: representa el hardware del robot.

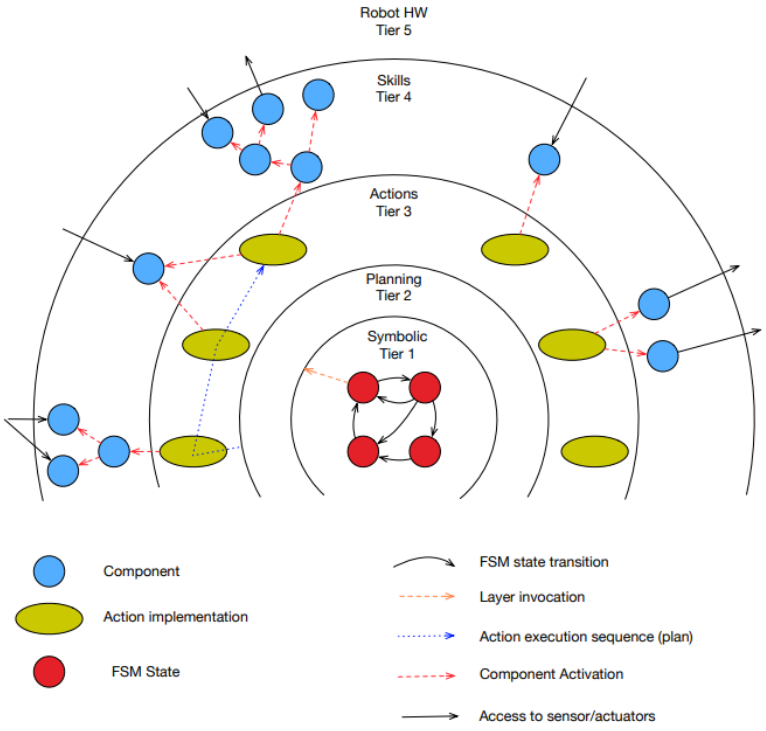


Figura 1.8: BICA
Fuente: *[28]*

El sistema de memoria de BICA se basa en el uso de un grafo cuyos nodos son los objetos que conoce el robot y cuyas aristas son las propiedades de los objetos, que pueden ser cadenas de texto o coordenadas. Para poder usar este sistema de memoria, se ha equipado el sistema de planificación con un mecanismo para sincronizar la información del grafo con el conocimiento que el robot tiene.

Por último, se ha añadido la utilización de árboles de comportamiento. Mediante el uso de estos árboles se consigue un mayor control del comportamiento. De esta forma, en la nueva versión de BICA, las acciones se pueden implementar como máquinas de estados o árboles de comportamiento.

DACOBOT (Guillaume Sarthou, et al.)

Una de las más recientes arquitecturas desarrolladas es DACOBOT (Deliberative Architecture for COllaborative roBOT) [29]. Esta arquitectura se ha usado para hacer que un robot lleve a cabo la Director Task [30] en el contexto de la interacción de robots con humanos. La Director Task consiste en el uso de la Teoría de la Mente en la comunicación y en la cognición social. La figura 1.9 presenta el diagrama de esta arquitectura.

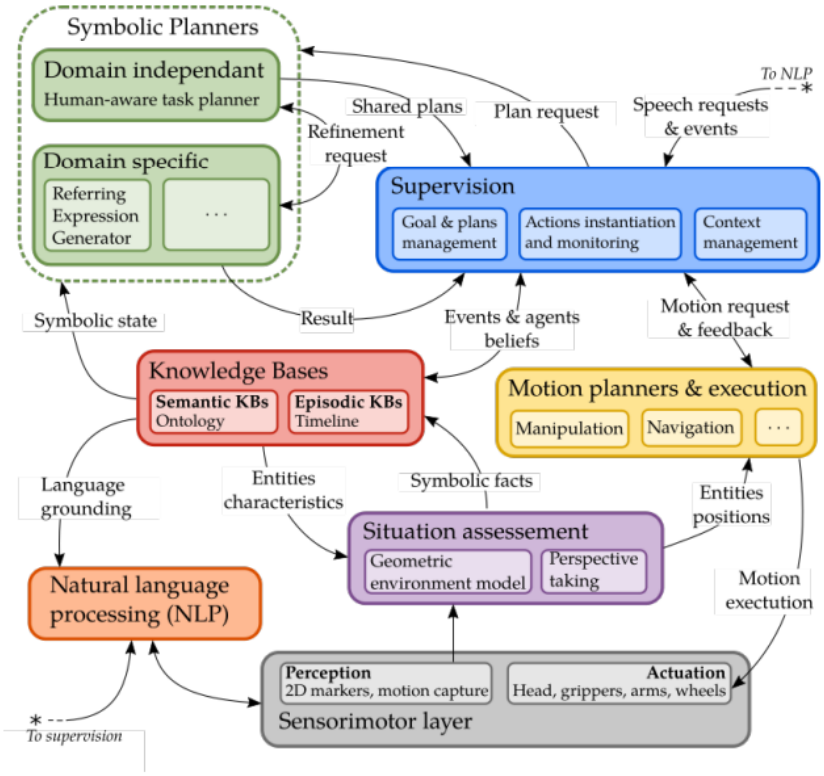


Figura 1.9: DACOBOT
Fuente: *[29]*

Como se puede ver, DACOBOT tiene los siguientes componentes:

- Knowledge Base: este componente se encarga de almacenar el conocimiento que tiene el robot sobre el mundo. Se han definido dos tipos de conocimiento: semántico y episódico. El conocimiento semántico representa el mundo, es decir, los objetos que existen, las propiedades y las acciones que se pueden realizar. Por otro lado, el conocimiento episódico sirve para mantener un historial de los cambios que sufre el conocimiento semántico.

- Symbolic Planners: este componente es el encargado de generar planes a partir del conocimiento del robot.
- Supervision: este componente se encarga de pedir planes y de gestionar las acciones de esos planes.
- Motion Planners & Execution: este componente se encarga de controlar los actuadores del robot para poder efectuar tareas de navegación y manipulación.
- Situation Assessment: este componente se encarga de generar representaciones geométricas a partir de los datos obtenidos de la Sensorimotor Layer. Después, esas representaciones se traducen a conocimiento simbólico que se puede almacenar en la base de conocimientos.
- Natural Language Processing: este componente se encarga de hacer el procesamiento del lenguaje natural. Los resultados obtenidos se podrán usar para generar nuevo conocimiento en el robot.
- Sensorimotor Layer: este componente se encarga de generar y agrupar los datos obtenidos de los sensores y los sistemas internos del robot.

1.2.3. Conclusión

En los trabajos presentados anteriormente se describen diferentes formas de diseñar e implementar arquitecturas software para robots. Se han presentado diferentes enfoques, esquemas y componentes. Sin embargo, aunque el inicio de las arquitecturas para robots comienza en el siglo XX, en la actualidad es un tema que aún no está del todo resuelto.

En todas las arquitecturas presentadas se pueden apreciar ciertos elementos similares. Un ejemplo de esto es el uso de sistemas deliberativos, lo que produce que se almacene el conocimiento que el robot tiene. Además, se usan sistemas reactivos para controlar los sensores y actuadores. La utilización de estos dos conceptos es lo que genera arquitecturas híbridas.

Por último, el modelo más extendido es el modelo en capas. Este modelo permite diferenciar y separar correctamente los diferentes sistemas de las arquitecturas. Además, se introducen ciertos conceptos en la creación de los componentes de las arquitecturas. Los más populares son las máquinas de estados y los árboles de comportamiento.

1.3. La definición del problema

El problema que se quiere resolver la gestión del comportamiento de robots de servicio. Para ello, se quiere diseñar, desarrollar y probar la arquitectura híbrida. Se parte de la arquitectura MERLIN [15].

La arquitectura necesitará un sistema de planificación similar a ROSPlan [16]. Además, será necesario un sistema para acceder y modificar el conocimiento que tiene el robot. Por este motivo se quiere desarrollar un sistema que encapsule el conocimiento en PDDL y que sea fácil de usar.

Por otro lado, la arquitectura estará basada en el uso de máquinas de estados. En MERLIN se empleaba la librería SMACH [17], sin embargo, esta librería no está disponible para ROS 2. Por este motivo se pretende desarrollar una nueva librería para desarrollar comportamientos mediante máquinas de estados.

Por último, el diseño de la arquitectura se basa en la utilización de diferentes patrones de diseño software que facilitarán su entendimiento, aportando flexibilidad y reutilización. Para desarrollarla se utilizará ROS 2 y Python3. Para probarla se realizarán tests unitarios de sus componentes y tests funcionales sobre la arquitectura total.

Capítulo 2

Gestión del proyecto software

En este capítulo se describen los aspectos relacionados con la gestión de proyecto. Se incluyen la planificación inicial del proyecto, el presupuesto y la gestión de recursos. Además, se describen las iteraciones realizadas siguiendo la metodología Scrumban presentada anteriormente.

2.1. Alcance del proyecto

El resultado final de este proyecto consiste en una arquitectura software para robots llamada MERLIN2 (Machined Ros2 pLannINg). Esta arquitectura se ha desarrollado principalmente en Python3. Además, está completamente integrada en ROS 2, que es la nueva versión de ROS (Robotic Operating System) [12], el framework estándar para desarrollar aplicaciones software para robots. Por último, la arquitectura tiene un diseño modular, lo que permite utilizar sus componentes de forma independiente y reemplazarlos de forma sencilla.

El uso de la arquitectura se basa en desarrollar nuevas aplicaciones basadas en los objetivos y acciones que tendrá el robot. De esta manera, primero se tiene que tratar el conocimiento que tendrá el robot, entre el que se incluyen los objetivos y las acciones. Para ello, se usará KANT (Knowledge mAnagemeNT). Después, las acciones se pueden implementar mediante máquinas de estados finitos utilizando YASMIN (Yet Another State MachINe), la nueva librería creada.

Por último, se quieren realizar varias pruebas de la arquitectura MERLIN2. Entre ellas, se quieren repetir los experimentos efectuados en [15]. Para ello, se quieren usar entornos virtuales simulados y reales.

2.2. Plan de trabajo

Se ha realizado una planificación inicial en la que se incluye el desarrollo de software. De esta forma, se parte de un plan basado en planificación tradicional. Posteriormente, se ha utilizado la metodología de desarrollo ágil Scrumban para gestionar el desarrollo del software. De esta forma, en esta sección se muestran las tareas que se han identificado, su estimación y su planificación.

2.2.1. Identificación de tareas

Las tareas que se han definido son las siguientes:

- **Tarea 1**: este grupo de tareas se centra en el estudio y comprensión de las nuevas tecnologías. A su vez, está formado por las siguientes tareas:
  - **Tarea 1.1**: Estudio de ROS 2. Esta es la tecnología más importante porque será la base sobre la que se construirá toda la arquitectura. Por este motivo, es necesario comprender el funcionamiento de los sistemas de comunicación empleados en ROS 2.
  - **Tarea 1.2**: Estudio de MongoDB. Esta base de datos se utilizará para almacenar el conocimiento del robot utilizando la nueva herramienta KANT. De esta forma, es necesario saber modificar bases de datos de MongoDB desde Python3.
- **Tarea 2**: este grupo de tareas resumen el análisis y diseño de la solución. Está formado por las siguientes tareas:
  - **Tarea 2.1**: Estudio del estado de la cuestión. La primera tarea del diseño consiste en explorar las tecnologías y trabajos pasados y presentes.
  - **Tarea 2.2**: Definir los requisitos. Se realiza la especificación de requisitos para describir el sistema que se quiere desarrollar. Estos requisitos se centrarán en todos los componentes de MERLIN2, KANT y YASMIN.
  - **Tarea 2.3**: Definir la arquitectura. Se definirán los componentes de la arquitectura a alto nivel siguiendo los componentes que tenía MERLIN.
  - **Tarea 2.4**: Elegir el software. Para llevar a cabo ciertos componentes, es necesario buscar el software necesario para su creación.

- **Tarea 3**: este grupo de tareas se centra en el desarrollo software. Está formado por las siguientes tareas:
  - **Tarea 3.1**: KANT. Creación del sistema de gestión del conocimiento. Este sistema consistirá en varios paquetes ROS 2 que permitirán su uso desde otros paquetes.
  - **Tarea 3.1**: YASMIN. Creación de la librería para generar comportamientos mediante máquinas de estados. Este sistema consistirá en varios paquetes ROS 2 que permitirán su utilización desde otros paquetes.
  - **Tarea 3.3**: Arquitectura MERLIN2. El desarrollo de la arquitectura se ha dividido en las siguientes tareas:
    - **Tarea 3.3.1**: Generación de misiones. Al igual que MERLIN, MERLIN2 necesita un mecanismo para generar las misiones u objetivos que el robot tendrá que conseguir.
    - **Tarea 3.3.2**: Sistema de planificación. Es necesario crear un sistema que genera los planes, secuencia de acciones, para satisfacer los objetivos definidos. Para ello, será necesario desarrollar un sistema que acceda al conocimiento, utilice un planificador y gestione la ejecución de las acciones.
    - **Tarea 3.3.3**: Acciones de MERLIN2. Es necesario desarrollar una acción base a partir de la cual se puedan implementar nuevas acciones, como por ejemplo la navegación.
    - **Tarea 3.3.4** Sistemas reactivos. Es necesario migrar o implementar los sistemas necesarios. De esta forma, serán necesarios un sistema de navegación topológica, un sistema de reconocimiento del habla y un sistema de síntesis de voz.
  - **Tarea 3.4**: Desarrollar las pruebas. Se quieren desarrollar varias pruebas mediante MERLIN2. Para ello, se utilizarán entornos virtuales y robots simulados y reales.
- **Tarea 4**: Experimentación. Con el fin de evaluar el funcionamiento de la arquitectura, se quieren realizar varias ejecuciones de las pruebas implementadas.
- **Tarea 5**: Documentación. A lo largo del desarrollo del proyecto, se quieren desarrollar las diferentes partes de la memoria del Trabajo de Fin de Máster.

2.2.2. Estimación de tareas

El proyecto tiene una duración de 300 horas. Esto equivale a 100 días laborales, es decir, 5 meses de trabajo si se dedican 3 horas diarias. De esta forma, en la tabla 2.1 se presenta la estimación de la duración de las tareas descritas anteriormente. Además, para cada tarea se especifican sus fechas de inicio y fin, teniendo en cuenta los fines de semana y festivos, así como sus dependencias.

Tabla 2.1: Estimación tareas.

Tareas	Días	Fecha inicio	Fecha fin	Predecesoras
Tarea 1	15	06/10/2020	27/10/2020	-
Tarea 1.1	15	06/10/2020	27/10/2020	-
Tarea 1.2	5	06/10/2020	13/10/2020	-
Tarea 2	20	06/10/2020	04/11/2020	-
Tarea 2.1	15	06/10/2020	27/10/2020	-
Tarea 2.2	5	28/10/2020	04/11/2020	Tarea.2.1
Tarea 2.3	5	28/10/2020	04/11/2019	Tarea.2.1
Tarea 2.4	5	28/10/2020	04/11/2019	Tarea.2.1
Tarea 3	65	05/11/2020	02/03/2021	Tarea 1, Tarea 2
Tarea 3.1	25	05/11/2020	11/12/2020	-
Tarea 3.2	20	05/11/2020	02/12/2020	-
Tarea 3.3	50	05/11/2020	09/02/2021	-
Tarea 3.3.1	15	14/12/2020	19/01/2021	Tarea 3.1
Tarea 3.3.2	30	14/12/2020	09/02/2021	Tarea 3.1
Tarea 3.3.3	25	03/12/2020	26/01/2021	Tarea 3.2
Tarea 3.3.4	25	05/11/2020	11/12/2020	-
Tarea 3.4	15	10/02/2021	02/03/2021	Tarea 3.3
Tarea 4	15	03/03/2021	23/03/2021	Tarea 3.4
Tarea 5	100	06/10/2020	23/03/2021	-

2.2.3. Planificación de tareas

En la figura 2.1 se muestra el diagrama Gantt que se corresponde con las tareas anteriormente definidas. Como se puede ver, se especifican las dependencias y los solapamientos que hay. De esta forma, la planificación inicial tiene una duración de 5

meses con un trabajo diario de 3 horas, desde el 6 de octubre de 2020 hasta el 26 de febrero de 2021.

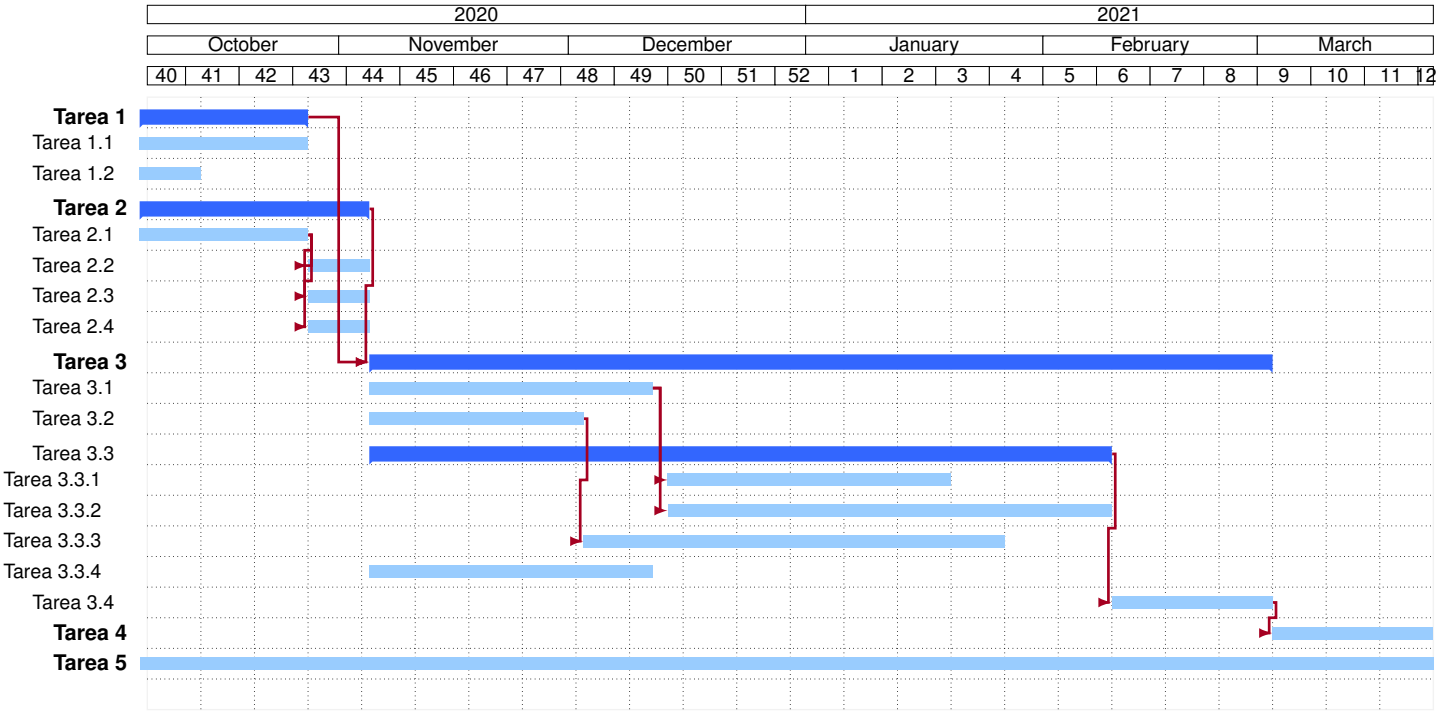


Figura 2.1: Diagrama Gantt de la planificación inicial de las tareas.

2.2.4. Iteraciones de Scrumban

En esta sección se presentan las iteraciones de Scrumban que se han llevado a cabo durante el desarrollo del proyecto. Cada iteración ha tenido una duración de cuatro semanas. En la tabla 2.2 se presentan las iteraciones llevadas a cabo.

Tabla 2.2: Iteraciones de Scrumban.

Iteración	Fecha inicio	Fecha fin	Tarea	Subtarea
1	06/10/2020	27/10/2020	Tarea 1.1	Comprender los nuevos mecanismos de comunicación básicos de ROS 2: Topics y Servicios.
			Tarea 1.1	Comprender el funcionamiento del spin, los ejecutores y los grupos de callback de ROS 2.
			Tarea 1.1	Comprender las Actions de ROS 2.

Continúa en la siguiente página

Tabla 2.2 – *Continuación de la página anterior*

Iteración	**Fecha inicio**	**Fecha fin**	**Tarea**	**Subtarea**
			Tarea 1.1	Crear un nodo ROS 2 basado en Multi-Thread.
			Tarea 1.2	Comprender MongoDB y visualizarlo con Compass.
			Tarea 1.2	Aprender a usar mongoengine para trabajar con MongoDB desde Python3.
			Tarea 2.1	Buscar arquitecturas software para robots.
			Tarea 2.2	Definir los requisitos de KANT.
			Tarea 2.2	Definir los requisitos de MERLIN2.
			Tarea 2.3	Diseñar KANT.
			Tarea 2.3	Elección del software para el reconocimiento del habla.
			Tarea 2.3	Elección del software para la síntesis de voz.
2	02/11/2020	27/11/2020	Tarea 2.2	Definir los requisitos de YASMIN.
			Tarea 2.3	Diseñar YASMIN.
			Tarea 2.3	Diseñar MERLIN2.
			Tarea 2.4	Elección del software para el reconocimiento del habla.
			Tarea 2.4	Elección del software para la síntesis de voz.
			Tarea 3.1	Crear los DTOs del PDDL (types, objects predicates, propositions, actions).
			Tarea 3.1	Crear los DAOs basados en MongoDB.
			Tarea 3.1	Crear una Knowledge Base gestionada por un nodo ROS 2.
			Tarea 3.1	Crear las interfaces de ROS 2 de la Knowledge Base.
			Tarea 3.1	Crear los DAOs basado en el nodo ROS 2.

Continúa en la siguiente página

Tabla 2.2 – *Continuación de la página anterior*

Iteración	**Fecha inicio**	**Fecha fin**	**Tarea**	**Subtarea**
			Tarea 3.3.2	Crear las interfaces de ROS 2 de la Planning Layer.
			Tarea 3.3.4	Migrar el sistema de reconocimiento del habla de MERLIN.
			Tarea 3.3.4	Desarrollar una nueva navegación topológica.
3	30/11/2020	25/12/2020	Tarea 3.1	Crear las factorías para los DAOs.
			Tarea 3.1	Integrar la configuración de KANT en ROS 2 mediante parámetros de nodos.
			Tarea 3.1	Crear una demo de ejemplo de KANT.
			Tarea 3.2	Crear la clase base State.
			Tarea 3.2	Crear la clase StateMachine.
			Tarea 3.3.2	Desarrollar un componente para generar el PDDL del robot usando KANT.
			Tarea 3.3.2	Desarrollar el componente encargado de crear planes.
			Tarea 3.3.2	Desarrollar el componente encargado de ejecutar las acciones.
			Tarea 3.3.3	Crear las interfaces de ROS 2 de las acciones de MERLIN2.
			Tarea 3.3.3	Desarrollar la clase base para producir acciones de MERLIN2.
			Tarea 3.3.4	Desarrollar un nuevo sistema de síntesis del habla.
4	04/01/2021	29/01/2021	Tarea 3.2	Crear un estado que integre los clientes de servicios de ROS 2.
			Tarea 3.2	Crear un estado que integre los clientes de acciones de ROS 2.
			Tarea 3.2	Crear un visualizador de máquinas de estados mediante Flask y React.js.

Continúa en la siguiente página

Tabla 2.2 – *Continuación de la página anterior*

Iteración	**Fecha inicio**	**Fecha fin**	**Tarea**	**Subtarea**
			Tarea 3.3.1	Migrar el componente encargado de añadir objetivos a la Knowledge Base.
			Tarea 3.3.1	Creación de un nodo base para crear nuevos nodos que gestionen las misiones del robot.
			Tarea 3.3.2	Desarrollar un componente que hace de fachada entre la Planning Layer y la Mission Layer.
			Tarea 3.3.3	Desarrollar la clase base para producir acciones de MERLIN2 basadas en máquinas de estados.
			Tarea 3.3.3	Crear estados básicos y una factoría para instanciarlos.
			Tarea 3.3.4	Desarrollar un nuevo sistema de síntesis del habla.
5	01/02/2021	26/02/2021	Tarea 3.1	Crear tests unitarios automatizados de KANT.
			Tarea 3.2	Crear tests unitarios automatizados de YASMIN.
			Tarea 3.3.2	Crear tests unitarios automatizados de MERLIN2.
			Tarea 3.4	Migrar el simulador del robot RB1 a ROS 2 Foxy.
			Tarea 3.4	Integrar el simulador en ROS 2 Foxy con el software desarrollado en ROS 2.
			Tarea 3.3.4	Introducir el simulador del robot RB1 en mundos de virtuales.
6	01/03/2021	25/03/2021	Tarea 3.4	Implementar nuevas pruebas de MERLIN2 utilizando el robot RB1 simulado y los robots reales RB1 y TIAGo.
			Tarea 3.4	Recrear el experimento descrito en [15].

Continúa en la siguiente página

Tabla 2.2 – *Continuación de la página anterior*

Iteración	Fecha inicio	Fecha fin	Tarea	Subtarea
			Tarea 4	Hacer varias ejecuciones de las pruebas desarrolladas.
			Tarea 4	Comparar los resultados obtenidos.
			Tarea 5	Completar la memoria del TFM.

2.3. Gestión de recursos

En esta sección se presentan los recursos que se han usado en el proyecto, exponiendo sus costes y características.

2.3.1. Especificación de recursos

Los recursos usados en el proyecto son los siguientes:

- **Ordenador de sobremesa**: uno de los computadores empleados en el desarrollo del proyecto es un ordenador de sobremesa personal. Sus componentes son:
  - Procesador i7 6700: frecuencia básica de 3,4 GHz, 4 núcleos, 8 hilos y 8 MB de memoria caché.
  - RAM: 16GB
  - Almacenamiento: 2T de disco duro + 512GB de SSD
  - Tarjeta Gráfica: Nvidia GeForce GTX 1060 con 6GB
- **Ordenador portátil**: también se ha utilizado un ordenador portátil. Está formado por los siguientes componentes:
  - Procesador i7 8750H: frecuencia básica de 2,2 GHz, 6 núcleos, 12 hilos y 9 MB de memoria caché.
  - RAM: 8GB
  - Almacenamiento: 1T de disco duro + 240 de SSD
  - Tarjeta Gráfica: Nvidia GeForce GTX 1050Ti con 4GB

- **Micrófono Rode VideoMic Rycote**: para llevar a cabo el desarrollo del sistema de reconocimiento del habla se usó un micrófono Rode VideoMic Rycote. Este micrófono se presenta en la figura 2.2. Sus características son las siguientes: enfoque direccional y opción de 80Hz de respuesta.

Figura 2.2: Micrófono Rode VideoMic Rycote.
Fuente: *https://images-na.ssl-images-amazon.com/images/I/818T4PT92YL._SX355_.jpg*

- **Robot RB1**: para realizar pruebas se ha utilizado el robot RB1, que se presenta en la figura 2.3. Este robot ha sido creado por Robotnik [31]. Es un robot móvil que usa ROS 1 Kinetic. Además, está compuesto por los siguientes elementos:
  - Base: tiene varias ruedas para que el robot se mueva en diferentes direcciones. Además, tiene un sensor láser Hokuyo [32] con un rango de 240 grados y una distancia de 5,6 metros.
  - Torso: este elemento se monta sobre la base. En su interior hay un router, para facilitar el trabajo con el robot; y un ordenador, en el que se ejecutan los paquetes de ROS. Además, hay un brazo Kinova Jaco [33] con seis grados de libertad.
  - Cabeza: contiene una cámara ASUS Xtion. Es de tipo RBG-D (Red Green Blue and Deep) y se puede utilizar para obtener colores y profundidades.

Figura 2.3: Robot RB1.
Fuente:
https://www.robotnik.es/web/wp-content/uploads/2014/06/RB-1_Galeria_001.png

- **Robot TIAGo**: este robot se presenta en la figura 2.4. Ha sido creado por PAL Robotics [34]. TIAGo es un robot móvil que usa ROS. Se puede dividir en las siguientes partes:
  - Base: tiene varias ruedas para mover el robot en diferentes direcciones. Además, la base tiene un sensor láser Hokuyo con una distancia de 10 metros.
  - Torso: se monta sobre la base. En su interior hay un ordenador en el que se ejecutan los paquetes ROS diseñados para este robot. También tiene un micrófono, un altavoz, una Nvidia Jetson TX2 [35] en su parte trasera y una tablet en su parte delantera.
  - Cabeza: tiene una cámara similar a la ASUS Xtion.

Figura 2.4: Robot TIAGo.
Fuente: *http://erl.pal-robotics.com/wp-content/uploads/2018/11/TIAGo_Iron_2017_with_screen_04-600x1359-1.png*

- **Sistema Operativo**: el sistema operativo usado en los computadores descritos anteriormente es Ubuntu 20.04 LTS [36].
- **ROS** (Robot Operating System): ROS es el framework más extendido en el desarrollo de aplicaciones para robots. Está formado por librerías que permiten abstraerse del hardware y visualizar datos. ROS se basa en un sistema distribuido de nodos que se comunican de diferentes formas. Se ha utilizado ROS 2 Foxy Fitzroy para desarrollar la arquitectura MERLIN2. Además, se ha utilizado ROS 1 Noetic para ejecutar los diferentes simuladores y poder trabajar con los robots reales durante la evaluación.
- **Entorno real de pruebas**: el Laboratorio de Investigación del Grupo de Robótica de la Universidad de León se ha utilizado como entorno real para realizar pruebas. Está situado en el edificio MIC (Módulo de Investigación en Cibernética). Su composición se presenta en la figura 2.5. Está formado por las siguientes secciones:

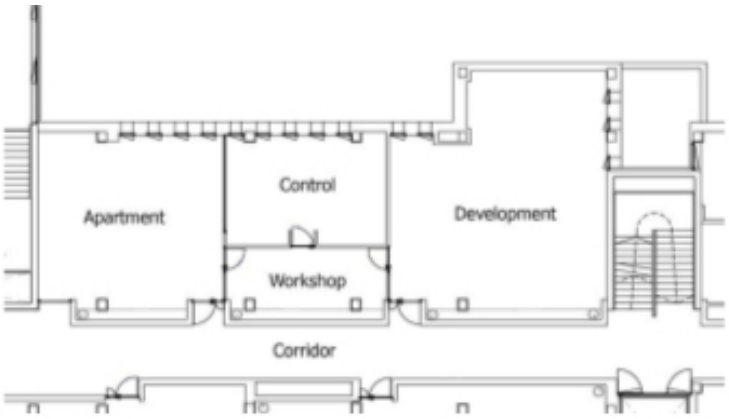


Figura 2.5: Mapa del Laboratorio del Grupo de Robótica de la Universidad de León.
Fuente: *http://robotica.unileon.es/index.php/Benchmark_dataset_for_evaluation_of_range-based_people_tracker_classifiers_in_mobile_robots*

- Zona de desarrollo: esta zona está destinada al desarrollo de los proyectos de los investigadores del grupo.
- Zona de mantenimiento: el mantenimiento de los robots se realiza en esta zona.
- Sala de control: esta zona sirve para monitorizar el apartamento.
- Apartamento simulado: esta zona se basa en un apartamento simulado para efectuar pruebas. Se puede ver en la figura 2.6. Consta de varias partes que tendría un apartamento real.

Figura 2.6: Apartamento simulado del Grupo de Robótica de la Universidad de León.
Fuente: *http://robotica.unileon.es/index.php/Benchmark_dataset_for_evaluation_of_range-based_people_tracker_classifiers_in_mobile_robots*

2.3.2. Presupuesto

En esta sección se muestra el presupuesto estimado del proyecto.

Coste de personal

Para realizar este proyecto se necesita un Ingeniero Informático. Además, se ha necesitado un Scrum Master. Para obtener los salarios del personal se ha usado la página de la Seguridad Social de Bases y tipos de cotizaciones [37]. De esta información se pueden aproximar los salarios medios. Además, para calcular el coste del personal es necesario aplicar las tasas de la Seguridad Social al salario base. Estas tasas se presentan en la tabla 2.3.

Tabla 2.3: Tasas de la Seguridad Social.

Tasa	Valor
Contingencias comunes de la empresa	23,6 %
Cuota de desempleo de la empresa	0,2 %
Cuota de formación de la empresa	6,7 %
Fogasa	0,6 %
Accidentes de trabajo	1 %
Total	32,1 % €

En la tabla 2.4 se presenta el salario del personal del proyecto. El coste por hora se ha calculado multiplicando 1,321 por su salario. Además, hay que tener en cuenta la duración, 5 meses, y el trabajo diario de 3 horas.

Tabla 2.4: Coste del personal

Cargo	Salario/Hora	Coste/Hora (Salario * 1,385)	Horas trabajadas	Total
Scrum Master	16 €	19,39	50	969,50 €
Ingeniero Informático	14 €	19,39	300	5.817,00 €
Total				6.786,50 €

Coste de hardware

En este apartado se presentan los costes del hardware definido en 2.3.1. De esta manera, se tienen los siguientes costes:

- **Ordenador de sobremesa**: el coste sin IVA de este computador es 1247,93 €. El tiempo para amortizarlo es de 3 años. Se ha utilizado durante 5 meses por lo que su coste estimado sin IVA es **173,32 €**.
- **Ordenador portátil**: el coste sin IVA de este computador es 743,8 €. El tiempo para amortizarlo es de 3 años. Se ha utilizado durante 5 meses por lo que su coste estimado sin IVA es **103,31 €**.
- **Micrófono Rode VideoMic Rycote**: el coste sin IVA de este micrófono es de 74,38 €. Su tiempo de amortización es de 2 años. Como se ha usado durante 1 mes, su coste estimado sin IVA es **3,10 €**.
- **Robot RB1**: el coste sin IVA del RB1 es 69.050,00 € . Su tiempo de amortización es de 4 años. Como se ha usado durante 1 mes, su coste estimado sin IVA es **1.438,54 €**.
- **Robot TIAGo**: el coste sin IVA del TIAGo es 34.000,00 €. Su tiempo de amortización es de 4 años. Como se ha usado durante 1 mes, su coste estimado sin IVA es **708,33 €**.

El coste total del hardware que se ha presentado anteriormente se resume en la tabla 2.5.

Tabla 2.5: Coste del hardware.

Hardware	Coste
Ordenador de sobre mesa personal	173,32 €
Ordenador portátil personal	103,31 €
Micrófono Rode VideoMic Rycote	3,10 €
Robot OrbiOne	1.438,54 €
Robot Tiago	708,33 €
Total	2.426,60 €

Costes indirectos industrial

En este apartado se presenta el cálculo de los costes indirectos. Estos costes incluyen la luz, el agua, la red, el alquiler y otros conceptos que no se tienen en cuenta de forma directa en el proyecto. Según las Presupuestos de la Universidad de León [38], es el 15 % y se calcula sobre el coste total del proyecto sin IVA. De esta forma, el beneficio industrial que se obtiene en este proyecto se presenta en la tabla 2.6.

Tabla 2.6: Costes indirectos.

Concepto	Valor
Coste del personal	6.786,50 €
Coste del hardware	2.426,60 €
Subtotal	9.213,10 €
Costes indirectos (15 %)	1.381,96 €

Beneficio industrial

En este apartado se presenta el beneficio industrial. Es el porcentaje que se obtiene de beneficio con los resultados. Según el BOE [39], es el 6 % y se calcula sobre el coste total del proyecto sin IVA. De esta forma, el beneficio industrial que se obtiene en este proyecto se presenta en la tabla 2.7.

Tabla 2.7: Beneficio industrial.

Concepto	Valor
Coste del personal	6.786,50 €
Coste del hardware	2.426,60 €
Subtotal	9.213,10 €
Beneficio industrial (6 %)	552,79 €

Coste total

Después de calcular los costes del personal, los costes del hardware y el beneficio industrial se calcula el coste total. En este cálculo se aplica el IVA. Estos cálculos se presentan en la tabla 2.8.

Tabla 2.8: Coste total

Concepto	Valor
Coste del personal	6.786,50 €
Coste del hardware	2.426,60 €
Costes indirectos (15 %)	1.381,96 €
Beneficio industrial (6 %)	552,79 €
Subtotal	11.147,86 €
IVA (21 %)	2.341,05 €
Total	13.488,91 €

Capítulo 3

Solución

3.1. Descripción de la solución

En este proyecto se presenta MERLIN2, una arquitectura cognitiva para robots. Se ha diseñado para ser usada en la creación y desarrollo de nuevos paquetes de ROS 2 que quieran gestionar las tareas de un robot. Estas tareas se basarían en problemas específicos que se pueden dar en entornos reales.

MERLIN2 está formada por varios paquetes de ROS 2. De esta forma, se instala igual que cualquier paquete normal. Esto consiste en mover el código de MERLIN2 al directorio de código del workspace de ROS 2 y emplear colcon [40], mediante colcon build. Al hacer esto, el código Python3 será agregado a la variable de entorno PYTHONPATH haciendo que este código se pueda utilizar en cualquier archivo Python3.

Para crear esta arquitectura se ha diseñado y desarrollado un nuevo sistema de gestión del conocimiento basado en PDDL llamado KANT. Para ello, se han empleado los patrones de diseño software DTO, DAO, Abstract Factory y Factory Method. Gracias a esto, ya no se generan los archivos PDDL del dominio, en el que se definen los tipos de objetos que puede haber, los predicados y las acciones que puede realizar el robot; y el problema, en el que se definen los objetos del entorno, sus proposiciones y los objetivos que quiere cumplir el robot. Además, este sistema se ha usado para desarrollar un nuevo sistema de planificación que reemplazará a ROSPlan [16].

Por otro lado, se ha llevado a cabo el diseño y desarrollo de una librería de Python3 para reemplazar a SMACH [17] llamada YASMIN. De esta manera, YASMIN se utilizará para desarrollar nuevos comportamientos mediante la creación de máquinas de estados. Estas máquinas se emplearán especialmente en la creación de las acciones de

MERLIN2. Además, se ha desarrollado un visualizador para presentar la ejecución de las máquinas de estados.

Por último, para utilizar la arquitectura MERLIN2 es necesario implementar dos componentes. El primero consiste en las acciones que el robot podrá realizar. Estas acciones serían máquinas de estados que emplean otros sistemas, como la navegación, la síntesis de voz y el reconocimiento del habla. El segundo consiste en la generación de los objetivos y las misiones que el robot tendrá que cumplir.

3.2. Diseño de la solución

En esta sección se describen la especificación de requisitos, el software elegido, el diseño de la base de datos, el diseño de KANT, el diseño de YASMIN y el diseño de la arquitectura MERLIN2.

3.2.1. Requisitos

Antes de comenzar el desarrollo del proyecto se definen los requisitos que el sistema tendrá que cumplir. De esta forma, se distinguen requisitos funcionales y no funcionales.

Los requisitos funcionales son los siguientes:

1. El robot tiene que mantener el conocimiento en una base de conocimientos.
2. La base de conocimiento puede ser un nodo ROS 2 o una base de datos.
3. El robot tiene que poder acceder al conocimiento de la base de conocimientos.
4. El robot tiene que poder crear nuevo conocimiento en la base de conocimientos.
5. El robot tiene que poder editar el conocimiento de la base de conocimientos.
6. El robot tiene que poder eliminar el conocimiento de la base de conocimientos.
7. El robot ejecutará sus objetivos. Los objetivos son las proposiciones que el robot quiere cumplir que se almacenarán en la base de conocimientos.
8. El robot puede cancelar la ejecución de los objetivos.
9. El robot tiene que generar los archivos PDDL a partir del conocimiento de la base de conocimientos.

10. El robot tiene que crear planes para resolver los objetivos definidos en la base de conocimientos.
11. La generación de planes se basa en el uso de planificadores de PDDL.
12. El robot tiene que ejecutar sus acciones en la secuencia especificada en los planes generados.
13. El robot tendrá un conjunto de acciones. La gestión de la ejecución de las acciones es una de las partes importantes en la arquitectura.
14. Las acciones usarán los sistemas reactivos, como la navegación, el reconocimiento del habla o la síntesis de voz.
15. El robot puede comprobar el cumplimiento de estas acciones.
16. El robot puede cancelar el plan que se está ejecutando, cancelando la acción que se está ejecutando.
17. Las acciones se autorregistrarán en la base de conocimientos cuando se instancien en tiempo de ejecución.
18. El robot navegará por las zonas definidas con anterioridad.
19. La navegación se configura mediante el uso de mapas pregenerados.
20. El robot tendrá una lista de puntos a los que podrá navegar.
21. Se podrán añadir nuevos puntos de navegación en tiempo de ejecución.
22. El robot reconocerá el habla de las personas con las que interactúe.
23. El reconocimiento del habla puede ser offline u online.
24. El robot sintetizará voz para interactuar con las personas del entorno.
25. La síntesis de voz puede ser offline u online.
26. El habla reconocida se traducirá para facilitar su utilización en otros componentes.
27. Los desarrolladores podrán usar máquinas de estados para desarrollar comportamientos integrados en ROS 2.
28. Los desarrolladores podrán crear objetivos para el robot.
29. Los desarrolladores tienen una clase base para crear nuevas acciones integradas en la ejecución MERLIN2.

30. Los desarrolladores podrán visualizar las máquinas de estados que se están ejecutando.
31. Se puede seleccionar visualizar una sola máquina de estados o todas ellas a la vez.
32. Los desarrolladores podrán visualizar el conocimiento del robot. Para ello, se mostrará el conocimiento almacenado en la base de conocimientos.

Por último, también se han definido una serie de requisitos no funcionales que se quieren cumplir:

1. Se quiere que el uso y mantenimiento del conocimiento del robot sea sencillo.
2. Se quiere que la generación de nuevos objetivos sea sencilla.
3. Se quiere que la creación de nuevas acciones para desarrollar nuevas aplicaciones sea sencilla.
4. Se quiere que la utilización de máquinas de estados permita crear acciones flexibles y de forma sencilla.
5. Se quiere que MERLIN2 sea fácil de comprender.
6. Se quiere que el flujo de ejecución de MERLIN2 sea consistente, propagando tanto la ejecución como la cancelación de sus componentes de manera correcta.

3.2.2. Software elegido

En esta sección se presentan las tecnologías empleadas para llevar a cabo el proyecto. Durante el desarrollo del proyecto se ha realizado un estudio de las tecnologías, comparando en algunos casos varios alternativas. De esta forma, las tecnologías usadas son las siguientes:

- ROS 2: es la nueva versión de ROS [12], el framework más extendido para desarrollar aplicaciones para robots. Su uso se centra en la creación de nodos que se comunican mediante diferentes tipos de comunicación síncronos y asíncronos; topics, servicios y acciones. Este framework se ha descrito en mayor profundidad en el Estudio del problema 1.1.1. Además, como Data Distributed Services (DDS) se usa eProsima Fast RTPS, que es el DDS por defecto en ROS 2.
- Python3: el lenguaje Python [11] es un lenguaje de programación de propósito general. Es multiplataforma y es interpretado.

- pylint: la librería pylint [10] consiste en una herramienta software para buscar errores de código y de estilo para Python.
- pytest: la librería pytest [8] consiste en un framework para crear tests de Python.
- unittest: la librería unittest [13] consiste en un framework para producir tests de Python.
- PDDL: el lenguaje PDDL [9] es el lenguaje para planificación basada en inteligencia artificial más usado en la robótica. Se basa en el uso de un archivo dominio, que contiene los tipos, predicados y acciones; y un archivo problema, que contiene los objetos, proposiciones y objetivos.
- POPF 2: el planificador POPF [41] es un planificador para PDDL que incluye conceptos de la planificación de orden parcial. Esto significa que, durante la búsqueda, al aplicar una acción de PDDL se intenta introducir únicamente las restricciones de orden necesarias para resolver los problemas, en vez de insistir en realizar la nueva acción después del resto de acciones que ya tiene el plan.
- MongoDB: es una base de datos [6] no relacional, distribuida, basada en documentos y de uso general. Las alternativas que se estudiaron y que se rechazaron son ArangoDB [42], por tener peor rendimiento que MongoDb; y Neo4J [43], por no tener la misma flexibilidad para almacenar datos que MongodDB. También se valoró la posibilidad de usar Mongo Atlas [44], la versión en la nube de MongoDB, sin embargo, se rechazó porque su utilización añade demasiado retardo al funcionamiento de la arquitectura.
- mongoengine: [7] es un DOM (Document-Object Mapper) para trabar con MongoDB desde Python.
- JavaScript: el lenguaje JavaScript [45] es un lenguaje de programación multiparadigma. Es ligero, interpretado y puede ser compilado. Se usa especialmente en el desarrollo de aplicaciones web.
- React.js: la librería React [46] es una librería de JavaScript para crear aplicaciones web. Se ha empleado para producir el visualizador de YASMIN.
- Cytoscape.js: la librería Cytoscape[47] es una librería de JavaScript para visualizar grafos. Se ha usado para dibujar las máquinas de estados de YASMIN.
- Flask: la librería Flask [48] es una librería de Python3 para desarrollar aplicaciones web.

- eSpeak: eSpeak [49] es un software de código abierto para realizar síntesis de voz en inglés. Es offline.
- Speech Dispatcher: este proyecto [50] proporciona una interfaz para efectuar síntesis de voz. Es offline.
- Festival: el framework Festival [51] es un framework que permite hacer síntesis de voz en diferentes idiomas. Es offline.
- gTTS: la librería de gTTS[52] es una librería de Python que proporciona una interfaz para usar la API de síntesis de voz de Google Translate. Es online.

3.2.3. Diseño de KANT

En esta sección se presenta el diseño de KANT (Knowled mAnagemeNT). Para ello, se ha desarrollado en diseño presentado en la figura 3.1.

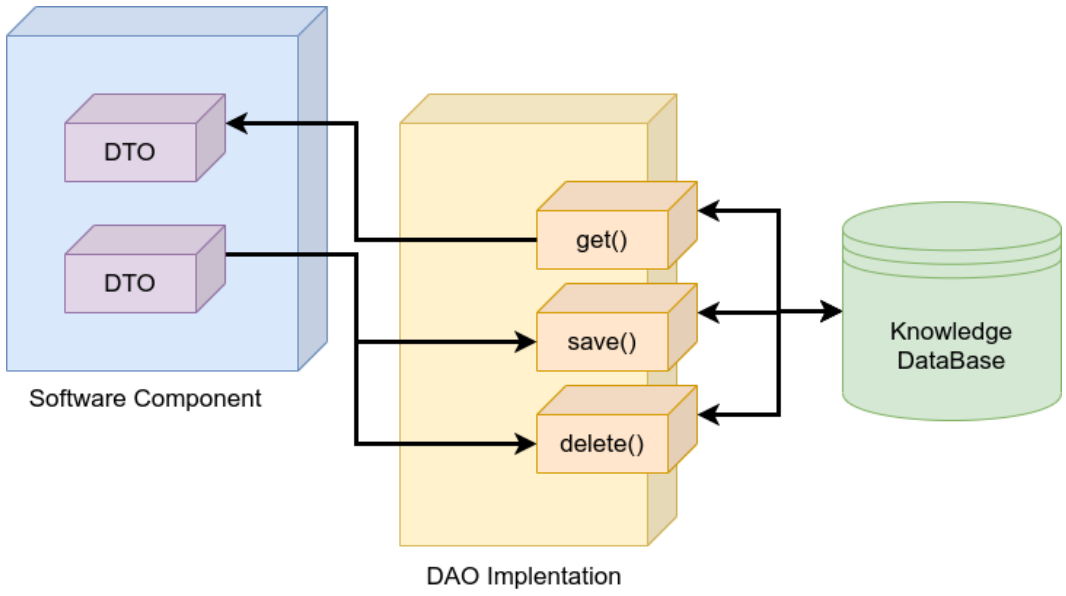


Figura 3.1: Arquitectura DTO y DAO de KANT.

Como se puede ver, el patrón DTO se emplea para encapsular el PDDL. Por otro lado, el DAO se usa para acceder a la base del conocimiento del robot. Esta base de conocimiento puede ser un nodo de ROS 2 o una base de datos de MongoDB. Para crear las instancias de los diferentes DAO se han diseño varias factorías siguiendo el Abstract Factory Pattern. De esta forma, hay una factoría por cada tipo de almacenamiento. Por último, para facilitar el cambio del tipo de almacenamiento al iniciar una ejecución se ha utilizado el Factory Method Pattern. Este patrón creará instancias de factoría

que tendrán diferente lógica, dependiendo del tipo de almacenamiento. Los diagramas UML de clases de KANT se presenta en las figuras C.2, C.3, C.5, C.6 y C.4 del anexo C.

Diseño de la base de datos

En este apartado se presenta el diseño de la base de datos de MongoDB. Esta base de datos se ha basado en ciertos aspectos del PDDL que se quieren modelar. En la figura 3.2 se muestra el diagrama de la base de datos desarrollada, creado con la herramienta MoonModeler [53]. Los documentos azules son documentos normales y los documentos verdes son documentos embebidos. A continuación se describen las colecciones de documentos creadas:

- **pddl_type**: este documento está formado por un ID y un string, que representa el nombre del tipo en PDDL.
- **pddl_object**: este documento está formado por un ID y un string, que representa el nombre del objeto en PDDL. Además, tiene un documento pddl_type, que representa el tipo del objeto.
- **pddl_predicate**: este documento está formado por un ID y un string, que representa el nombre del predicado en PDDL. Además, tiene una lista de documentos pddl_type, que representan los argumentos del predicado PDDL.

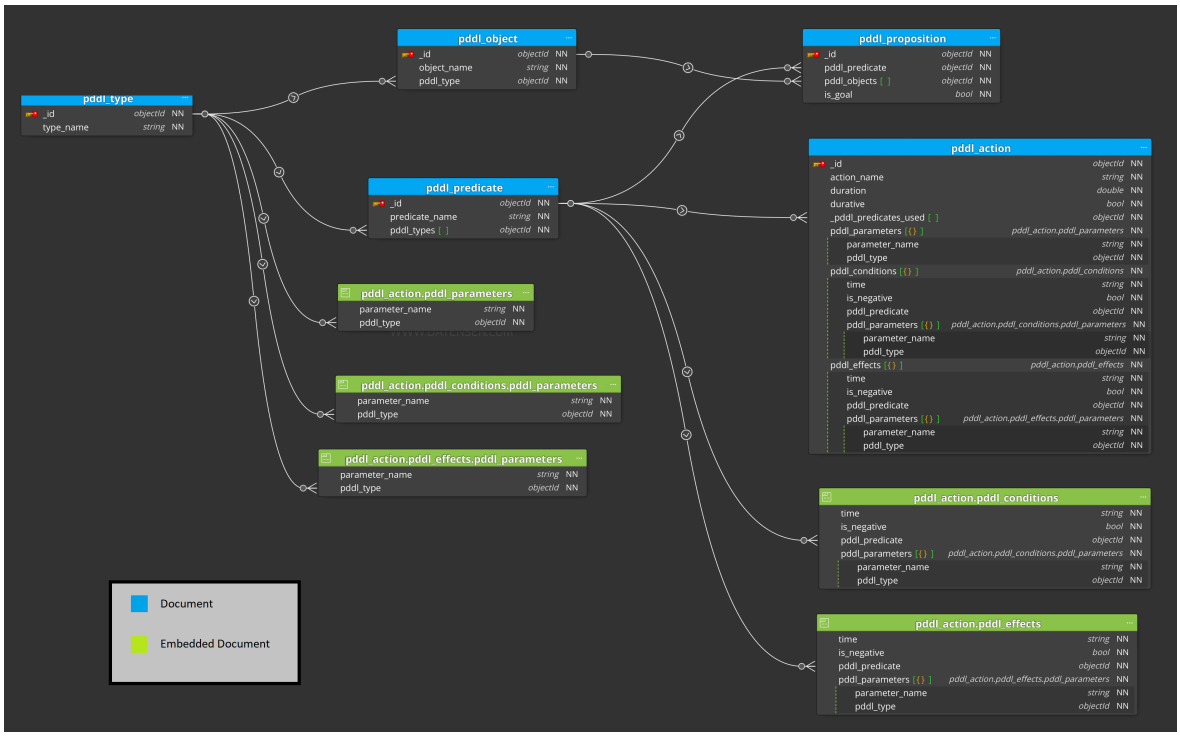


Figura 3.2: Diagrama de la base de datos de MongoDB.

- **pddl_proposition**: este documento está formado por un ID y una variable booleana para determinar si la proposición es un objetivo o no. Además, tiene un documento pddl_predicate, que representa el predicado de la proposición PDDL. Por último, tiene una lista de documentos pddl_object, que representan los objetos que forman la proposición PDDL.
- **pddl_action**: este documento está formado por un ID y un string, que representa el nombre de la acción en PDDL; un número decimal, para representar su duración; y una variable booleana, parar representar si la acción es durativa o no. Además, se tiene una lista de documentos pddl_predicate para almacenar los predicados que esta acción necesita. Por último, se tienen las siguientes listas de documentos embebidos:
  - pddl_parameter: esta lista de documentos embebidos pddl_parameter representa los parámetros de la acción PDDL. Este documento está formado por un string, que es el nombre del parámetro; y un documento pddl_type, que es el tipo PDDL del parámetro.
  - pddl_conditions: esta lista de documentos embebidos pddl_condition representa las condiciones que la acción tiene que cumplir para poder ejecutarse. Este documento está formado por un string, que representa el momento en el que se tiene que cumplir la condición y puede ser al inicio, al final o durante la ejecución; una variable booleana para determinar si la condición es negativa o no; un documento pddl_predicate; y una lista de documentos embebidos pddl_parameter, que son los parámetros de la acción que se utilizan para construir la condición.
  - pddl_effects: esta lista de documentos embebidos pddl_effect representa los efectos que la acción produce al ejecutarse. La estructura de este documento es similar a la del documento pddl_condition.

3.2.4. Diseño de YASMIN

En esta sección se presenta el diseño de YASMIN. Como se ha mencionado anteriormente, YASMIN se basa en SMACH [17]. De esta forma, YASMIN tiene una clase State base que se usará para crear sus otras clases, como la StateMachine. Además, se tiene la clase YasminViewerPub, encargada de generar el estado de las máquinas de estados para visualizarlas posteriormente. El diagrama UML de clases de YASMIN se presenta en la figura C.7 del anexo C.

3.2.5. Diseño de MERLIN2

En esta sección se presenta el diseño la arquitectura MERLIN2. Es una arquitectura híbrida clásica formada por dos componentes principales: Deliberative y Behavioral. En la figura 3.3 se muestra el diagrama de la arquitectura MELRIN2.

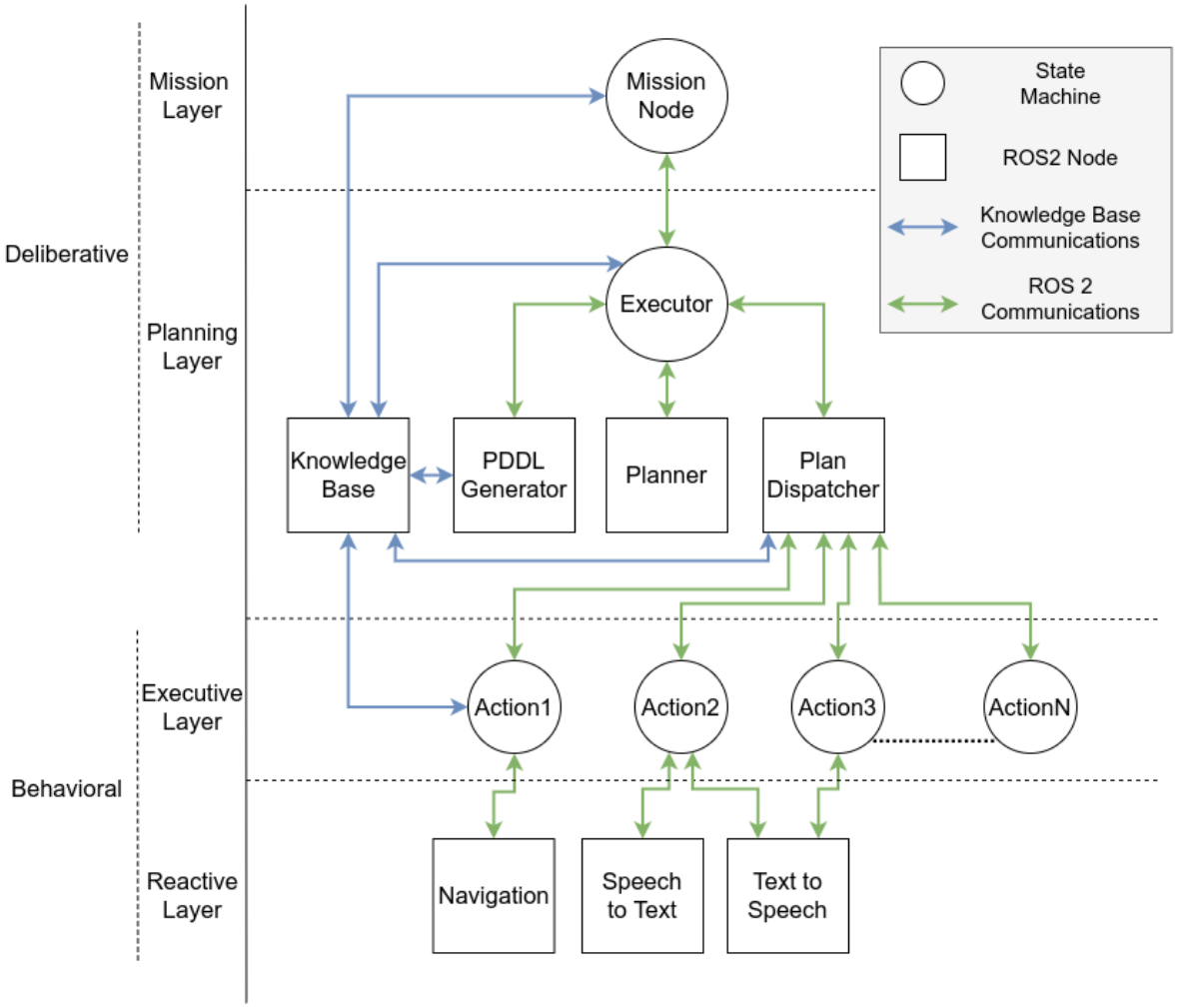


Figura 3.3: Diagrama de la arquitectura MERLIN2.

Como se puede ver, estos componentes están divididos en capas que, al igual que en MERLIN [15], son las siguientes:

- **Mission Layer**: es la capa encargada de gestionar los objetivos del robot. Su función es crear objetivos que serán usados por la Planning Layer para generar planes.
- **Planing Layer**: es la capa encargada de generar y ejecutar planes. Para ello, se genera el PDDL del estado actual y el dominio utilizando KANT. Después, ese PDDL se usa en el planificador para generar el plan, la secuencia de acciones.

Por último, la secuencia de acciones se ejecuta llamando a cada acción, que son los componentes de la Executive Layer.

- **Executive Layer**: es la capa formada por las acciones empleadas para producir planes. Para mejorar la gestión de las acciones, se han diseñado como máquinas de estados de YASMIN, cuyos estados pueden procesar datos o llamar a los sistemas de la Reactive Layer.
- **Reactive Layer**: es la capa formada por los sistemas reactivos. Algunos sistemas reactivos son la navegación, el reconocimiento de objetos, la síntesis de voz y el reconocimiento de objetos. Todos estos sistemas tienen que tener una interfaz de comunicación de ROS 2 para poder usarse.

3.3. Implementación

En esta sección se presenta la implementación realizada. El proyecto está formado por 460 archivos y 41.845 líneas. Por otro lado, se han creado 38 paquetes de ROS 2, 23 nodos de ROS 2 y 92 clases. En la tabla 3.1 se resumen estas líneas.

Tabla 3.1: Tabla resumen de las líneas de código escritas generado con VS Code Counter [2]

language	files	code	comment	blank	total
Python	255	9.517	3.567	4.230	17.314
Shell Script	14	81	12	36	129
JavaScript	11	382	28	82	492
HTML	2	18	23	3	44
CSS	3	46	1	8	55
ROS Message	37	135	17	47	199
JSON	5	16.900	0	2	16.902
XML	83	3.899	97	649	4.645
Properties	12	48	0	12	60
YAML	19	1.299	74	139	1.512
Markdown	19	308	20	165	493

3.3.1. Paquete simple_node

Antes de comenzar la implementación de los componentes descritos anteriormente, se decidió crear una clase base Node que encapsulara cierta funcionalidad para facilitar el uso de los nodos de ROS 2. El diagrama de clases se presenta en la figura C.1 del anexo C.

En ROS 2, los ejecutores son los encargados de gestionar el sistema de threading que ejecuta los callbacks. Por defecto se usa el SingleThreadExecutor, que utiliza un único hilo. El principal problema es que si solo se tiene un hilo de ejecución para cada nodo, no se podrá tratar más de un callback a la vez. Además, los clientes de servicios y acciones también usan callbacks para recibir información de sus servidores. Por este motivo no se podrían utilizar clientes dentro de callbacks.

Para solucionar esto, se creó el paquete de ROS 2 simple_node en el que se desarrolló la clase Node. La principal aportación de este nodo es que usa un hilo de Python3 para ejecutar un MultiThreadedExecutor. Este tipo de ejecutor reparte el trabajo a nuevos hilos, es decir, los callbacks son tratados por hilos nuevos. Con esto se consigue que el hilo principal y el hilo del ejecutor estén libres.

Por último, se desarrolló un nuevo cliente para las acciones, ActionClient. Con este nuevo cliente se pueden ejecutar objetivos (goals) de forma más sencilla ya que se encarga de esperar por el servidor de forma transparente para el desarrollador. Además, se puede comprobar su estatus utilizando sus métodos, como por ejemplo is_succeed, is_aborted y is_canceled. También se desarrollaron dos tipos de servidores de acciones. El primero de ellos es el ActionSingleServer, que solo admite un goal ejecutándose a la vez y aborta el goal actual si se recibe uno nuevo. El segundo es el ActionQueueServer, que encola los goals y los va tratando en orden de llegada (FIFO).

3.3.2. Implementación de KANT

En este apartado se presenta la implementación de KANT (Knowledge mAnageMeNT). Está dividido en cuatro paquetes de ROS 2, siendo uno de ellos el paquete de las interfaces de ROS 2 empleadas en KANT. A continuación se presenta de forma más detallada la implementación de estos paquetes.

Paquete kant_dto

Para poder utilizar el conocimiento basado en PDDL desde código Python3, se ha utilizado el patrón de diseño software DTO (Data Transfer Object). Este patrón tiene el objetivo de encapsular la transferencia de los datos entre las clases y los módulos de una aplicación, que en este caso es el PDDL. DTO se basa en la creación de atributos privados que representan los datos. Se debe acceder a estos atributos y editarlos con las funciones getter, setter y el constructor.

Antes de usar el patrón DTO, se han definido los elementos PDDL que se quieren utilizar. En PDDL existen muchos tipos de elementos, pero no todos ellos están soportados por los planificadores disponibles. Además, hay elementos PDDL que no tienen tanta utilidad dentro de la robótica. De esta forma, los elementos PDDL elegidos son los siguientes:

- **Types**: los tipos representan las clases de objetos que se pueden dar en el problema.
- **Objects**: los objetos son las instancias que se pueden dar en el problema.
- **Predicates**: los predicados son los tipos de características que puede haber. Son multi-argumento, que significa que una misma característica puede afectar a varios objetos.
- **Propositions**: las proposiciones son las instancias de los predicados. Si una proposición existe en el problema, entonces se toma como que es verdadera. Todo lo que no esté en el problema será falso. Además, hay proposiciones que representan objetivos (goals), que son proposiciones que se quiere que sean verdades, es decir, que aparezcan en el problema.
- **Actions**: las acciones son el mecanismo para alterar el problema. Las acciones tienen condiciones, que son proposiciones que tienen que ser verdad para poder usar la acción; y efectos, que son proposiciones que se harán verdad al completar la acción. Se tienen dos tipos de acciones: las acciones normales y las acciones durativas, que marcan el momento en el que se cumplen las condiciones y efectos. Estas últimas son las más usadas.

Los elementos PDDL se han encapsulado en clases DTO. De esta forma, se ha desarrollado una clase DTO para cada elemento PDDL. Estas clases se presentan en el diagrama de clases presentado en la figura C.2 del anexo C. La estructura de cada clase DTO es la siguiente:

- **Types (TypeDTO)**: un tipo se compone de un solo atributo string, que representa su nombre.
- **Objects (ObjectDTO)**: un objeto se compone de un atributo TypeDTO, que representa su tipo PDDL; y un atributo string, que representa su nombre.
- **Predicate (PredicateDTO)**: un predicado se compone de un atributo string, que representa su nombre; y un atributo lista de TypeDTO, que representa sus argumentos.
- **Propositions (PropositionDTO)**: una proposición se compone de un atributo string, que representa su nombre; y un atributo lista ObjectDTO, que representa sus objetos PDDL. También tiene un atributo booleano, que representa si es un objetivo.
- **Actions (ActionDTO)**: una acción se compone de los siguientes atributos:
  - Un atributo string, que representa su nombre.
  - Un atributo booleano, que representa si es una acción durativa.
  - Un atributo entero, que representa su duración.
  - Un atributo lista de ObjectDTO, que representa sus parámetros.
  - Dos atributos listas de DTO de Condition/Effect, que representan sus condiciones y efectos. El DTO de Condition/Effect es similar al PropositionDTO, pero también tiene un atributo string, que representa el momento en el que debe ocurrir la condición o efecto (at start, at end, over all); y un atributo booleano, que representa si es una condición o efecto negativo.

En las clases se han implementado los métodos getter y setter, para trabajar con sus atributos, y el método `__str__`, que traduce el contenido de la clase a texto PDDL. Por último, para gestionar estas clases se ha utilizado un paquete ROS 2. Gracias a los paquetes ROS 2 de Python, el código de las clases DTO se añadirá a la variable de entorno de Python, lo que hace posible acceder a ellas desde cualquier código Python, no solo desde código ROS 2.

Paquete kant_dao

Para acceder y modificar el conocimiento PDDL se ha utilizado el patrón de diseño software DAO (Data Access Object). El objetivo de este patrón es usar un objeto de acceso a datos para abstraer y encapsular todo acceso a la fuente de datos. De esta

forma, el DAO maneja la conexión a la fuente de datos para obtener y modificar datos, por lo que debería encapsular la lógica para recuperar, guardar, actualizar y eliminar datos. La fuente de datos puede ser una base de datos, un sistema de archivos, un proceso que lo almacene en memoria; entre otros.

Al combinar los patrones DTO y DAO, los diferentes componentes software de un sistema pueden consultar, guardar, editar y eliminar los datos de la base de conocimiento sin tener que preocuparse por el tipo e implementación de la fuente de datos. La arquitectura que se da al usar estos patrones se presenta en la figura 3.1. Por un lado, la función get del DAO es la función para consultar datos. En el caso de tipos, objetos, predicados y acciones PDDL, esta función utiliza sus nombres para identificarlos. Sin embargo, en el caso de las proposiciones PDDL, hay tres funciones get: get_by_predicate, para buscar una lista de proposiciones con un nombre de predicado dado; get_goals, para buscar las proposiciones que son objetivos; y get_no_goals, para buscar las proposiciones que no son objetivos. Por otro lado, la función de guardar se implementa en todas las clases de DAO y se utiliza para crear nuevo conocimiento y editar el conocimiento existente. Finalmente, la función de eliminación también se implementa en todas las clases de DAO y se utiliza para eliminar el conocimiento.

La implementación del DAO se basa en la creación de clases interface que son la base para desarrollar las clases de DAO concretas. Estas clases concretas se clasifican en familias DAO y gracias a las interfaces, todas las familias tendrán las mismas funciones para acceder al conocimiento. Hay dos familias de DAO:

- **MONGODB**: esta familia DAO usa MongoDB [6] para almacenar el conocimiento, lo que significa que MongoDB es la base de conocimiento. Gracias a esto se puede utilizar MongoDB Compass [54], un visualizador de MongoDB, para visualizar el conocimiento actual. El diagrama de clases de esta familia se presenta en la figura C.5 del anexo C. Para implementar las clases concretas para esta familia se ha utilizado MongoEngine [7]. MongoEngine es un DOM (Document-Object Mapper) escrito en Python para trabajar con MongoDB. Para usarlo, los modelos de Documentos y Documentos Embebidos deben estar definidos para cada elemento PDDL. Después, las clases concretas de esta familia pueden traducir los modelos de MongoEngine a DTO y viceversa. Finalmente, empleando los modelos de MongoEngine, las clases concretas pueden acceder y modificar el conocimiento de MongoDB.
- **ROS2**: esta familia DAO se basa en almacenar el conocimiento en un nodo ROS 2, que se presentará más adelante. Este nodo utiliza tres servicios de ROS 2 para cada elemento PDDL, uno para hacer consultas, otro para guardar y editar y

otro para eliminar todas las instancias. Las clases concretas de esta familia deben traducir los mensajes de los servicios de ROS 2 a DTO y viceversa. Finalmente, las clases concretas emplean clientes de ROS 2 para ejecutar estos servicios y poder acceder y modificar el conocimiento del nodo ROS 2. El diagrama de clases de esta familia se presenta en la figura C.6 del anexo C.

Para gestionar la creación de instancias de objetos DAO se han usado los patrones Abstract Factory Pattern y Factory Method Pattern. Como resultado, se tiene el diagrama de clases presentado en la figura C.4 del anexo C.

Por un lado, la creación de las instancias DAO se gestiona utilizando el Abstract Factory Pattern. Para usar este patrón se usa una clase factoría base. Después, cada familia DAO tiene una clase factoría concreta para crear sus objetos (MongoDaoFactory y Ros2DaoFactory). Gracias al uso de una factoría base, las factorías de cada familia tendrán las mismas funciones para crear los objetos DAO.

Por otro lado, para gestionar la creación de las instancias factoría de cada familia DAO se ha creado la clase DaoFactoryMethod siguiendo el Factory Method Pattern. Esta clase utiliza una enumeración, DaoFamilies, que asigna un número a su familia DAO correspondiente.

Por último, se tiene la clase ParameterLoader. Esta clase recibe un nodo de ROS 2 y, mediante ciertos parámetros de ROS 2 predefinidos dentro, crea una factoría DAO. Estos parámetros son la familia DAO y la uri de MongoDB. Como se puede acceder al código Python3 de ROS 2 desde cualquier código Python3, los ejecutables ROS 2, que están escritos en Python3, pueden usar la enumeración DaoFamilies y los parámetros predefinidos en el ParameterLoader para elegir la familia DAO. Esto significa que solo con la enumeración se puede cambiar la gestión del conocimiento sin grandes cambios de código.

Paquete kant_knowledge_base

Como se ha explicado anteriormente, la familia DAO ROS2 se basa en utilizar un nodo de ROS 2 que hará de base de conocimientos y almacenará el conocimiento PDDL. El diagrama de clases del paquete de ROS 2 que gestiona esta base de conocimientos se presenta en la figura C.3 del anexo C. Como se puede ver, esta base de conocimientos se basa en el uso de los DTO y mensajes de ROS 2. De esta forma, se tienen las siguientes clases:

- **DtoMsgParser**: esta clase se encarga de traducir de DTO a mensajes de ROS 2.
- **MsgDtoParser**: esta clase se encarga de traducir los mensajes de ROS 2 a DTO.
- **KnowledgeBase**: esta clase se encarga de almacenar los objetos DTO en listas o diccionarios de Python. Además, tiene métodos para acceder, crear, editar y actualizar el conocimiento.
- **KnowledgeBaseNode**: esta clase representa el nodo ROS 2 de la Knowledge Base que hace de intermediario entre la clase KnowledgeBase y el resto de componentes software. De esta forma, para comunicarse con el resto de nodos de ROS 2 se han implementado tres servicios de ROS 2 para cada elemento PDDL, teniendo en total quince servicios. El primer servicio se utiliza para consultar la base de conocimientos, el segundo servicio se utiliza para guardar, editar o eliminar el conocimiento y el tercer servicio se utiliza para eliminar todas las instancias de uno de los elementos de PDDL. Además, este nodo emplea las clases DtoMsgParser y MsgDtoParser para traducir los mensajes que recibe de los servicios y las instancias DTO almacenadas en la KnowledgeBase.

Paquete kant_interfaces

En este apartado se presentan las interfaces de ROS 2 (mensajes, servicios y acciones) que se han creado en KANT. De esta forma, se tienen las siguientes interfaces:

- **Mensajes (msg)**: se han creado varios tipos de mensajes para encapsular el PDDL de forma similar al uso del patrón DTO presentado anteriormente. Se han implementado los siguientes mensajes:
  - **PddlType.msg**: este mensaje encapsula la información de los tipos de PDDL.
  - **PddlObject.msg**: este mensaje encapsula la información de los objetos de PDDL.
  - **PddlPredicate.msg**: este mensaje encapsula la información de los predicados de PDDL.
  - **PddlProposition.msg**: este mensaje encapsula la información de las preposiciones de PDDL.
  - **PddlAction.msg**: este mensaje encapsula la información de las acciones de PDDL.

- **PddlConditionEffect.msg**: este mensaje encapsula la información de las condiciones y efectos de PDDL.
- **UpdateKnowledge.msg**: este mensaje encapsula el tipo de actualización que se quiere hacer sobre la base de conocimientos. Los tipos son SAVE, para guardar o actualizar el conocimiento, y DELETE, para eliminar el conocimiento.

- **Services (srv)**: se han creado varios mensajes de servicios para poder comunicar el nodo de ROS 2 que gestiona la base de conocimientos. Se han definido los siguientes servicios:
  - **GetPddlType.srv**: la request de este servicio contiene el nombre del tipo que se quiere buscar y su response devuelve una lista de tipos PDDL (PddlType.msg).
  - **UpdatePddlType.srv:** la request de este servicio contiene el tipo PDDL (PddlType.msg) que se quiere actualizar y el tipo de actualización (UpdateKnoledge.msg). Su response devuelve si la actualización ha tenido éxito.
  - **GetPddlObject.srv:** la request de este servicio contiene el nombre del objeto que se quiere buscar y su response devuelve una lista de objetos PDDL (PddlObject.msg).
  - **UpdatePddlObject.srv:** la request de este servicio contiene el objeto PDDL (PddlObject.msg) que se quiere actualizar y el tipo de actualización (UpdateKnoledge.msg). Su response devuelve si la actualización ha tenido éxito.
  - **GetPddlPredicate.srv:** la request de este servicio contiene el nombre del predicado que se quiere buscar y su response devuelve una lista de predicados PDDL (PddlPredicate.msg).
  - **UpdatePddlPredicate.srv:** la request de este servicio contiene el predicado PDDL (PddlPredicate.msg) que se quiere actualizar y el tipo de actualización (UpdateKnoledge.msg). Su response devuelve si la actualización ha tenido éxito.
  - **GetPddlProposition.srv:** la request de este servicio contiene el nombre de un predicado y el tipo de búsqueda que se quiere realizar y su response devuelve una lista de proposiciones PDDL (PddlProposition.msg). Los tipos de búsqueda disponibles son ALL, busca todas las proposiciones; GOALS, busca todas las proposiciones que son objetivos; NO_GOALS, busca todas

las proposiciones que no son objetivos; y BY_PREDICATE, busca todas las proposiciones con el predicado enviado en la request.

- **UpdatePddlProposition.srv:** la request de este servicio contiene la proposición PDDL (PddlProposition.msg) que se quiere actualizar y el tipo de actualización (UpdateKnoledge.msg). Su response devuelve si la actualización ha tenido éxito.
- **GetPddlAction.srv:** la request de este servicio contiene el nombre de la acción que se quiere buscar y su response devuelve una lista de acciones PDDL (PddlAction.msg).
- **UpdatePddlAction.srv:** la request de este servicio contiene la acción PDDL (PddlAction.msg) que se quiere actualizar y el tipo de actualización (UpdateKnoledge.msg). Su response devuelve si la actualización ha tenido éxito.

3.3.3. Implementación de YASMIN

En este apartado se presenta la implementación de YASMIN (Yet Another State MachINe). YASMIN es una librería escrita en Python que permite desarrollar comportamientos mediante la creación de máquinas de estados. Por otro lado, YASMIN se puede utilizar para implementar el State Pattern. Este patrón de diseño de comportamiento permite a un objeto modificar su comportamiento cuando su estado cambia. El diagrama de clases de YASMIN se presenta en la figura C.7 del anexo C. Estas clases están organizadas en los siguientes paquetes de ROS 2.

Paquete yasmin

Este paquete consiste en la base de YASMIN. Esta formado por las siguientes clases:

- **Blackboard**: esta clase se basa en el Blackboard Pattern. Este patrón de diseño software incluye el uso de una estructura global de memoria que contiene datos de diferentes módulos. De esta forma, los estados de YASMIN serán los esos módulos que usan el Blackboard para almacenar e intercambiar datos. Por otro lado, las máquinas de estados serán los componentes de control que deciden que módulos se ejecutan.
- **State**: esta clase se basa en el patrón de diseño software State que tiene como objetivo gestionar el comportamiento de un objeto dependiendo del estado actual.

Esta clase es la base para desarrollar nuevos estados de YASMIN. De esta manera, para crear un nuevo estado se hereda de la clase State y se implementa el método execute. Además, se puede redefinir el método cancel_state que permite definir la manera en la que se cancelará el estado. Por último, es necesario definir las salidas del estado (outcomes) que serán strings utilizados posteriormente en la definición de las transiciones de los estados.

- **CbState**: esta clase representa un tipo de estado que recibe una función callback. De esta forma, se puede crear un nuevo estado definiendo una función en vez de una nueva clase.
- **StateMachine**: esta clase representa las máquinas de estados de YASMIN. Mediante la función add_state se pueden añadir nuevos estados. Esta función recibe una instancia de State y un diccionario, que representa las transiciones de ese estado. Una transición consiste en enlazar una de las salidas del estado con otro estado o una de las salidas finales de la máquina de estados. Por último, como StateMachine hereda de State se pueden añadir máquinas de estados anidadas.

Paquete yasmin_ros

Este paquete contiene clases de estados para facilitar la integración de YASMIN con ROS 2. Se tienen las siguientes clases:

- **ActionState**: esta clase representa un estado que ejecuta un cliente de acción de ROS 2. Para usarlo basta con elegir el tipo de la acción y definir la función create_goal, en la que se instancia el goal que se enviará al servidor de la acción. Este estado ya está preparado para cancelar el cliente de acción si se cancela el estado cancelando la acción.
- **Service State**: esta clase representa un estado que ejecuta un cliente de servicio de ROS 2. Para utilizarlo basta con elegir el tipo del servicio y definir la función create_request, en la que se instancia la request que se enviará al servidor del servicio.

Paquete yasmin_viewer

Este paquete de ROS 2 contiene el código necesario para visualizar el estado de las máquinas de estados de YASMIN. Tiene las siguientes clases:

- **YasminViewerPub**: esta clase extrae la información de una máquina de estados y la publica en un topic de ROS 2.
- **YasminViewerNode**: esta clase lee del topic la información de las máquinas de estados y la presenta en el visualizador de máquinas de estados. Este visualizador se basa en una aplicación web formada por un backend, creado en Python3 con Flask [48]; y un frontend, creado con React.js [46]. Para mostrar las máquinas de estados se ha utilizado Cytoscape.js [47], que es una librería de JavaScript utilizada para dibujar grafos. De esta forma, se pueden visualizar las máquinas de estados desde cualquier equipo mediante un navegador web.

Paquete yasmin_interfaces

En este apartado se presentan las interfaces de ROS 2 (mensajes, servicios y acciones) que se han creado en YASMIN. Estos mensajes se han utilizado para publicar información de las máquinas de estados. De esta forma, se tienen las siguientes interfaces:

- **Transition.msg**: este mensaje representa las transiciones de los estados. Cada estado tiene un nombre y, al ejecutarse, produce una salida. De esta forma, este mensaje contiene dos strings, uno para el nombre de la salida y otro para el nombre del estado.
- **State.msg**: este mensaje representa un estado. Contiene la información del estado (StateInfo.msg). Por otro lado, contiene un booleano para mostrar si es una máquina de estados, en este caso anidada. Si es una máquina de estados también contiene una lista de estados (StateInfo.msg) y el nombre de su estado actual.
- **StateInfo.msg**: este mensaje contiene la información de un estado, es decir, su nombre, su lista de transiciones (Transition.msg) y sus salidas.
- **Structure.msg**: este mensaje representa la estructura de la máquina de estados principal. Para ello, contiene una lista de estados (State.msg) y sus salidas finales.
- **StateMachine.msg**: este mensaje representa una máquina de estados. Contiene su nombre, el nombre de su estado actual y su estructura (Structure.msg).

3.3.4. Implementación de MERLIN2

En este apartado se presenta la implementación de la arquitectura cognitiva MERLIN2 a partir de ROS 2, KANT y YASMIN. A continuación se describe la implementación realizada en cada capa en orden descendente.

Mission Layer

Esta capa se encarga de gestionar la generación de objetivos. Su diagrama de clases se presenta en la figura C.8 del anexo C. Como se puede ver, se tienen las siguientes clases:

- **Merlin2GoalDispatcher**: es una clase que encapsula la comunicación de esta capa con la siguiente capa, la Planning Layer. De esta forma, Merlin2GoalDispatcher tiene una factoría DAO creada con el ParameterLoader, que se utilizará para añadir objetivos a la base de conocimientos; y un cliente de acción, para ejecutar la acción del nodo Merlin2ExecutorNode de la Planning Layer.

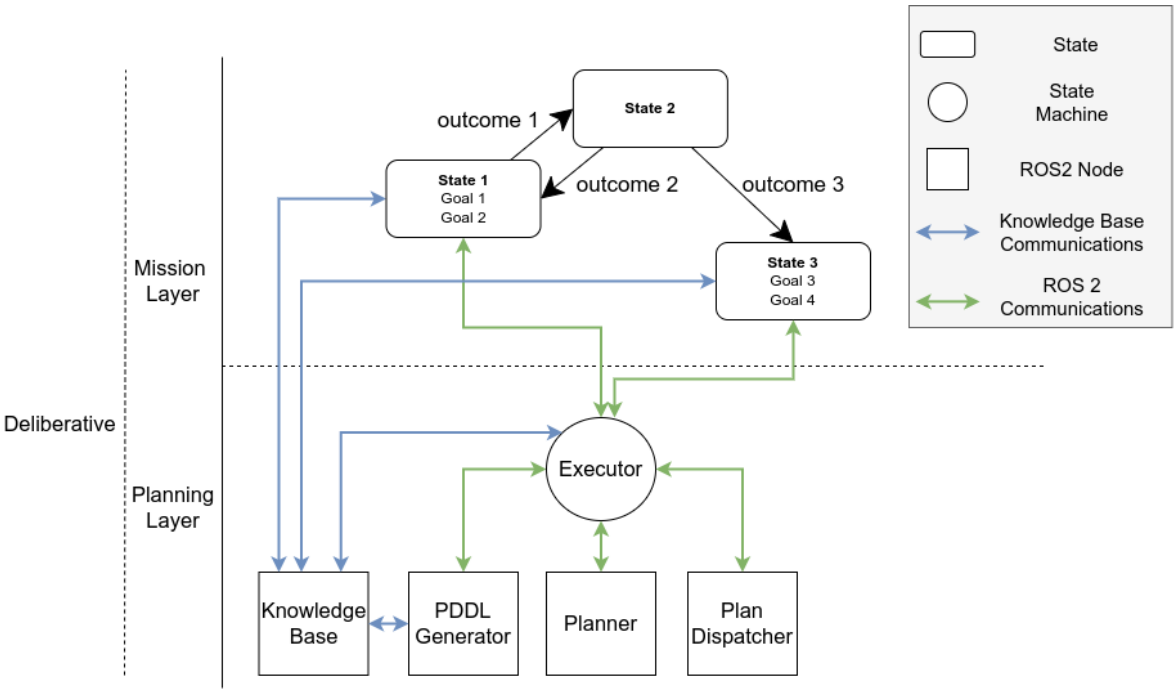


Figura 3.4: Diagrama de comunicación entre la Mission Layer, usando un máquina de estados, y la Planning Layer

- **Merlin2MissionNode**: es una clase abstracta que se emplea para gestionar los objetivos del robot. Para poder usarla basta con crear una nueva clase que herede

de ella y redefina los métodos execute_mission, en el que se crearán los objetivos; create_objects, donde se crearán los objetos iniciales; y create_proposition, donde se crearán las proposiciones iniciales, es decir, el estado inicial. Por otro lado, se tienen las funciones execute_goals y execute_goal, para ejecutar objetivos; y cancel_goals, para cancelar objetivos. Estos métodos emplean la clase Merlin2GoalDispatcher.

- **Merlin2FsmMissionNode**: es una clase utilizada para desarrollar nuevas misiones cuyo funcionamiento está organizado mediante una máquina de estados de YASMIN. De esta forma, cada estado podría ejecutar objetivos diferentes o simplemente procesar datos. Esto se puede ver en la figura 3.4.

Planning Layer

El diagrama de clases de esta capa se presenta en la figura C.9 del anexo C. Las clases presentadas se agrupan en los siguientes paquetes de ROS 2:

- **merlin2_pddl_generator**: Este paquete, que se puede ver en la figura figura C.10 del anexo C, se encarga de extraer el conocimiento de la base de conocimiento y crear el dominio y el problema que los planificadores basados en PDDL usan. Está compuesto por cuatro clases:
  - **Merlin2PddlDomainParser**: esta clase se encarga de generar el texto del dominio PDDL. Para ello, recibe las listas de objetos DTO de tipos, predicados y acciones. Además, se definen una serie de requisitos PDDL, que son: typing, negative-preconditions, durative-action.
  - **Merlin2PddlProblemParser**: esta clase se encarga de generar el texto del problema PDDL. Para ello, recibe las listas de objetos DTO de objetos y proposiciones.
  - **Merlin2PDDLGenerator**: esta clase se encarga de extraer el conocimiento de la base de conocimientos y, mediante las clases Merlin2PddlDomainParser y Merlin2PddlProblemParser, genera los textos de dominio y problema. Para extraer el conocimiento se usa KANT. Por este motivo, esta clase recibe como parámetro la factoría de una familia DAO con la que extraer el conocimiento.
  - **Merlin2PDLLGeneratorNode**: esta clase representa el nodo de ROS 2 que utilizará la clase Merlin2PDDLGenerator para generar el texto PDDL. Por un lado, utiliza el ParameterLoader para crear la factoría DAO con la

que extraer el PDDL. Por otro lado, para comunicarlo con el resto de nodos de ROS 2 se ha creado un servicio que recibe un mensaje vacío y devuelve un mensaje con el dominio y el problema PDDL en formato string.

- **merlin2_planner**: Este paquete, que se puede ver en la figura figura C.11 del anexo C, se encarga de generar los planes a partir del dominio y del problema. Está compuesto por cuatro clases:
  - Merlin2PlannerFactory: esta clase es una factoría, creada siguiendo el patrón de diseño software Factory Method Pattern, que encapsula la creación de los planificadores. Asigna una clase Planner a cada elemento de la enumeración Merlin2Planners.
  - **Merlin2Planners**: esta clase es una enumeración. Contiene un elemento por cada planificador.
  - **Merlin2Planner**: esta clase abstracta es la base para desarrollar nuevos planificadores. Contiene varias funciones para trabajar sobre el plan generado y métodos abstractos que hay que implementar, como el método _generate_plan, para generar el plan; y _parse_plan, para traducir el plan a una lista de PlanAction, que es uno de los mensajes de MERLIN2 para encapsular planes. También se tienen los métodos has_solution, para comprobar si se ha encontrado una solución; get_plan_actions, para obtener la lista de PlanAction; y get_str_plan, para obtener el plan en formato texto.
  - **PopfMerlin2Planner**: esta clase encapsula la ejecución del planificador POPF [41]. Para ello, se crean archivos temporales con el dominio y el problema PDDL y se llama al ejecutable. Posteriormente, se traduce el texto obtenido del ejecutable. Para traducirlo se ha definido el método get_lines_with_actions, que devuelve las líneas en las que hay acciones; y el método parse_action_str, que traduce una acción en formato string a PlanAction.
  - **Merlin2PlannerNode**: esta clase representa el nodo de ROS 2 que utilizará el resto de clases de este paquete. Para comunicarla con el resto de nodos de ROS 2 se ha creado un servicio que recibe un mensaje con el dominio y el problema en formato string y devuelve un mensaje con el plan, que es una lista de PlanAction. PlanAction es un mensaje que contiene el nombre de una acción y los objetos a los que afecta. Por último, este nodo tiene un parámetro de ROS 2 para elegir el planificador.

- **merlin2_plan_dispatcher**: Este paquete está compuesto por una única clase, Merlin2PlanDispatcherNode, que se encarga de ejecutar los planes. Para comunicarse con el resto de nodos de ROS 2 se ha creado un servidor de acción de ROS 2 que recibe el plan. Después de recibir un plan, una lista de PlanAction, tiene que ejecutar cada acción del plan y actualizar el conocimiento con sus efectos.

  Por un lado, para cada acción que se tiene que ejecutar se crea un cliente de acción, ya que las acciones tienen un servidor de acción que inicia su ejecución. A este servidor se le envían los objetos de la base de conocimientos que el planificador ha determinado que son necesarios.

  Por otro lado, para actualizar el conocimiento se usa KANT. Para ello, se utiliza la clase ParameterLoader para obtener una factoría DAO. Mediante esta factoría se instancian los DAOs para las acciones y las proposiciones. Con el DAO de las acciones se obtienen los efectos de una acción con los que se pueden crear las proposiciones que se cumplirán al completar la acción. De esta forma, al completar una acción, se emplea el DAO de las proposiciones para actualizar la base de conocimientos.

- **merlin2_executor**: Este paquete, que se puede ver en la figura figura C.12 del anexo C, se encarga de ejecutar y gestionar los componentes de los paquetes anteriormente explicados. Para ello, se ha utilizado el Facade Pattern, que es un patrón de diseño software basado en proporcionar una interfaz simple a un subsistema complejo, que en este caso es la Planning Layer de MERLIN2. De esta forma, se tienen las siguientes clases:

  - **Merlin2ExecutorNode**: esta es la clase principal del paquete. Es el nodo que hace de fachada. Como se puede ver en el diagrama de clases, se ha implementado como una máquina de estados de YASMIN. Para ejecutar los diferentes componentes de la Planning Layer se usa un estado por componente. Cuando uno de esos estados falla, el nodo entero falla. Como resultado se tiene la máquina de estados presentada en la figura 3.5. El funcionamiento de esta máquina de estados se puede visualizar con YASMIN, lo que permite conocer el estado de la Planning Layer a alto nivel.

  - **Merlin2GeneratePddlState**: esta clase representa el estado encargado de generar el PDDL. Para ello, se tiene un cliente de servicio que ejecutará la generación de PDDL. El resultado obtenido, el dominio y el problema PDDL, se añaden al blackboard de la máquina de estados. También se añade al blackboard si esta tarea ha tenido éxito.

- **Merlin2GeneratePlanState**: esta clase representa el estado encargado de generar el plan. Para ello, se tiene un cliente de servicio que ejecutará el planificador. El dominio y el problema se obtienen del blackboard de la máquina de estados. El resultado obtenido, el plan, también se añaden al blackboard. También se añade al blackboard si esta tarea ha tenido éxito.
- **Merlin2DispatchPlanState**: esta clase representa el estado encargado de activar la ejecución de los planes. Para ello, se tiene un cliente de acción que toma el plan del blackboard y lo envía para ser ejecutado. También se añade al blackboard si esta tarea ha tenido éxito.

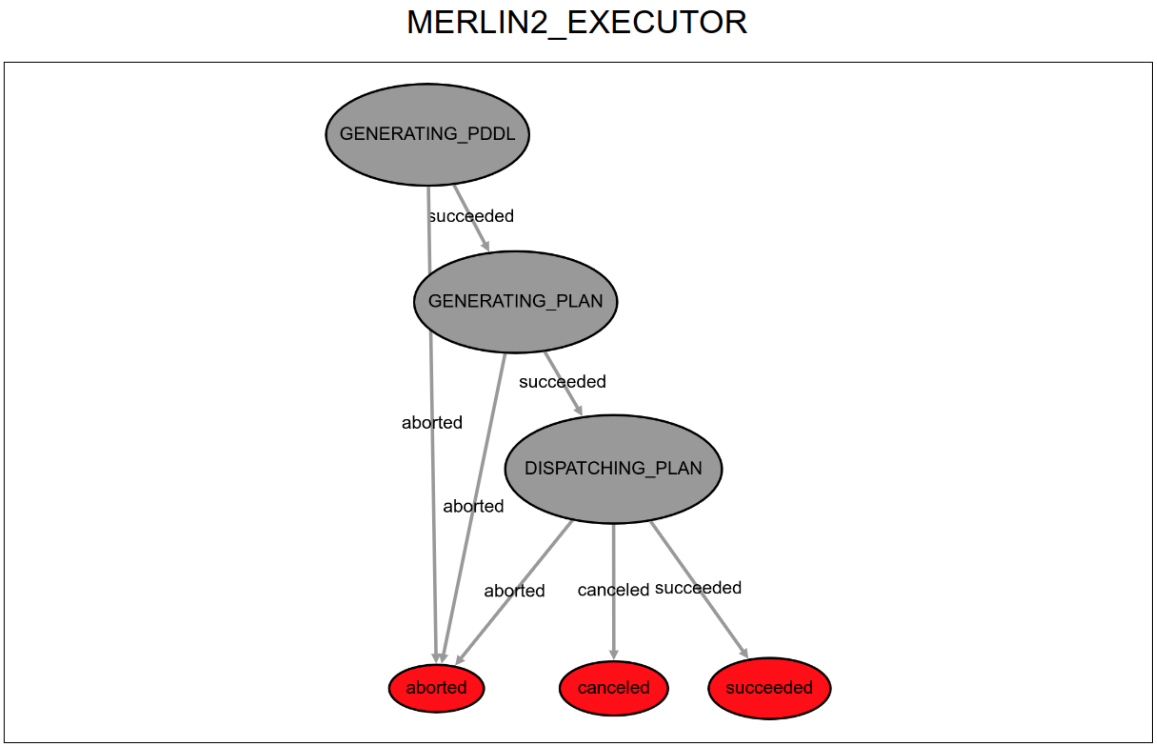


Figura 3.5: Máquina de estados de la clase Merlin2ExecutorNode visualizada en el visualizador de YASMIN.

EXECUTIVE LAYER

La Executive Layer está formada por las acciones de MERLIN2. Las acciones se utilizarán para crear planes que satisfagan los objetivos del robot. De esta forma, estas acciones se tienen que definir en PDDL y se tienen que implementar en Python3 de forma que estén integradas en ROS 2. Para cumplir estas dos condiciones se tienen las clases presentadas en el diagrama de clases de la figura C.13 del anexo C. Se tienen dos clases principales en este diagrama:

- **Merlin2Action**: Como se puede ver en el diagrama de clases, mediante herencia múltiple se consigue que Merlin2Action sea a la vez un nodo de ROS 2 y un PddlActionDto, cumpliendo las dos condiciones anteriores.

  Por un lado, Merlin2Action es un node de ROS 2 por lo que la implementación de la acción está integrada en ROS 2. Para activar las acciones se ha creado un servidor de acción de ROS 2 que tendrá el nombre de la acción y que recibirá mensajes con los objetos PDDL que puede necesitar según el planificador. Además, gracias al uso de acciones de ROS 2 se consigue que las acciones de MERLIN2 se puedan cancelar. Esto hace que se puedan cancelar planes completos de MERLIN2.

  Por otro lado, como Merlin2Action es un PddlActionDto, se pueden definir sus parámetros, sus condiciones y sus efectos. De esta forma, cuando una se instancia, puede registrarse a sí misma en la base de conocimientos, es decir, esa instancia se puede guardar a sí misma como un elemento DTO normal. Además, cuando se destruye la acción se elimina automáticamente de la base de conocimientos.

  De esta forma, si se quieren crear nuevas acciones basta con crear nuevas clases que hereden de Merlin2Action. Después, se tendrán que definir los elementos PDDL, los parámetros, las condiciones y los efectos; usando las clases de KANT explicadas anteriormente. Por último, se tendrá que implementar la acción sobrescribiendo el método run_action de Merlin2Action. Este método recibe la información del planificador y devolverá si la acción ha tenido éxito o no en forma de booleano.

- **Merlin2FsmAction**: Volviendo a usar la herencia múltiple, se ha creado la clase Merlin2FsmAction. Esta clase hereda de Merlin2Action y de la clase StateMachine de YASMIN. De esta forma se consigue tener acciones de MERLIN2 que funcionen como máquinas de estados finitos. Para ejecutar la acción basta con ejecutar la máquina de estados. Además, para cancelar la acción basta con cancelar la máquina de estados que se resume en cancelar el estado que se esté ejecutando en ese momento.

  Para facilitar la creación de nuevas acciones basadas en la clase Merlin2FsmAction se ha desarrollador una factoría que permita instanciar estados básicos que utilizarán los sistemas de la Reactive Layer. Esta factoría está representada por la clase Merlin2StateFactory del diagrama. Se han creado las siguientes clases:

  - **Merlin2SttState**: este estado realiza el reconocimiento del habla (Speech to Text (STT)). Para ello, se ha implementado como una máquina de estados cuyos estados activarán el reconocimiento del habla y comprobarán

el resultado. Si no ha tenido éxito, se avisará mediante síntesis de voz, se calibrará el ruido del ambiente y se volverá a intentar.

- **Merlin2NavigationState**: este estado realiza la navegación. Se basa en activar la navegación topológica que mantiene una lista de puntos y sus coordenadas.
- **Merlin2TtsState**: este estado realiza la síntesis de voz (Text to Speech (TTS)). Se basa en enviar el texto que se quiere decir al sistema de síntesis de voz.

Por último, como las acciones basadas en Merlin2FsmAction son máquinas de estados de YASMIN, se publica su estado y se puede visualizar con el yasmin_viewer. De esta forma, se puede monitorizar el funcionamiento de MERLIN2.

Con todo esto, una vez definidas e implementadas las acciones ya se pueden emplear en la realización de misiones de MERLIN2. Para ejecutar las acciones basta con emplear los métodos normales de ROS 2 ya que son nodos de ROS 2. De esta forma, se puede crear un archivo launch de ROS 2 que ejecute todas las acciones necesarias para cumplir los diferentes objetivos que el robot puede generar en el Mission Layer.

Reactive Layer

Esta capa está formada por los paquetes de ROS 2 que implementan los sistemas encargados de la navegación, el reconocimiento del habla y la síntesis de voz. Cada sistema tiene una interfaz basada en topics, servicios o acciones de ROS 2. Esto hace que otros nodos los puedan usar. Las acciones de MERLIN2 serán las encargadas de ejecutar estos elementos.

El diagrama de clases de esta capa se presenta en la figura C.14 del anexo C. A continuación se explica cada elemento:

- **Navegación Topológica**: se tiene una única clase llamada TopoNavNode, que representa al nodo encargado de realizar la navegación topológica. Cuando se ejecuta este nodo, se cargan los puntos, es decir, los nombres de los puntos y sus coordenadas. Por este motivo, es necesario hacer un archivo YAML [14] de configuración con estos datos. Este archivo contiene una lista de strings. Cada cinco strings se tiene un punto, ya que cada punto necesita un identificador o nombre, las coordenadas (x,y) y la orientación (z,w).

Para comunicar este nodo con el resto de nodos se ha creado un servidor de acción de ROS 2. De esta forma, el servidor recibe el nombre del punto al que se quiere navegar. Con este nombre se buscan las coordenadas y se llama la navegación de ROS 2, es decir, a Nav2 [55]. Cuando Nav2 termina, se recoge su estado y se usa para determinar el estado de éxito del servidor. Por otro lado, si el servidor se cancela se tiene que cancelar Nav2, lo que hace que el robot se pare en su posición actual.

Por último, se han creado varias interfaces de ROS 2. De esta manera, se tienen los siguientes mensajes:

- **Mensajes (msg)**: los mensajes creados son:
  - **Point.msg**: este mensaje representa un punto. Contiene el nombre del punto y su posición. Esta posición consiste en un mensaje geometry_msgs/Pose de ROS 2 que contiene coordenadas.
- **Services (srv)**: los mensajes de servicios creados:
  - **AddPoint.srv**: la request de este mensaje contiene un punto (Point.msg) a añadir y su response contiene un booleano para indicar si se ha sobrescrito el punto.
  - **GetPoints.srv**: la request de este mensaje está vacía y su response contiene una lista de puntos (Point.msg).
  - **GetPoint.srv**: la request de este mensaje contiene el nombre de un punto y su response contiene un punto (Point.msg).
- **Actions (action)**: los mensajes de acciones creados:
  - **TopoNav.action**: el objetivo de esta acción contiene el nombre de un punto al que se quiere navegar con el robot. Su feedback y su resultado está vacío.

- **Síntesis de voz**: para realizar la síntesis de voz se ha empleado el Strategy Pattern. Este patrón de diseño de comportamiento permite definir una serie de algoritmos de la misma familia encapsulando cada uno de ellos en una clase intercambiable. De esta forma, se tiene la familia TtsTool que tiene cuatro estrategias diferentes para hacer la síntesis de voz. Son las siguientes:
  - **ESpeakTtsTool**: esta clase encapsula el uso de eSpeak [49], que es un sintetizador de voz offline de código abierto para inglés y otros idiomas, disponible para Linux y Windows.

- **SpdSayTtsTool**: esta clase encapsula el uso de Speech Dispatcher [50], que es una herramienta que proporciona una interfaz offline independiente del dispositivo para llevar a cabo síntesis de voz de manera simple y estable.
- **FestivalTtsTool**: esta clase encapsula el uso de Festival [51], que es un framework que aporta un conjunto de APIs para realizar síntesis de voz offline. Se puede usar como comandos de shell y desde C ++ y Java. Soporta varios idiomas como inglés y español.
- **GTtsTtsTool**: esta clase encapsula el uso de gTTS (Google Text-to-Speech) [52], que es la librería de Python3 para utilizar la API de síntesis de voz de Google Translate. Es una herramienta online que devuelve la voz en formato mp3. Para reproducir el audio generado se usa mpg321 [56], un reproductor de mp3 que se puede usar desde la línea de comandos en Linux.

Para integrar la síntesis de voz se ha creado la clase TtsNode que es el nodo de ROS 2 que lo encapsula. Este nodo tiene un servidor de acción de ROS 2 que recibirá el texto que se quiere decir, la herramienta que se usará y su configuración. Como se ha usado un servidor de acción, la síntesis de voz se puede cancelar en cualquier momento.

Por último, se han creado varias interfaces de ROS 2. De esta forma, se tienen los siguientes mensajes:

- **Mensajes (msg)**: los mensajes creados son:
  - **Config.msg**: este mensaje representa la configuración a usar en la síntesis de voz. Contiene la herramienta que se quiere usar, el volumen, la frecuencia, el idioma y el género de la voz.
- **Actions (action)**: los mensajes de acciones creados:
  - **TTS.action**: el objetivo de esta acción contiene el texto que se quiere decir y su configuración (Config.msg). Su feedback y su resultado está vacío.

- **Reconocimiento del habla**: para realizar el reconocimiento del habla se ha empleado el Chain of Responsibility Pattern. Este patrón de diseño de comportamiento se basa en tener una cadena de manejadores que procesarán la información. Una vez procesada, cada manejador pasa su resultado al siguiente manejador. De esta forma, se tienen las siguientes clases que formarían parte de la cadena:

- **STTNode**: esta clase es un nodo de ROS 2 que usa la libraría de Python SpeechRecognition [57] para reconocer el habla. Esta librería puede acceder al micrófono y recoger el audio. Para escuchar del micrófono se utiliza un umbral de ruido. De esta manera, cuando se supera ese umbral se empieza a escuchar y cuando se desciende de ese umbral se detiene. Por este motivo, para poder detener esto, se utiliza un thread que se encarga de escuchar. Así se puede parar de escuchar deteniendo el thread.

  SpeechRecognition puede usar Google o PockeSphinx. Esto se controla mediante el uso de parámetros de ROS 2. De esta manera, se tiene una opción online, Google que utiliza su API; y una opción offline, PockeSphinx. Para usar PockeSphinx es necesario crear una gramática, que es un conjunto de reglas que definen las posibles combinaciones de frases que pueden tener lugar. Estas gramáticas utilizan el formato JSGF [5].

  Por último, este nodo tiene varias comunicaciones de ROS 2. Primero, se tienen dos servicios para iniciar y parar el reconocimiento. También se tiene un servicio para calibrar el umbral del ruido del ambiente. Para terminar, se tiene un topic en el que se publica el resultado del reconocimiento.

- **NLPNode**: esta clase es un nodo de ROS 2 que usa la libraría nltk [58] para realizar procesamiento de lenguaje natural. El procesamiento llevado a cabo es simple. Solo trata frases en inglés en las que procesa los nombres, los números cardinales, los pronombres, los verbos, los adjetivos, las preposiciones y las conjunciones. Para comunicarse con los otros nodos utiliza un topic en el que publica sus resultados.

- **ParserNode**: esta clase es un nodo de ROS 2 que usa la librería jsgf [5]. Esta librería se basa en gramáticas con formato JSGF cuyas reglas tienen etiquetas. De esta forma, para un texto dado, el nodo usa la librería para buscar las reglas de la gramática que se cumplen. Después, de cada regla devuelve las etiquetas asociadas. Para terminar, se tiene un topic en el que se publica el resultado del reconocimiento.

- **DialogManagerNode**: esta clase es un nodo de ROS 2 que forma parte de la cadena y también actúa como fachada del sistema de reconocimiento del habla siguiendo el Facade Pattern. Para hacer de fachada utiliza un servidor de acción de ROS 2. Este servidor emplea los servicios y topics de las clases anteriormente explicadas. Como resultado devuelve el texto después de haber sido tratado por los otros nodos.

Por último, se han creado varias interfaces de ROS 2 para comunicar los nodos. De esta forma, se tienen los siguientes mensajes:

- **Mensajes (msg)**: los mensajes creados son:
  - **StringArray.msg**: este mensaje representa una lista de strings.
- **Services (srv)**: los mensajes de servicios creados:
  - **Empty.srv**: este mensaje de servicio pertenece a ROS 2. Su request y su response están vacías. Se ha utilizado para activar el calibrado de ruido del entorno, para iniciar la escucha y para detenerla.
- **Actions (action)**: los mensajes de acciones creados:
  - **ListenOnce.action**: el objetivo de esta acción contiene un booleano para indicar si es necesario calibrar el ruido del entorno. Su feedback está vacío y su resultado consiste en una lista de strings (StringArray.msg).

Interfaces de ROS 2

En este apartado se presentan las interfaces de ROS 2 (mensajes, servicios y acciones) que se han creado en MERLIN2. De esta forma, se tienen las siguientes interfaces:

- **Mensajes (msg)**: los mensajes creados son:
  - **PlanAction.msg**: este mensaje representa una acción del plan creado en Merlin2PlannerNode. Contiene el nombre de la acción y los nombres de los objetos necesarios.
- **Services (srv)**: los mensajes de servicios creados:
  - **GeneratePddl.srv**: la request de este mensaje es vacía, no tiene contenido, y su response contiene el texto PDDL del dominio y el problema generados en Merlin2PDLLGeneratorNode.
  - **GeneratePlan.srv**: la request de este mensaje contiene el texto PDDL del dominio y el problema y su response contiene una lista de acciones (PlanAction.msg) y un booleano para indicar si el planificador ha encontrado una solución.
- **Actions (action)**: los mensajes de acciones creados:
  - **DispatchPlan.action**: el objetivo de esta acción contiene una lista de acciones (PlanAction.msg) que se tienen que ejecutar. Su feedback y su resultado

están vacíos. Se usa para activar el merlin2_plan_dispatcher y ejecutar las acciones definidas en el plan creado.

- **DispatchAction.action**: el objetivo de esta acción contiene una acción (PlanAction.msg) que se tiene que ejecutar. Su feedback y su resultado están vacíos. Se usa para activar una Merlin2Action.
- **Execute.action**: esta acción se utiliza para activar el Merlin2ExecutorNode, que gestiona los componentes de la Planning Layer. Su objetivo y su feedback están vacíos. Su resultado consiste en tres booleanos para indicar si la generación de PDDL, el planificador y la ejecución del plan han tenido éxito.

3.3.5. Demos

Para probar el funcionamiento general de MERLIN2 se han realizado dos demos en el paquete de ROS 2 merlin2_demo. El diagrama de clases de estas demos se presenta en la figura C.15 del anexo C. A continuación se explican estas dos demos.

Demo 1

Esta demo consiste en que el robot tiene que atender a una persona que está en una zona de un apartamento. Para atenderla, el robot tendrá que moverse a la zona del apartamento que la persona diga. Con esto se quiere comprobar el funcionamiento general de las capas de MERLIN2 y las comunicaciones entre los diferentes componentes.

Para desarrollar esta demo se han empleado las siguientes clases del diagrama de clases:

- **Merlin2DemoNode**: esta clase es un nodo de ROS 2 que se encarga de definir el estado inicial y los objetivos del robot. El estado inicial consiste en los objetos del problema, la persona y las zonas del apartamento; y sus características, es decir, las proposiciones que definen la zona en la que está el robot y en al que está la persona. Para realizar todo esto se utiliza KANT. Por otro lado, el objetivo que se quiere cumplir es atender a la persona.
- **Merlin2NavigationAction**: esta clase es la acción de MERLIN2 para hacer que el robot navegue. Tiene como condición que el robot esté en una zona origen y como efecto que termine en una zona destino. De esta forma, esta es la acción para mover al robot por las diferentes zonas del apartamento.

- **Merlin2NavigationFsmAction**: esta es la misma acción presentada antes, pero implementada como una máquina de estados de YASMIN. Como es una máquina de estados de YASMIN, se puede visualizar en el visualizador desarrollado. Esto se presenta en la figura 3.6. Como se puede ver, su primer estado, PREPARING_GOAL, se encarga de preparar el objetivo que se enviará al servidor de acción de la navegación topológica. Después, en el estado NAVIGATING se enviará este objetivo mediante un cliente de acción y se esperará por el resultado. Como salidas finales se tienen éxito (succeeded), abortado (aborted), y cancelado (canceled), que son los estados finales de las acciones de ROS 2.

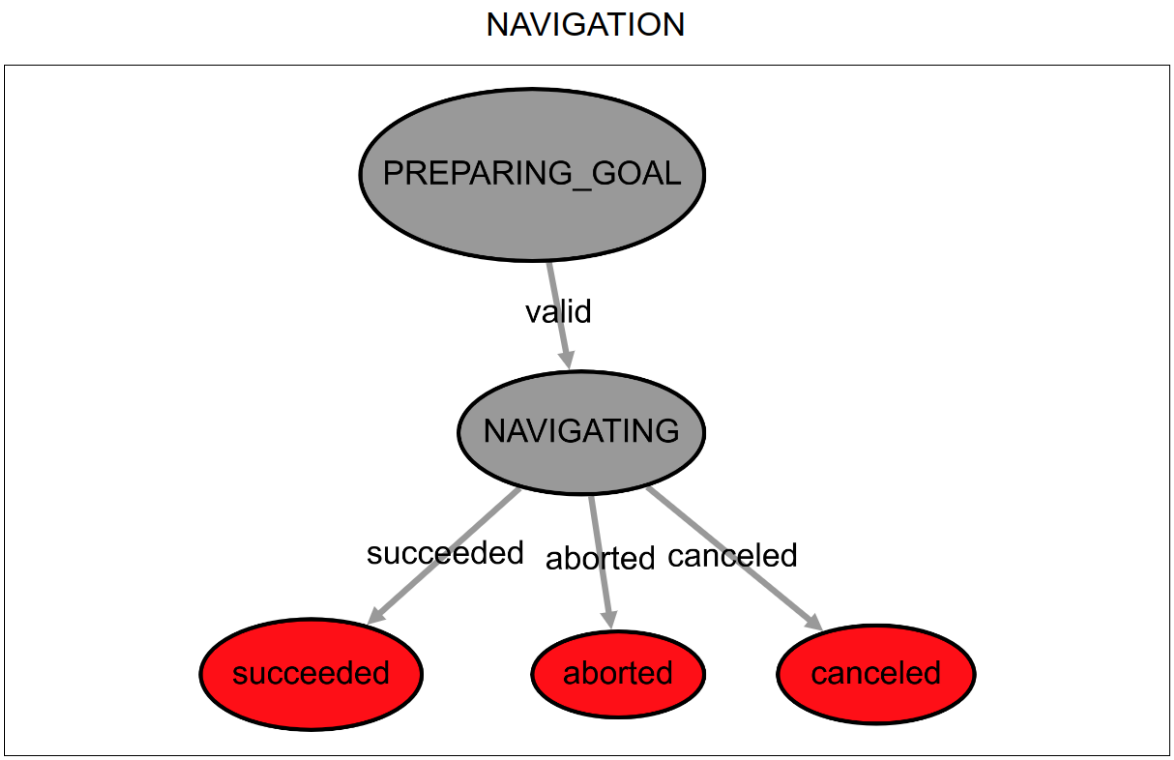


Figura 3.6: Máquina de estados de la clase Merlin2NavigationFsmAction visualizada en el visualizador de YASMIN.

- **Merlin2HiNavigationFsmAction**: esta clase es la acción de MERLIN2 para interactuar con la persona y preguntarle la zona a la que quiere que vaya el robot. Después de que la persona haya comunicado la nueva zona, esta acción también moverá al robot. La máquina de estados de esta acción se presenta en la figura 3.7. Como se puede ver, su primer estado, ASKING, usará síntesis de voz para preguntar la zona a la que moverse. Después, se utilizará la máquina de estados anidada LISTENING para escuchar la respuesta de la persona. A continuación, se comprueba que el resultado es correcto. Si no es correcto se

repite la pregunta y se vuelve a escuchar. Si es correcta, se navega hasta la zona especificada.

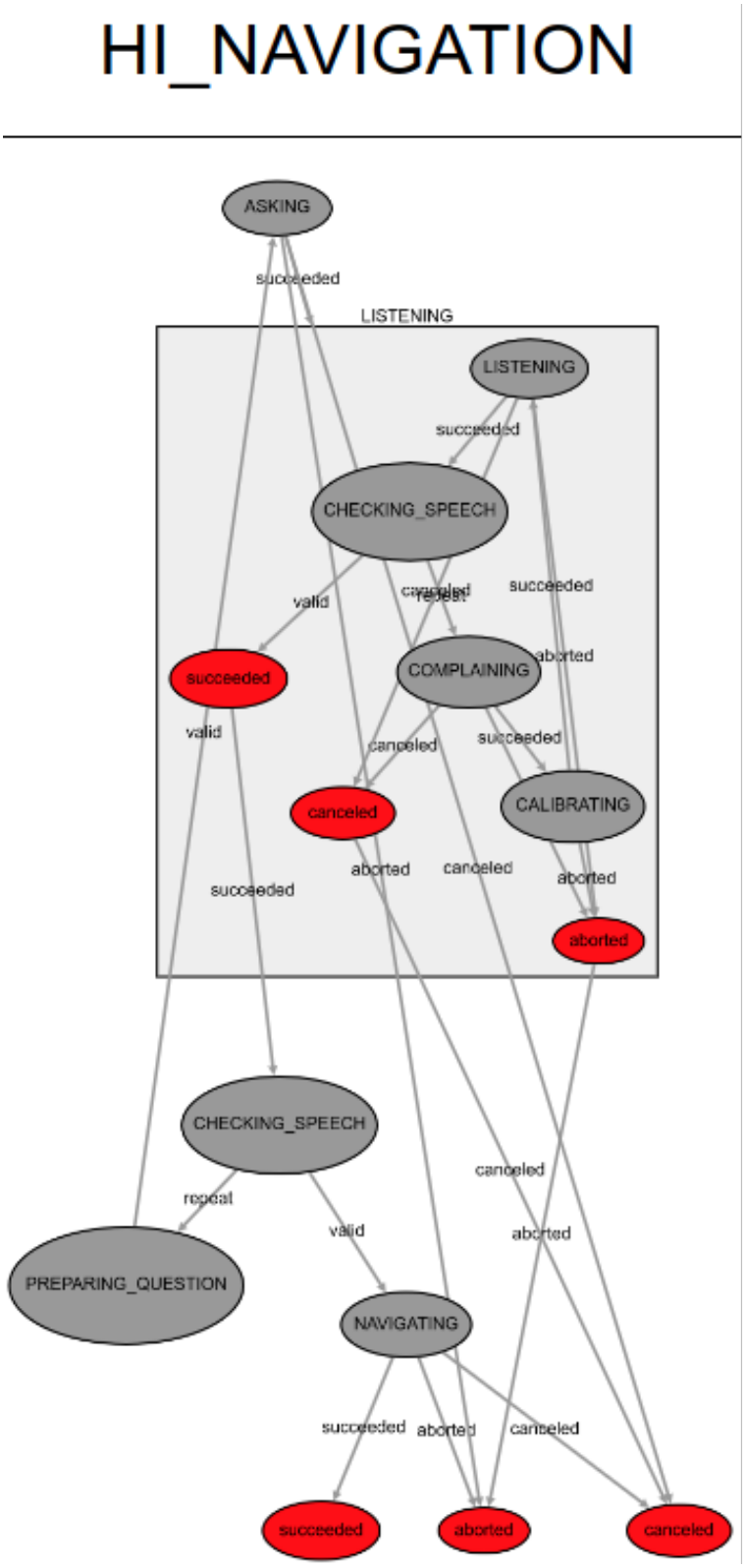


Figura 3.7: Máquina de estados de la clase Merlin2HiNavigationFsmAction visualizada en el visualizador de YASMIN.

Demo 2

Esta demo consiste en repetir la prueba descrita en [15], en la que un robot tiene que vigilar los diferentes puntos de un apartamento. Con esto se quiere comprobar que MERLIN2 se puede cancelar correctamente para cambiar los objetivos del robot y comparar los resultados obtenidos.

Para desarrollar esta demo se han empleado las siguientes clases del diagrama de clases:

- **Merlin2Demo2Node**: esta clase es un nodo de ROS 2 que se encarga de definir el estado inicial y los objetivos del robot. El estado inicial consiste en los objetos del problema, las zonas del apartamento; y sus características, es decir, las proposiciones que definen la zona en la que está el robot. Para realizar todo esto se utiliza KANT. Por otro lado, se encarga de generar objetivos y cancelarlos, tal y como se explica en [15]. Los objetivos consisten en comprobar zonas del apartamento. Por otro lado, como se puede ver, hereda de Merlin2FsmMissionNode, por lo que es una máquina de estados de YASMIN. La máquina resultante se presenta en la figura 3.8.

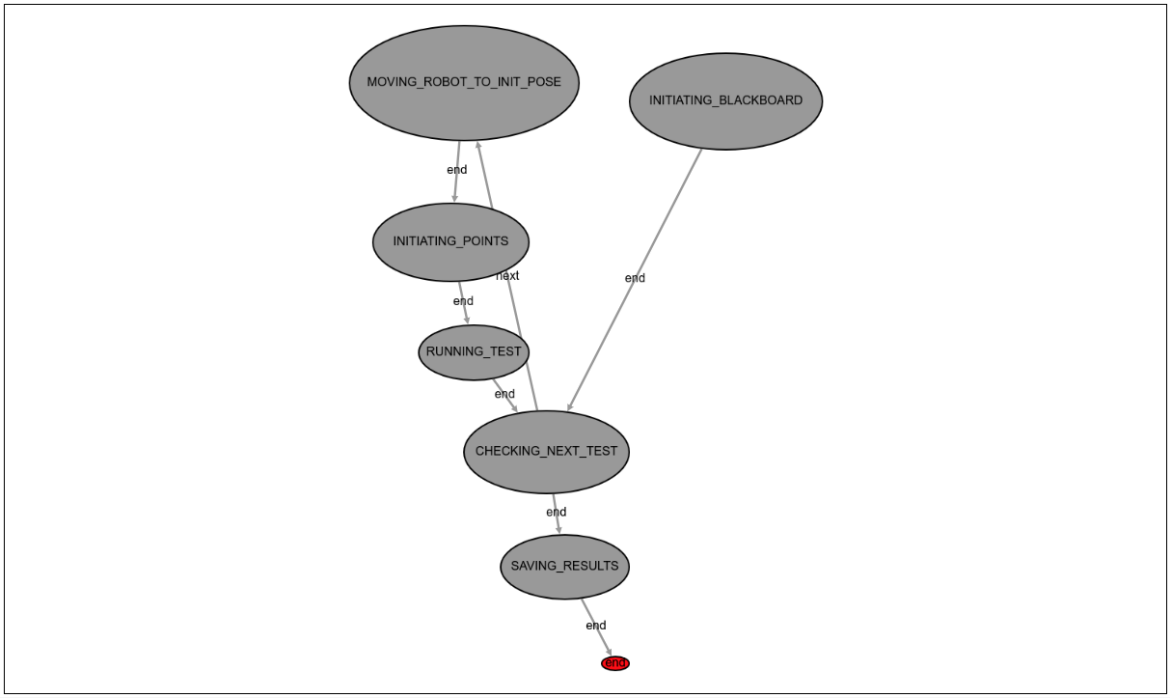


Figura 3.8: Máquina de estados de la clase Merlin2Demo2Node visualizada en el visualizador de YASMIN.

Se usan diferentes estados para controlar el funcionamiento de la misión. Su funcionamiento se basa en comprobar si quedan tests por hacer. Si hay tests, se lleva al robot a su posición inicial usando MERLIN2, es decir, se crea el objetivo de que el robot esté en la posición inicial y se ejecuta. Después, se crea la lista de puntos aleatorios a los que se moverá el robot, definiendo cuáles se van a cancelar. Posteriormente, se ejecuta un test, que consiste en crear y ejecutar objetivos con la lista mencionada antes. Por último, cuando ya no queden tests, se guardarán los datos de tiempos y distancias en un archivo CSV para su posterior análisis.

- **Merlin2NavigationFsmAction**: esta clase es la acción de MERLIN2 para hacer que el robot navegue explicada anteriormente.
- **Merlin2CheckWpAction**: esta clase es la acción de MERLIN2 que simula la comprobación de la zona. Se basa en utilizar la síntesis de voz para anunciar que ha comprobado el punto. Como condición se tiene que el robot tiene que estar en la zona que se quiere comprobar y como efecto final se tiene que la zona es comprobada.
- **Merlin2CheckWpFsmAction**: esta clase es la misma acción anterior, Merlin2CheckWpAction, pero implementada como una máquina de estados. La máquina de estados resultante se presenta en la figura 3.9.

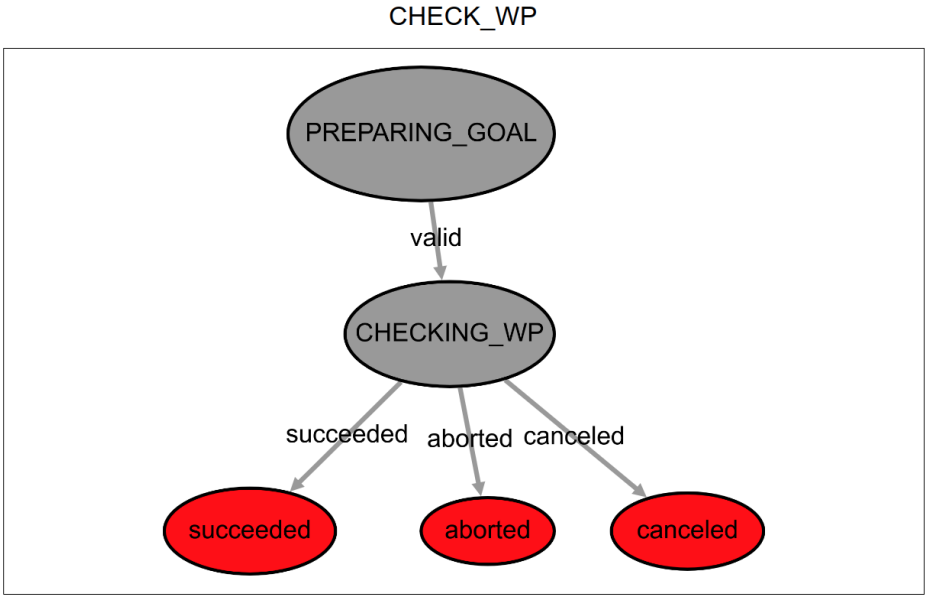


Figura 3.9: Máquina de estados de la clase Merlin2CheckWpFsmAction visualizada en el visualizador de YASMIN.

3.4. Preparación del entorno virtual

Para poder realizar pruebas sobre entornos virtuales se ha migrado el simulador del robot RB1 a ROS 2 Foxy. Para ello ha sido necesario utilizar las partes necesarias de su modelo, que está escrita en URDF [59]. Además, se han tenido que usar nuevos controladores y plugins de ROS 2 para Gazebo [3]. De esta forma, se han utilizado plugins que hay por defecto para simular el lídar, la cámara y el IMU del robot RB1. Por otro lado, como controlador se ha usado un controlador diferencial. Este tipo de controlador se emplea en robots con dos ruedas separadas situadas a ambos lados del robot.

Después de completar la migración del robot RB1 simulado, se ha configurado Nav2 [55], el sistema de navegación de ROS 2. Esto consiste en especificar los topics de los sensores necesarios, que son el sensor del lídar, de la odometría y del IMU; y el topic para mover al robot. Además, es necesario configurar las velocidades y aceleraciones máximas y mínimas. Por último, se integró el RB1 simulado en el mundo simulado descrito en [15]. Esto se puede ver en la figura 3.10.

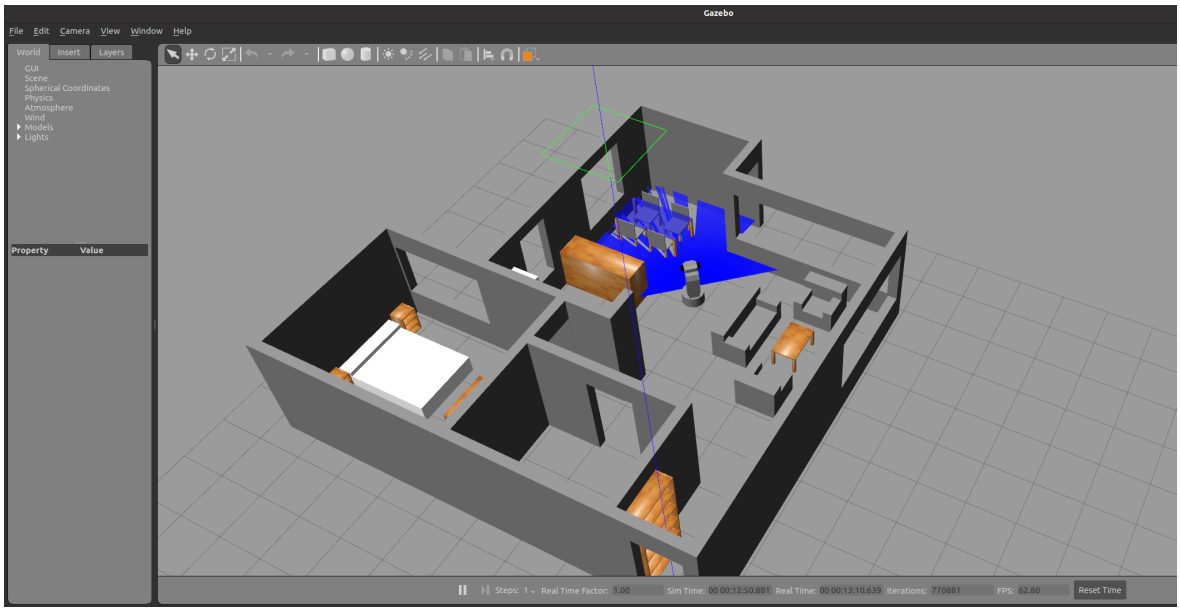


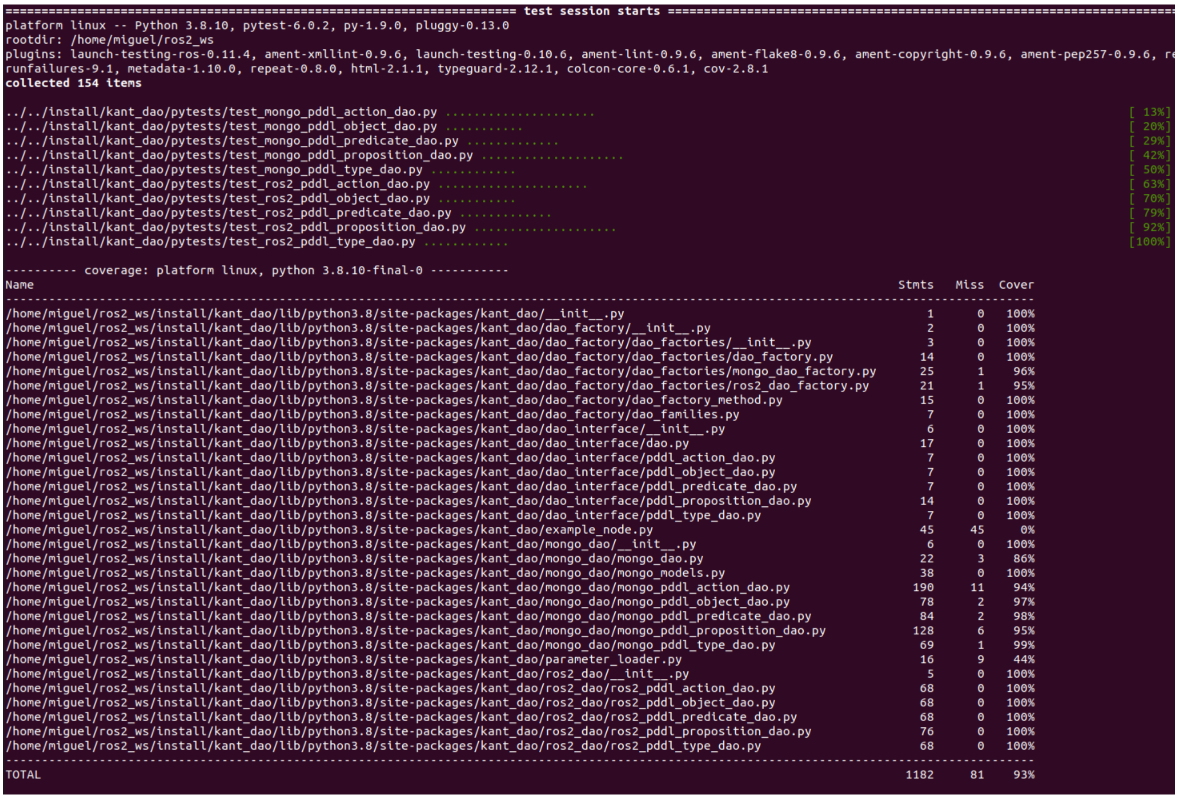
Figura 3.10: Simulación en Gazebo del robot RB1 en el mundo Granny Annie.

3.5. Pruebas

Para realizar análisis sobre el código del proyecto se ha utilizado la herramienta pylint [10]. Esta herramienta analiza el código Python buscando errores de codificación y de estilo. Además, se han realizado tests unitarios sobre el código desarrollado.

3.5.1. Tests unitarios

Para realizar los tests unitarios se han utilizado las librerías de Python unittest [13] y pytest [8]. Los tests efectuados se centran en probar los componentes más importantes de los paquetes de ROS 2 desarrollados. En total se han realizado 313 tests obteniendo una buena cobertura de los componentes que se quieren tratar. De esta forma, se han realizado los tests presentados en las figuras 3.12, 3.11, 3.13, 3.14, 3.15, 3.16, 3.17, 3.18 y 3.19. La mayoría de los archivos que no llegan al 100 % de cobertura se debe a que son clases abstractas, nodos de ROS 2 o código que depende de ROS 2 y que no se testea inicialmente en los tests unitarios.



```
======================================================================= test session starts ========================================================================
platform linux -- Python 3.8.10, pytest-6.0.2, py-1.9.0, pluggy-0.13.0
rootdir: /home/miguel/ros2_ws
plugins: launch-testing-ros-0.11.4, ament-xmllint-0.9.6, launch-testing-0.10.6, ament-lint-0.9.6, ament-flake8-0.9.6, ament-copyright-0.9.6, ament-pep257-0.9.6, re
runfailures-9.1, metadata-1.10.0, repeat-0.8.0, html-2.1.1, typeguard-2.12.1, colcon-core-0.6.1, cov-2.8.1
collected 154 items

../../install/kant_dao/pytests/test_mongo_pddl_action_dao.py .....................    [ 13%]
../../install/kant_dao/pytests/test_mongo_pddl_object_dao.py ...........              [ 20%]
../../install/kant_dao/pytests/test_mongo_pddl_predicate_dao.py .............         [ 29%]
../../install/kant_dao/pytests/test_mongo_pddl_proposition_dao.py ....................  [ 42%]
../../install/kant_dao/pytests/test_mongo_pddl_type_dao.py ............               [ 50%]
../../install/kant_dao/pytests/test_ros2_pddl_action_dao.py .....................     [ 63%]
../../install/kant_dao/pytests/test_ros2_pddl_object_dao.py ...........               [ 70%]
../../install/kant_dao/pytests/test_ros2_pddl_predicate_dao.py .............          [ 79%]
../../install/kant_dao/pytests/test_ros2_pddl_proposition_dao.py ....................   [ 92%]
../../install/kant_dao/pytests/test_ros2_pddl_type_dao.py ............                [100%]

---------- coverage: platform linux, python 3.8.10-final-0 -----------
Name                                                                                                      Stmts   Miss  Cover
-----------------------------------------------------------------------------------------------------------------------------
/home/miguel/ros2_ws/install/kant_dao/lib/python3.8/site-packages/kant_dao/__init__.py                          1      0   100%
/home/miguel/ros2_ws/install/kant_dao/lib/python3.8/site-packages/kant_dao/dao_factory/__init__.py              2      0   100%
/home/miguel/ros2_ws/install/kant_dao/lib/python3.8/site-packages/kant_dao/dao_factory/dao_factories/__init__.py   3   0   100%
/home/miguel/ros2_ws/install/kant_dao/lib/python3.8/site-packages/kant_dao/dao_factory/dao_factories/dao_factory.py   14   0   100%
/home/miguel/ros2_ws/install/kant_dao/lib/python3.8/site-packages/kant_dao/dao_factory/dao_factories/mongo_dao_factory.py   25   1    96%
/home/miguel/ros2_ws/install/kant_dao/lib/python3.8/site-packages/kant_dao/dao_factory/dao_factories/ros2_dao_factory.py   21   1    95%
/home/miguel/ros2_ws/install/kant_dao/lib/python3.8/site-packages/kant_dao/dao_factory/dao_factory_method.py   15      0   100%
/home/miguel/ros2_ws/install/kant_dao/lib/python3.8/site-packages/kant_dao/dao_factory/dao_families.py          7      0   100%
/home/miguel/ros2_ws/install/kant_dao/lib/python3.8/site-packages/kant_dao/dao_interface/__init__.py           6      0   100%
/home/miguel/ros2_ws/install/kant_dao/lib/python3.8/site-packages/kant_dao/dao_interface/dao.py               17      0   100%
/home/miguel/ros2_ws/install/kant_dao/lib/python3.8/site-packages/kant_dao/dao_interface/pddl_action_dao.py    7      0   100%
/home/miguel/ros2_ws/install/kant_dao/lib/python3.8/site-packages/kant_dao/dao_interface/pddl_object_dao.py    7      0   100%
/home/miguel/ros2_ws/install/kant_dao/lib/python3.8/site-packages/kant_dao/dao_interface/pddl_predicate_dao.py   7     0   100%
/home/miguel/ros2_ws/install/kant_dao/lib/python3.8/site-packages/kant_dao/dao_interface/pddl_proposition_dao.py   14   0   100%
/home/miguel/ros2_ws/install/kant_dao/lib/python3.8/site-packages/kant_dao/dao_interface/pddl_type_dao.py      7      0   100%
/home/miguel/ros2_ws/install/kant_dao/lib/python3.8/site-packages/kant_dao/example_node.py                   45     45     0%
/home/miguel/ros2_ws/install/kant_dao/lib/python3.8/site-packages/kant_dao/mongo_dao/__init__.py              6      0   100%
/home/miguel/ros2_ws/install/kant_dao/lib/python3.8/site-packages/kant_dao/mongo_dao/mongo_dao.py            22      3    86%
/home/miguel/ros2_ws/install/kant_dao/lib/python3.8/site-packages/kant_dao/mongo_dao/mongo_models.py         38      0   100%
/home/miguel/ros2_ws/install/kant_dao/lib/python3.8/site-packages/kant_dao/mongo_dao/mongo_pddl_action_dao.py   190   11    94%
/home/miguel/ros2_ws/install/kant_dao/lib/python3.8/site-packages/kant_dao/mongo_dao/mongo_pddl_object_dao.py    78    2    97%
/home/miguel/ros2_ws/install/kant_dao/lib/python3.8/site-packages/kant_dao/mongo_dao/mongo_pddl_predicate_dao.py   84   2    98%
/home/miguel/ros2_ws/install/kant_dao/lib/python3.8/site-packages/kant_dao/mongo_dao/mongo_pddl_proposition_dao.py   128   6    95%
/home/miguel/ros2_ws/install/kant_dao/lib/python3.8/site-packages/kant_dao/mongo_dao/mongo_pddl_type_dao.py      69    1    99%
/home/miguel/ros2_ws/install/kant_dao/lib/python3.8/site-packages/kant_dao/parameter_loader.py               16      9    44%
/home/miguel/ros2_ws/install/kant_dao/lib/python3.8/site-packages/kant_dao/ros2_dao/__init__.py               5      0   100%
/home/miguel/ros2_ws/install/kant_dao/lib/python3.8/site-packages/kant_dao/ros2_dao/ros2_pddl_action_dao.py     68      0   100%
/home/miguel/ros2_ws/install/kant_dao/lib/python3.8/site-packages/kant_dao/ros2_dao/ros2_pddl_object_dao.py     68      0   100%
/home/miguel/ros2_ws/install/kant_dao/lib/python3.8/site-packages/kant_dao/ros2_dao/ros2_pddl_predicate_dao.py   68     0   100%
/home/miguel/ros2_ws/install/kant_dao/lib/python3.8/site-packages/kant_dao/ros2_dao/ros2_pddl_proposition_dao.py   76   0   100%
/home/miguel/ros2_ws/install/kant_dao/lib/python3.8/site-packages/kant_dao/ros2_dao/ros2_pddl_type_dao.py       68      0   100%
-----------------------------------------------------------------------------------------------------------------------------
TOTAL                                                                                                       1182     81    93%
```

Figura 3.11: Ejecución e informe de cobertura de los tests del paquete kant_dao.

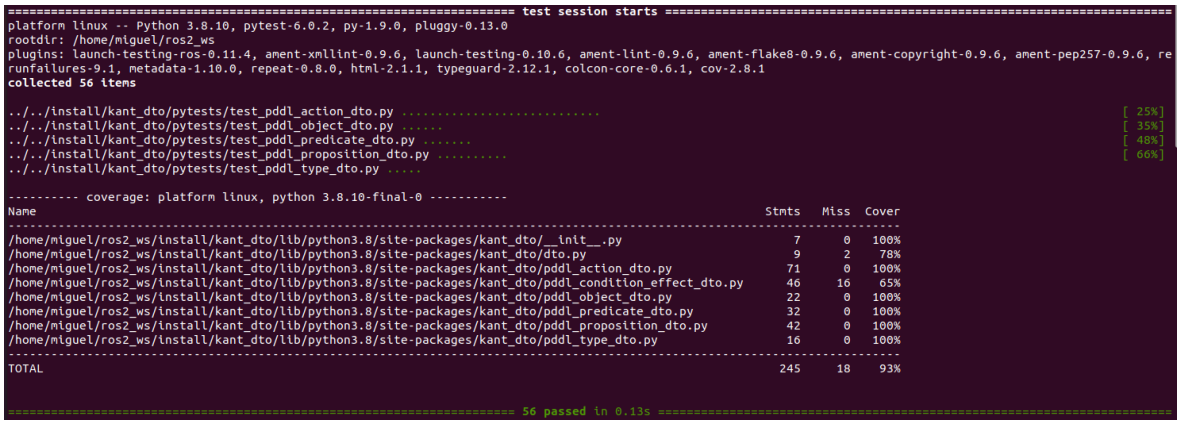


Figura 3.12: Ejecución e informe de cobertura de los tests del paquete kant_dto.

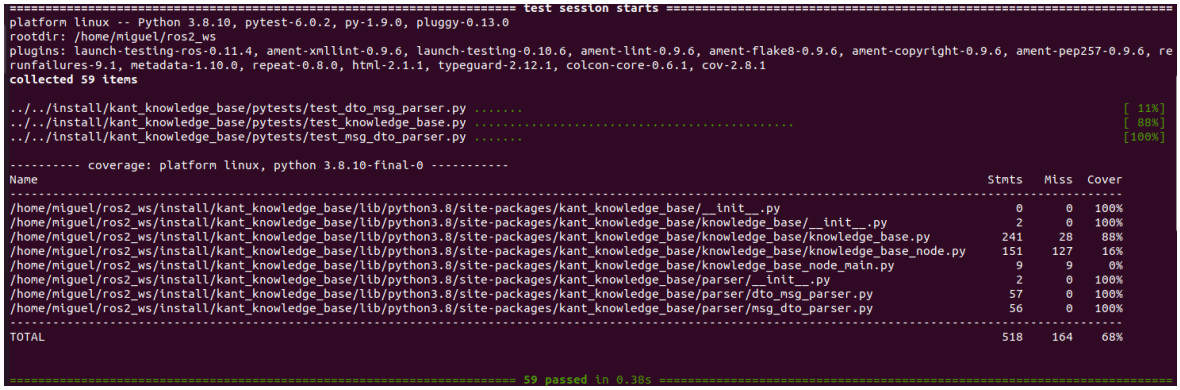


Figura 3.13: Ejecución e informe de cobertura de los tests del paquete kant_knowledge_base.

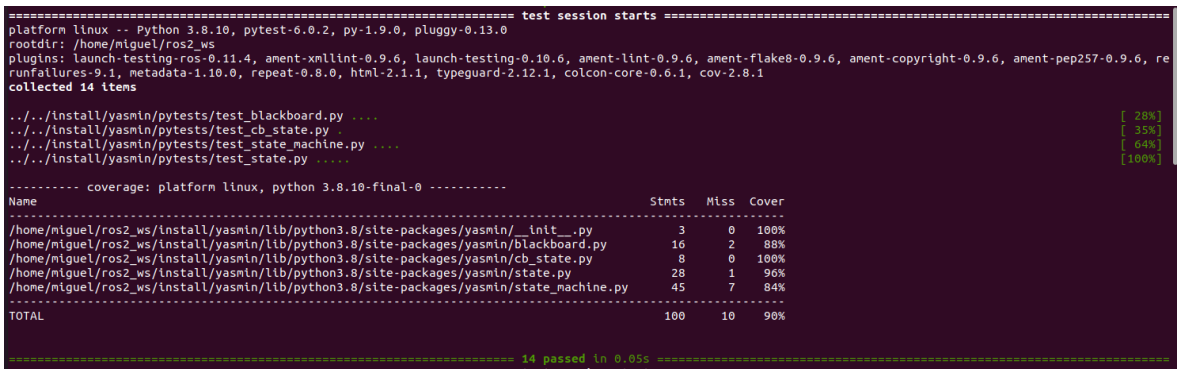


Figura 3.14: Ejecución e informe de cobertura de los tests del paquete yasmin.

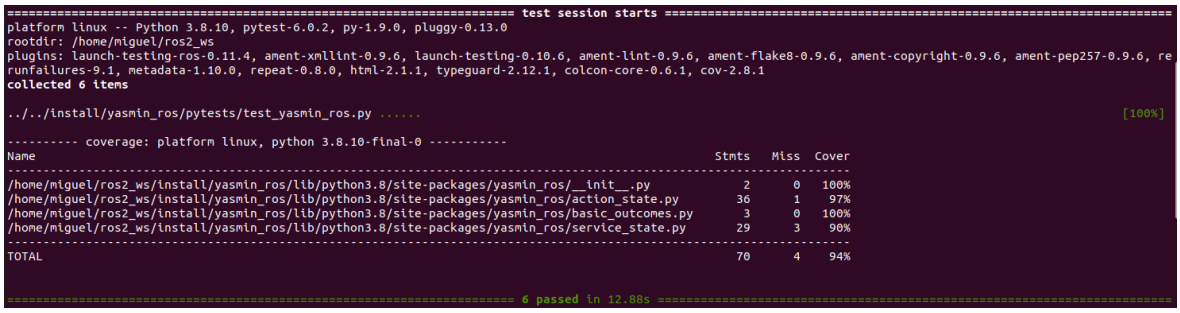


Figura 3.15: Ejecución e informe de cobertura de los tests del paquete yasmin_ros.

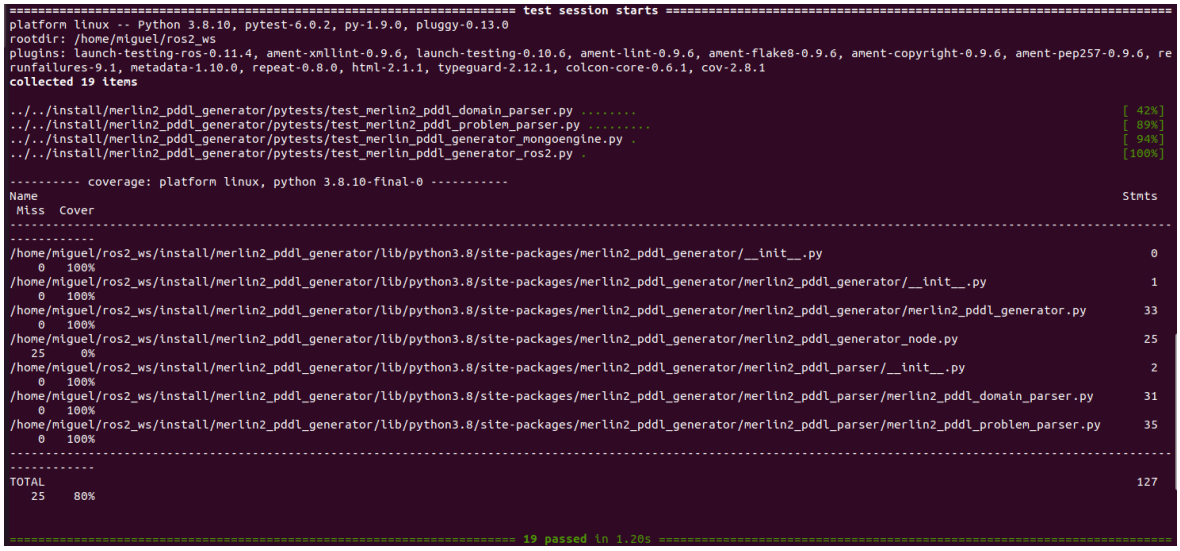


Figura 3.16: Ejecución e informe de cobertura de los tests del paquete merlin2_pddl_generator.

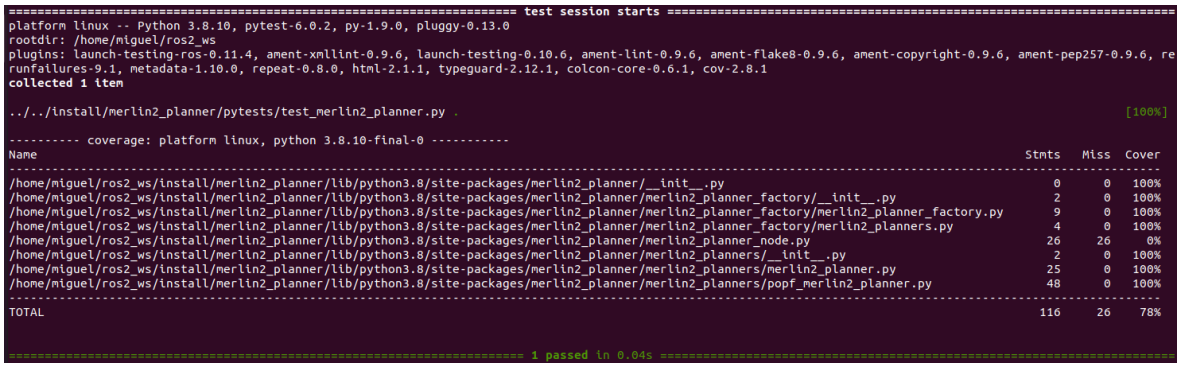


Figura 3.17: Ejecución e informe de cobertura de los tests del paquete merlin2_planner.

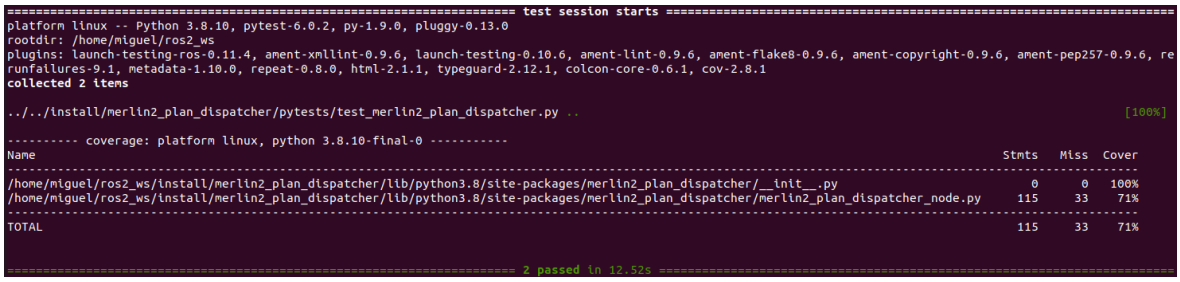


Figura 3.18: Ejecución e informe de cobertura de los tests del paquete merlin2_plan_dispatcher.

Figura 3.19: Ejecución e informe de cobertura de los tests del paquete merlin2_action.

3.5.2. Tests funcionales

Estos tests consisten en probar el programa entero. Para hacer esto, primero se usa Gazebo [3], una herramienta destinada a la simulación de robots en diferentes entornos, y posteriormente se usa el robot real. En estas pruebas se han empleado las demos de MERLIN2 presentadas anteriormente.

3.6. El producto del desarrollo

El producto final desarrollado es MERLIN2, una arquitectura software para robots que permite desarrollar diferentes comportamientos autónomos en robots. MERLIN2 proporciona los medios necesarios para desarrollar nuevas aplicaciones de ROS 2 para crear estos comportamientos. Además, proporciona gestión de navegación, reconocimiento del habla y síntesis de voz. Para usar MERLIN2 basta con desarrollar las acciones que hacen falta, usando la clase Merlin2Action o Merlin2FsmAction; y el nodo encargado de definir el estado inicial del problema y la secuencia de objetivos que el robot tendrá que cumplir. Mediante el YASMIN Viewer se puede visualizar el funcionamiento de las máquinas de estados de MERLIN2, es decir, las acciones de MERLIN2.

Por otro lado, si se utiliza MongoDB como base de conocimientos, se puede utilizar uno de sus visualizadores, como MongoDB Compass, para visualizar el conocimiento actual del robot.

Capítulo 4

Evaluación

En este capítulo se realizan las demostraciones de la validez de la solución elaborada. Primero se describe el proceso de evaluación y después se analizan los resultados obtenidos tras aplicarlo.

4.1. Proceso de evaluación

En esta sección se presenta el proceso de evaluación realizado. Se explica la evaluación realizada sobre MERLIN2.

4.1.1. Forma de evaluación

Para evaluar MERLIN2 se usa la demo 2 explicada en 3.3.5 sobre el entorno simulado con Gazebo [3] presentado en 3.4. Con todo esto se pretenden repetir los experimentos realizados en [15] para poder compararlo. Se han llevado a cabo varias ejecuciones de cada caso de prueba. Para automatizar la ejecución de las pruebas se ha hecho que la demo 2 reciba como parámetros el número de objetivos que se van a realizar y el número de repeticiones. Cuando se terminan de las pruebas, su tiempo de ejecución y la distancia recorrida por el robot se escriben en un archivo CSV. Posteriormente, los archivos CSV se han analizado utilizando JASP [60], que es una herramienta de código abierto para realizar análisis estadísticos.

En primer lugar, la métrica de tiempo mide el tiempo que necesita cada misión. De esta forma, se evaluará el impacto de pasar de una tarea a otra, cancelando o completando la tarea anterior. En segundo lugar, la distancia recorrida permite medir

la distancia necesaria para cumplir una misión. Un robot que navega distancias largas implica un mayor consumo de batería y un tiempo de servicio menos efectivo. Además, significa que el proceso de cambio de una tarea a otra no solo tiene un impacto en el tiempo, sino también en el rendimiento.

También se ha utilizado la demo 1 para probar MERLIN2 sobre robots reales. Para ello, se ha ejecutado esta demo en los robots RB1 y TiaGO en el apartamento simulado del Grupo de Robótica de la Universidad de León. Sin embargo, estas pruebas no se han analizado, solo se ha comprobado su correcto funcionamiento.

En todas las pruebas realizadas se ha utilizado la familia DAO de ROS2, explicada anteriormente en 3.3.2. El motivo de esto es que se quiere replicar el uso de MERLIN, que usaba ROSPlan. ROSPlan empleaba un nodo de ROS para almacenar el conocimiento del robot, igual que esta familia DAO.

4.1.2. Casos de prueba

Los casos de prueba se basan en los experimentos presentados en [15]. De esta manera, se quieren repetir los experimentos en los que se movía al robot a ciertos puntos. Como resultado se tienen los casos de prueba presentados en la tabla 4.1. Como se puede ver, se han definido tres tipos de pruebas, dependiendo del número de objetivos. Estas cantidades son 6, 20 y 120. Además, cada tipo tiene un número diferente de repeticiones. Esto se debe a que cuanto mayor sea el número de objetivos, mayor será su duración.

Figura 4.1: Vista cenital de Gazebo y el mapa desplegados para la validación experimental.

Tabla 4.1: Tabla que resume los casos de prueba para evaluar MERLIN2.

ID	Número de objetivos	Número de repeticiones
1	6	60
2	20	30
3	120	10

Como se ha explicado anteriormente, los objetivos consisten en realizar una tarea en ciertos puntos del mundo simulado. Para generar la lista de puntos aleatorios se utilizan cuatro puntos predefinidos. Estos puntos se pueden ver en la figura 4.1. Los puntos se han marcado con cruces rojas y se corresponden con el dormitorio, el baño, la cocina y la puerta principal del apartamento.

4.2. Análisis de resultados

Después de obtener los resultados de las ejecuciones de los tests definidos anteriormente, se ha usado JASP. De esta forma, de cada test presentado en la tabla 4.1 se han calculado la media, la mediana y la desviación típica. Los resultados obtenidos se presentan en la tabla 4.2.

Tabla 4.2: Tabla que resume los resultados de los tests.

	Tiempo (segundos)			Distancia (metros)		
	1 (6)	2 (20)	3 (120)	1 (6)	2 (20)	3 (120)
Media	103.413	366.275	2171.775	22.658	75.805	451.192
Mediana	104.402	362.521	2153.368	22.938	75.495	459.753
Desviación estándar	13.136	27.718	77.644	3.325	6.036	35.672

Al comprar estos resultados con los obtenidos en [15], se puede ver que los tiempos medios obtenidos son bastante similares o mejores a los obtenidos utilizando MERLIN (1: 104,46±12,30 segundos, 2: 369,14±18,06 segundos, 3: 2154 segundos). Por otro lado, las distancias obtenidas también son similares a las que se habían mostrado en [15] (1: 21,72±3,94 metros, 2: 80,23±6,26 metros, 3: 475 metros). Con esto se tiene que la migración de MERLIN a MERLIN2 no ha producido grandes cambios. Además, según los resultados obtenidos en [15], MERLIN tiene mejor rendimiento que ROSPlan [16], es decir, tiene un tiempo y una distancia recorrida inferiores. De esto se obtiene que MERLIN2 también es mejor que ROSPlan.

Capítulo 5

Conclusión

En este capítulo se presenta la conclusión del trabajo. Está formado por los resultados del proyecto, las aportaciones realizadas y los trabajos futuros. En este trabajo se ha presentado una arquitectura cognitiva híbrida llamada MERLIN2. Esta arquitectura permite modelar el comportamiento de un robot para realizar tareas específicas. A continuación se presentan las aportaciones y los trabajos futuros.

5.1. Aportaciones realizadas

En esta sección se presentan las aportaciones realizadas en este trabajo.

5.1.1. MERLIN2

La principal aportación de este proyecto es el diseño y desarrollo de la arquitectura **MERLIN2**. Esta arquitectura se puede emplear para desarrollar nuevas aplicaciones software que modelen nuevos comportamientos en los robots. MERLIN2 se ha implementado en Python3 siendo su núcleo dos librerías creadas para gestionar el conocimiento del robot, KANT (Knowledge mAnagemeNT); y para desarrollar comportamientos basados en máquinas de estados, YASMIN (Yet Another State MachINe).

Aportación 1 *Arquitectura cognitiva para robots de servicios* ***MERLIN2****.*

Aportación 2 *Diseño de una arquitectura cognitiva para robots de servicios, MERLIN2, desde el punto de vista de la ingeniería del software.*

MERLIN2 es una arquitectura híbrida que es el resultado de migrar la arquitectura MERLIN [15]. Para poder realizar la migración, se ha desarrollado un sistema

deliberativo que sustituya a ROSPlan [16]. El núcleo de este sistema es un sistema de planificación creado para generar planes que satisfagan los objetivos del robot. Además, se ha comparado MERLIN2 y MERLIN, repitiendo el experimento presentado en [15]. Como se ha mostrado en el capítulo 4, MERLIN2 es bastante similar a MERLIN. De esta forma, en este trabajo se puede ver cómo ha evolucionado la arquitectura MERLIN a MERLIN2, actualizando y mejorando sus componentes.

Aportación 3 *Sistema de planificación basado en PDDL para ROS 2.*

Aportación 4 *Comparación de las arquitecturas MERLIN2 y MERLIN.*

Aportación 5 *Evolución de la arquitectura cognitiva MERLIN.*

Por otro lado, este proyecto ha producido aportaciones secundarias. Se ha presentado un resumen de las arquitecturas software para robots, desde las arquitecturas precursoras hasta arquitecturas más modernas. MERLIN2 se ha desarrollado teniendo en cuenta diversos conceptos estudiados en el estado de la cuestión y en su versión anterior, MERLIN. Así, se usan paradigmas como las arquitecturas de tres capas, las arquitecturas híbridas, el uso de planificadores, la gestión del conocimiento y las máquinas de estados finitas.

Aportación 6 *Estado del arte de las arquitecturas software para robots.*

Por último, se han migrado varios elementos de MERLIN. Estos componentes son la navegación topológica, la síntesis de voz y el reconocimiento del habla. Además, se han desarrollado mediante el uso de varios patrones de diseño software.

Aportación 7 *Sistema de navegación basado en puntos con ID y coordenadas para ROS 2.*

Aportación 8 *Sistema de síntesis de voz para ROS 2.*

Aportación 9 *Sistema de reconocimiento del habla para ROS 2.*

5.1.2. Gestión del conocimiento

MERLIN2 utiliza KANT para gestionar el conocimiento del robot. KANT produce cierta abstracción de la base de conocimientos y del lenguaje PDDL gracias al uso de varios patrones de diseño software. Con esto se consigue una mayor flexibilidad a la hora de crear nuevos componentes que tengan que trabajar con el conocimiento del robot.

Aportación 10 ***KANT****, librería Python, integrada en ROS 2, para gestionar el conocimiento PDDL.*

Aportación 11 *Se han modelado elementos PDDL utilizando el patrón de Data Transfer Object (DTO). Los elementos PDDL modelados son type, object, predicate, proposition y actión.*

Aportación 12 *Se ha implementado el patrón Data Access Object (DAO) para abstraerse del tipo de almacenamiento de la base de conocimientos. Gracias a esto, se puede acceder al PDDL desde el código fuente con unos métodos comunes.*

Aportación 13 *Se han utilizado los patrones Abstract Factory y Factory Method para gestionar la creación de elementos DAO. Gracias a estos patrones los elementos DAO se pueden clasificar en familias según el tipo de almacenamiento.*

5.1.3. Máquinas de estados

Se ha diseñado y creado YASMIN, una librería de Python para reemplazar a SMACH [17]. Gracias a YASMIN se pueden desarrollar comportamientos mediante la creación de máquinas de estados. Además, se puede monitorizar el funcionamiento de las diferentes máquinas de estados que se estén ejecutando gracias a su visualizador.

Aportación 14 ***YASMIN****, librería Python, integrada en ROS 2, para desarrollar comportamientos mediante máquinas de estados.*

Aportación 15 *Visualizador de máquinas de estados.*

5.2. Trabajos futuros

El principal objetivo de este trabajo es la creación de la arquitectura cognitiva híbrida MERLIN2, que es la nueva versión de MERLIN. Además, se han desarrollado las librerías KANT y YASMIN. De esta forma, los trabajos futuros propuestos se centran en la mejora de los componentes de MERLIN2, KANT y YASMIN.

En cuanto a KANT, PDDL es un lenguaje con muchos elementos. Sin embargo, no todos ellos tienen la misma importancia. Principalmente, su importancia depende de la frecuencia de uso y del soporte de los planificadores actuales. Los elementos PDDL presentados en KANT son los más empleados, pero se pueden integrar más elementos PDDL, como los Numeric Fluents. También se propone desarrollar nuevas familias DAO para diferentes tipos de fuentes de datos y comprobar cuál es la mejor en diferentes

escenarios. Por otro lado, se podría desarrollar una nueva aplicación para visualizar el conocimiento del robot.

Con el desarrollo de YASMIN se buscaba crear una versión simplificada de SMACH. Sin embargo, se podrían añadir nuevos estados a YASMIN. Ya que hay estados para usar servicios y acciones de ROS 2, se podría desarrollar un estado para usar los topics. Otro posible trabajo futuro es la comparación de YASMIN y SMACH. Además, se puede crear una aplicación de escritorio para visualizar las máquinas de estados de YASMIN, igual que el visualizador que había en SMACH.

MERLIN2 es una arquitectura que puede seguir creciendo. Por este motivo se propone añadir otros planificadores a la Planning Layer, como por ejemplo SMTPlan+ [61]. Con esto se podría aplicar el Strategy Pattern al uso de diferentes planificadores, siendo cada planificador una estrategia diferente. Por otro lado, se podrían desarrollar nuevos sistemas reactivos, como por ejemplo un sistema para reconocer objetos para buscar objetos en el entorno. También se podría añadir un sistema de manipulación para que el robot pudiera interactuar con el entorno.

Por último, se propone migrar MERLIN2, KANT y YASMIN a C++, el otro lenguaje principal de ROS 2. Además, se podría comparar el uso de MERLIN2 en Python y en C++ para determinar si compensa desarrollar código en C++.

Bibliografía

[1] Oscar Lima, Rodrigo Ventura, and Iman Awaad. Integrating classical planning and real robots in industrial and service robotics domains. In *Proc. of PlanRob 2018 - 6th Workshop on Planning and Robotics, held at ICAPS 2018, Delft, The Netherlands*. 2018.

[2] Vs code counter. `https://marketplace.visualstudio.com/items?itemName=uctakeoff.vscode-counter`, 20/02/2021.

[3] Nathan Koenig and Andrew Howard. Design and use paradigms for gazebo, an open-source multi-robot simulator. In *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)(IEEE Cat. No. 04CH37566)*, volume 3, pages 2149–2154. IEEE.

[4] GitLab. `https://about.gitlab.com/`, 16/11/2020.

[5] pyjsgf. `https://pypi.org/project/pyjsgf/`, 12/01/2021.

[6] S. Bradshaw, K. Chodorow, and E. Brazil. *MongoDB: the Definitive Guide: Powerful and Scalable Data Storage*. The expert's voice in open source. O'Reilly Media, Incorporated, 2019.

[7] mongoengine. `http://mongoengine.org/#home`, 16/11/2020.

[8] pytest. `https://docs.pytest.org/en/stable/`, 20/02/2021.

[9] Maria Fox and Derek Long. Pddl2.1: An extension to pddl for expressing temporal planning domains. *J. Artif. Intell. Res. (JAIR)*, 20:61–124, 12 2003.

[10] Pylint. `https://www.pylint.org/`, 20/02/2021.

[11] Python. `https://es.wikipedia.org/wiki/Python`, 16/11/2020.

[12] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y Ng. Ros: an open-source robot operating system. 3:5, 2009.

[13] unittest. `https://docs.python.org/3/library/unittest.html`, 20/02/2021.

[14] YAML. `https://yaml.org/`, 20-12-2020.

[15] Miguel Á. González-Santamarta, Francisco J. Rodríguez-Lera, Claudia Álvarez-Aparicio, Ángel M. Guerrero-Higueras, and Camino Fernández-Llamas. MERLIN a cognitive architecture for service robots. *Applied Sciences*, 10(17):5989, aug 2020.

[16] Michael Cashmore, Maria Fox, Derek Long, Daniele Magazzeni, Bram Ridder, Arnau Carrera, N. Palomeras, N. Hurtós, and Marc Carreras. ROSplan: Planning in the robot operating system. In *Proceedings International Conference on Automated Planning and Scheduling, ICAPS*, volume 2015, pages 333–341, 01 2015.

[17] Jonathan Bohren and Steve Cousins. The smach high-level executive. *IEEE Robotics & Automation Magazine*, 17(4):18 – 20, 2011.

[18] Scrumban. `https://ora.pm/es/blog/scrum-vs-kanban-vs-scrumban`, 16/11/2020.

[19] G. Pardo-Castellote. Omg data-distribution service: architectural overview. In *23rd International Conference on Distributed Computing Systems Workshops, 2003. Proceedings.*, pages 200–206, 2003.

[20] Ronald Arkin and Tucker Balch. Aura: Principles and practice in review. *Journal of Experimental & Theoretical Artificial Intelligence*, 9, 02 1970.

[21] Erann Gat, R Peter Bonnasso, Robin Murphy, et al. On three-layer architectures. *Artificial intelligence and mobile robots*, 195:210, 1998.

[22] Erann Gat. Integrating planning and reacting in a heterogeneous asynchronous architecture for controlling real-world mobile robots. pages 809–815, 01 1992.

[23] Julio Rosenblatt. Damn: A distributed architecture for mobile navigation. *J. Exp. Theor. Artif. Intell.*, 9:339–360, 04 1997.

[24] L.E. Parker. Alliance: An architecture for fault tolerant multi-robot cooperation. 2 1995.

[25] David Kortenkamp and Reid Simmons. *Robotic Systems Architectures and Programming*, pages 187–206. 01 2008.

[26] Carlos Agüero, José Plaza, Francisco Martín, and Eduardo Perdices. Behavior-based iterative component architecture for soccer applications with the nao humanoid. 12 2012.

[27] Francisco Rodríguez Lera, Vicente Matellán, Miguel Conde-González, and Francisco Martín. Himop: A three-component architecture to create more human-acceptable social-assistive robots: Motivational architecture for assistive robots. *Cognitive Processing*, 19, 01 2018.

[28] Francisco Martín, Francisco Rodríguez Lera, Jonatan Ginés Clavero, and Vicente Matellán. Evolution of a cognitive architecture for social robots: Integrating behaviors and symbolic knowledge. *Applied Sciences*, 10:6067, 09 2020.

[29] Guillaume Sarthou, Amandine Mayima, Guilhem Buisan, Kathleen Belhassein, and Aurélie Clodic. The director task: a psychology-inspired task to assess cognitive and interactive robot architectures. In *2021 30th IEEE International Conference on Robot & Human Interactive Communication (RO-MAN)*, pages 770–777. IEEE, 2021.

[30] Paula Rubio-Fernández. The director task: A test of theory-of-mind use or selective attention? *Psychonomic bulletin & review*, 24(4):1121–1128, 2017.

[31] Robotnik. `http://www.robotnik.es/robotnik-rb-manipulador/`, 20/02/2021.

[32] Hokuyo. `https://www.hokuyo-aut.jp/search/single.php?serial=166`, 20/02/2021.

[33] Kinova. `https://www.kinovarobotics.com/en/products/assistive-technologies/kinova-jaco-assistive-robotic-arm`, 20/02/2021.

[34] PAL Robotics. `http://pal-robotics.com/es/`, 20/02/2021.

[35] JETSON TX2. `https://www.nvidia.com/es-es/autonomous-machines/embedded-systems/jetson-tx2/`, 20/02/2021.

[36] Ubuntu. `http://releases.ubuntu.com/20.04/`, 16/11/2020.

[37] Bases y tipos de cotización 2019. `http://www.seg-social.es/wps/portal/wss/internet/Trabajadores/CotizacionRecaudacionTrabajadores/36537?changeLanguage=es`, 20/02/2021.

[38] Presupuestos de la Universidad de León. `https://www.unileon.es/ficheros/estructura/gerencia/presupuesto2021.pdf`, 20-02-2021.

[39] BOE. `https://www.boe.es/buscar/pdf/2017/BOE-A-2017-12902-consolidado.pdf`, 20-02-2021.

[40] colcon. `https://colcon.readthedocs.io/en/released/`, 16/11/2020.

[41] Amanda Coles, Andrew Coles, Maria Fox, and Derek Long. Forward-chaining partial-order planning. pages 42–49, 01 2010.

[42] ArangoDb. `https://www.arangodb.com/`, 13/01/2021.

[43] Neo4J. `https://neo4j.com/`, 13/01/2021.

[44] Mongo Atlas. `https://www.mongodb.com/cloud/atlas`, 16/11/2020.

[45] Javascript. `https://developer.mozilla.org/es/docs/Web/JavaScript`, 20/02/2021.

[46] React.js. `https://es.reactjs.org/`, 20/02/2021.

[47] Cytoscape.js. `https://js.cytoscape.org/`, 20/02/2021.

[48] Flask. `https://flask.palletsprojects.com/en/2.0.x/`, 20/02/2021.

[49] espeak. `http://espeak.sourceforge.net/`, 12/01/2021.

[50] Speech dispatcher. `https://freebsoft.org/speechd`, 12/01/2021.

[51] Festival. `https://www.cstr.ed.ac.uk/projects/festival/`, 12/01/2021.

[52] gtts. `https://pypi.org/project/gTTS/`, 12/01/2021.

[53] Moon modeler. `https://www.datensen.com/data-modeling/moon-modeler-for-databases.html`, 20/02/2021.

[54] MongoDB Compass. `https://www.mongodb.com/products/compass`, 16/11/2021.

[55] Steve Macenski, Francisco Martín, Ruffin White, and Jonatan Ginés Clavero. The marathon 2: A navigation system. In *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2020.

[56] mpg321. `http://mpg321.sourceforge.net/`, 12/01/2021.

[57] Speechrecognition. `https://pypi.org/project/SpeechRecognition/`, 12/01/2021.

[58] nltk. `https://pythonspot.com/category/nltk/`, 12/01/2021.

[59] URDF. `http://wiki.ros.org/urdf`, 07/01/2021.

[60] JASP Team. JASP (Version 0.16)[Computer software], 2021.

[61] Michael Cashmore, Maria Fox, Derek Long, and Daniele Magazzeni. A compilation of the full pddl+ language into smt. In *Workshops at the Thirtieth AAAI Conference on Artificial Intelligence*, 2016.

[62] Google Meet. `https://meet.google.com/`, 15/01/2021.

[63] Discord. `https://discord.com/`, 15/01/2021.

[64] pyreverse. `https://www.logilab.org/blogentry/6883`, 13/01/2021.

[65] Draw.io. `https://www.draw.io/`, 13/01/2021.

Anexo A

Control de versiones

Para gestionar y controlar el código desarrollado se ha usado GitLab [4] como servicio de control de versiones. El GitLab empleado es un servicio privado del Grupo de Robótica de la Universidad de León. En este servicio cada usuario puede crear varios repositorios personales.

Como MERLIN2 es un proyecto modular en el que se han usado componentes independientes, se han usado varios repositorios. De esta forma, se han usado los siguientes repositorios:

- Repositorio para el paquete simple_node: este repositorio contiene el paquete de ROS 2 simple_node.
- Repositorio para KANT: este repositorio contiene el código de la librería KANT organizado en paquetes de ROS 2. También se incluyen las inrtefaces de ROS 2 creadas.
- Repositorio para YASMIN: este repositorio contiene el código de la librería YASMIN organizado en paquetes de ROS 2. También se incluyen las inrtefaces de ROS 2 creadas.
- Repositorio para el sistema de navegación: este repositorio contiene el código del sistema de navegación organizado en paquetes de ROS 2. También se incluyen las interfaces de ROS 2 creadas.
- Repositorio para el sistema de síntesis de voz: este repositorio contiene el código del sistema de síntesis de voz organizado en paquetes de ROS 2. También se incluyen las interfaces de ROS 2 creadas.

- Repositorio para el sistema de reconocimiento del habla: este repositorio contiene el código del sistema de reconocimiento del habla organizado en paquetes de ROS 2. También se incluyen las interfaces de ROS 2 creadas.
- Repositorio para MERLIN2: este repositorio contiene el código de MERLIN2 organizado en paquetes de ROS 2. También se incluyen las interfaces de ROS 2 creadas. Además, se han usado los submodules de Git para hacer que los repositorios anteriormente presentados sean subdirectorios de este repositorio. Esto permite mantener los commits de los diferentes elementos separados y clonar todos ellos de una sola vez.
- Repositorio para el simualdor de RB1: este repositorio contiene la configuración para simular el robot RB1 en ROS 2. Incluye varios paquetes de ROS 2 para preparar sus controladores simulados, para gestionar sus modelos y para lanzar la navegación junto al simulador.

Anexo B

Seguimiento de proyecto fin de carrera

B.1. Forma de seguimiento

Como se ha explicado en la introducción, la metodología empleada en este proyecto es Scrumban. Esta metodología incluye la realización de reuniones diarias y reuniones al final de cada iteración. De esta forma, a lo largo del proyecto se han realizado reuniones para realizar el seguimiento del trabajo.

Las reuniones que se realizan al final de cada iteración se han realizado en la sala de reuniones del Grupo de Robótica de la Universidad de León. Esta sala está situada en el edificio del Módulo de Investigación Cibernética, MIC. Por otro lado, las reuniones diarias se han llevado a cabo en el laboratorio del Grupo de Robótica también situado en el MIC. En algunos casos no ha sido posible asistir físicamente a las reuniones. Por este motivo, se han utilizado herramientas de apoyo como Google Meet [62] y Discord [63].

B.2. Planificación inicial

La planificación inicial de este proyecto se ha presentado en el apartado 2.2. La planificación de las tareas del proyecto se ha presentado en la tabla 2.1, mostrando las fechas de inicio y fin, la duración y las dependencias de cada tarea. Además, la figura 2.1 presenta el diagrama Gantt.

B.3. Planificación final

El desarrollo del proyecto se ha subdividido en tareas más pequeñas que se podrían controlar de forma más eficaz mediante el tablero de Scrumban. En la sección 2.2.4 se presentan las iteraciones realizadas. Como se puede ver, se han dado varios adelantos y atrasos con respecto a la planificación inicial. Además, como se puede ver en la tabla 2.2, han surgido nuevas tareas y subtareas durante el desarrollo del proyecto. Sin embargo, se puede ver que el proyecto se ha desarrollado dentro de los límites establecidos inicialmente. El tiempo restante desde la finalización del proyecto hasta la presentación se ha empleado en mejorar la memoria, especialmente el estado de la cuestión.

Anexo C

Diagramas de clases

En este anexo se presentan los diagramas de clases que se han desarrollado en el proyecto. Se han creado usando pyreverse [64] y Draw.io [65].

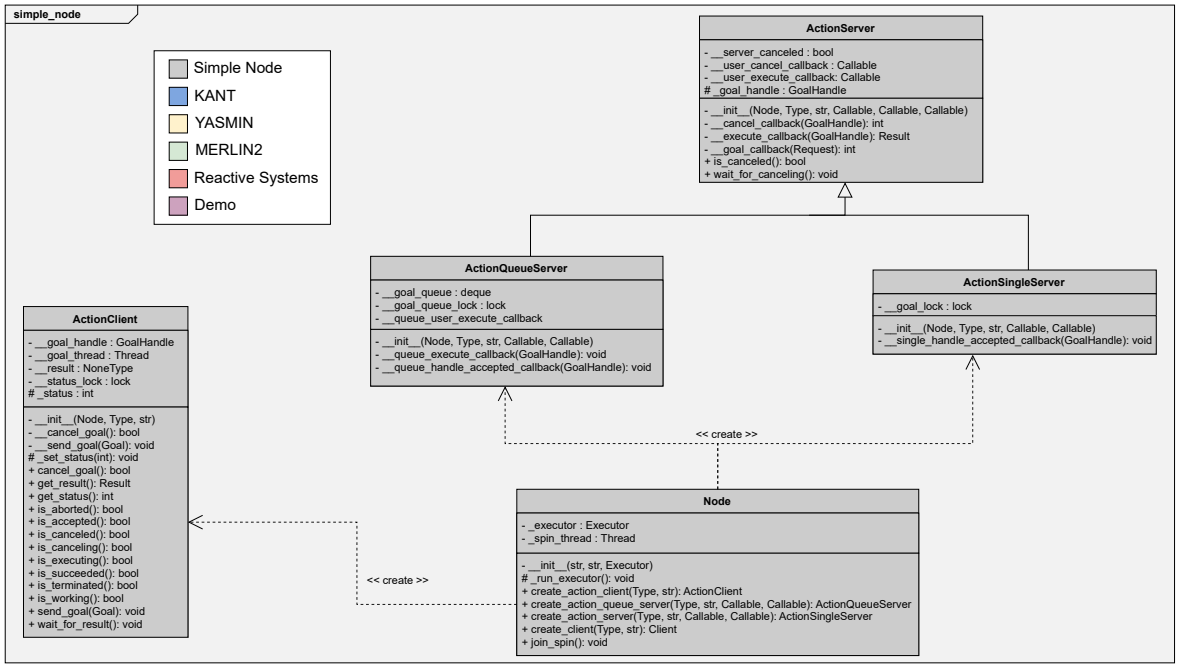


Figura C.1: Diagrama de clases del paquete de ROS 2 simple_node.

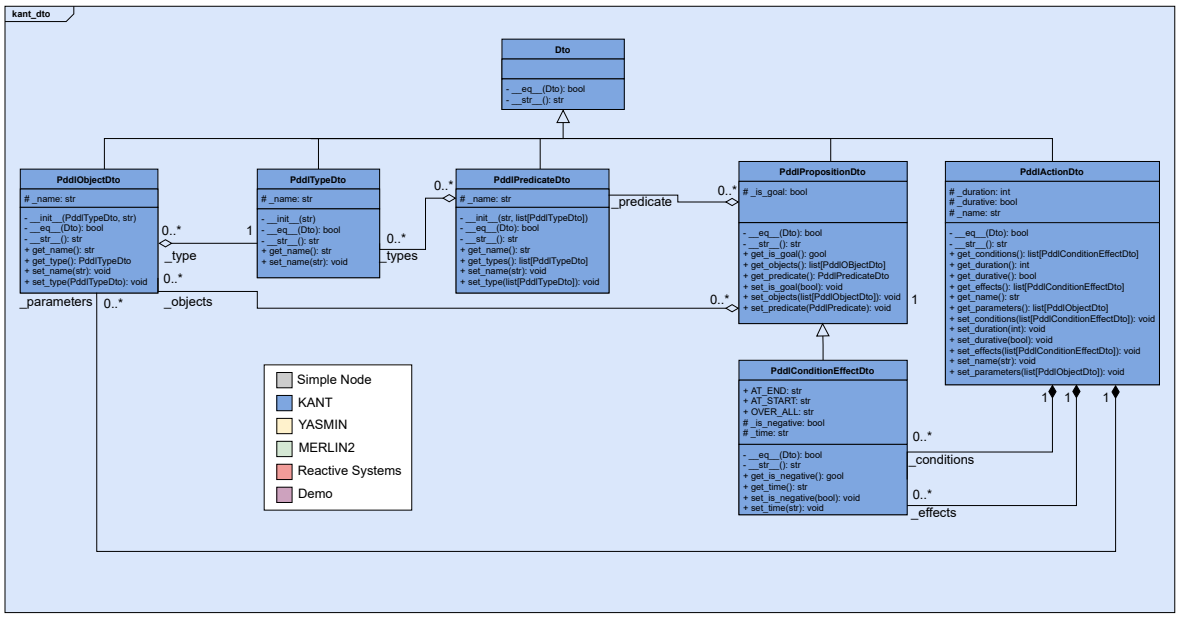


Figura C.2: Diagrama de clases del paquete de ROS 2 kant_dto.

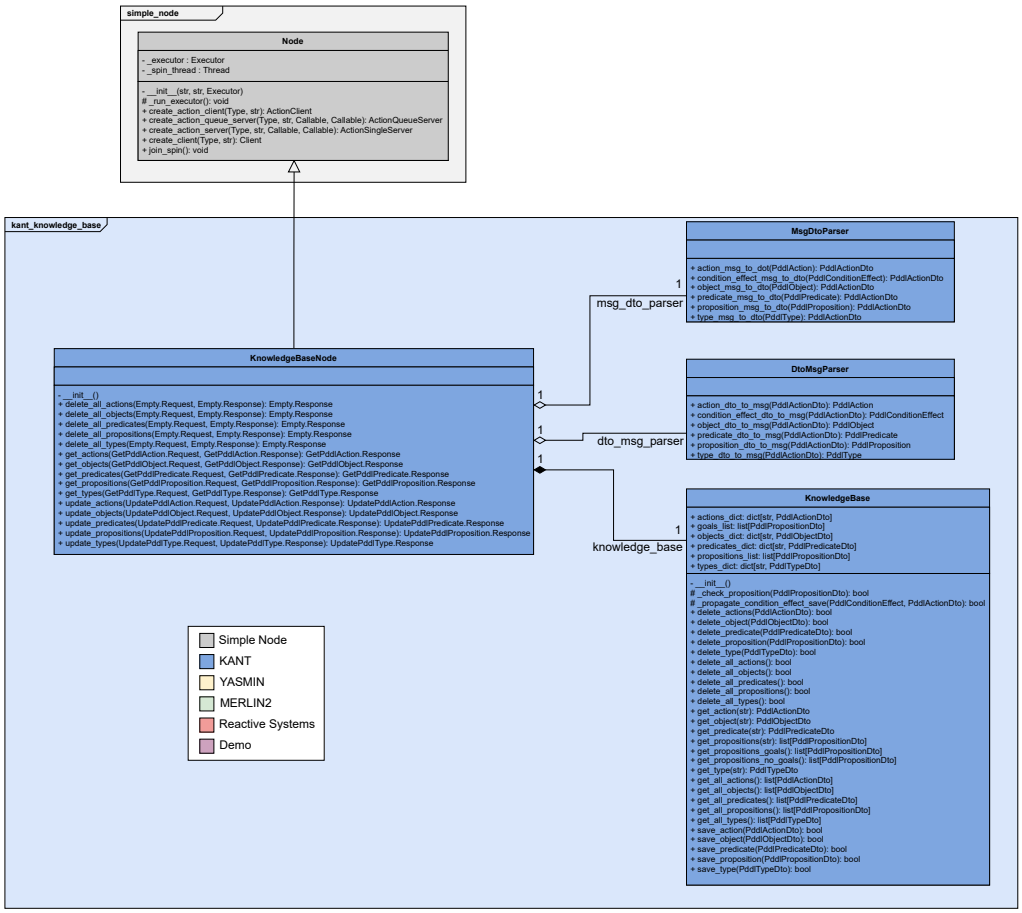


Figura C.3: Diagrama de clases del paquete de ROS 2 kant_knowledge_base.

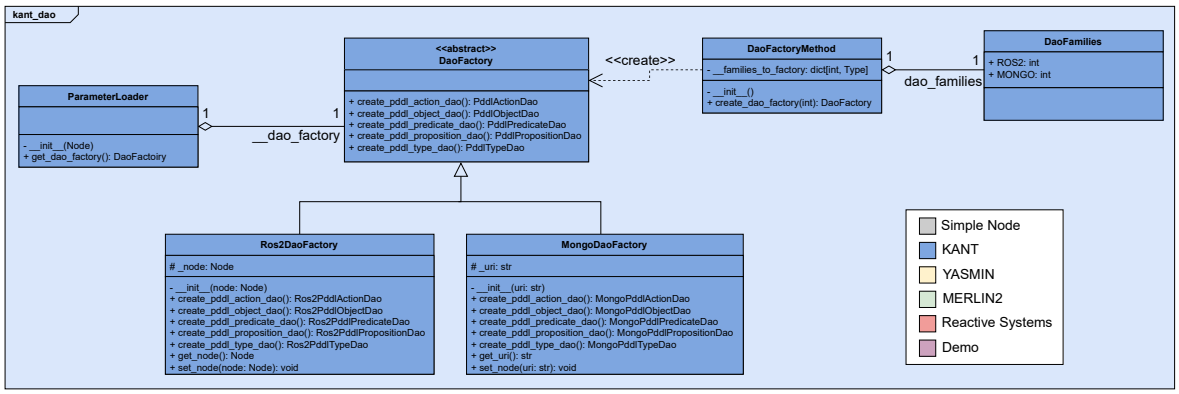


Figura C.4: Diagrama de clases del paquete de ROS 2 dao_factory de KANT.

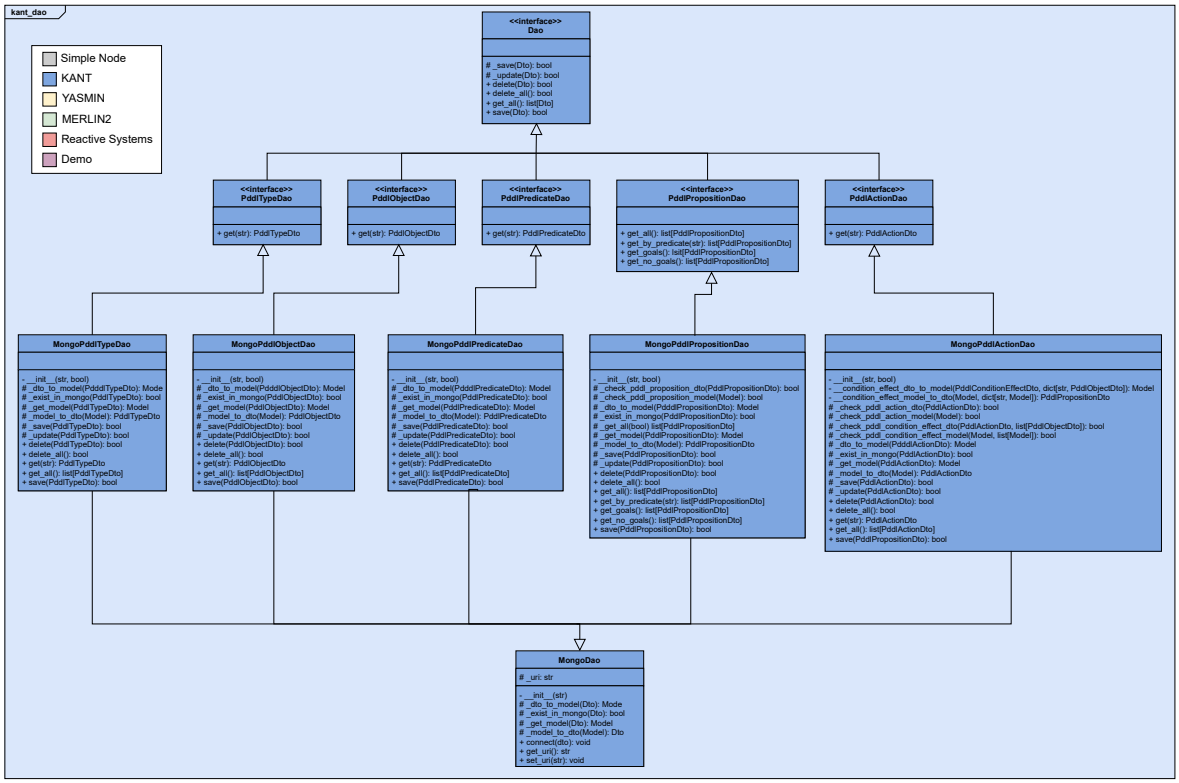


Figura C.5: Diagrama de clases del paquete de ROS 2 mongo_dao de KANT.

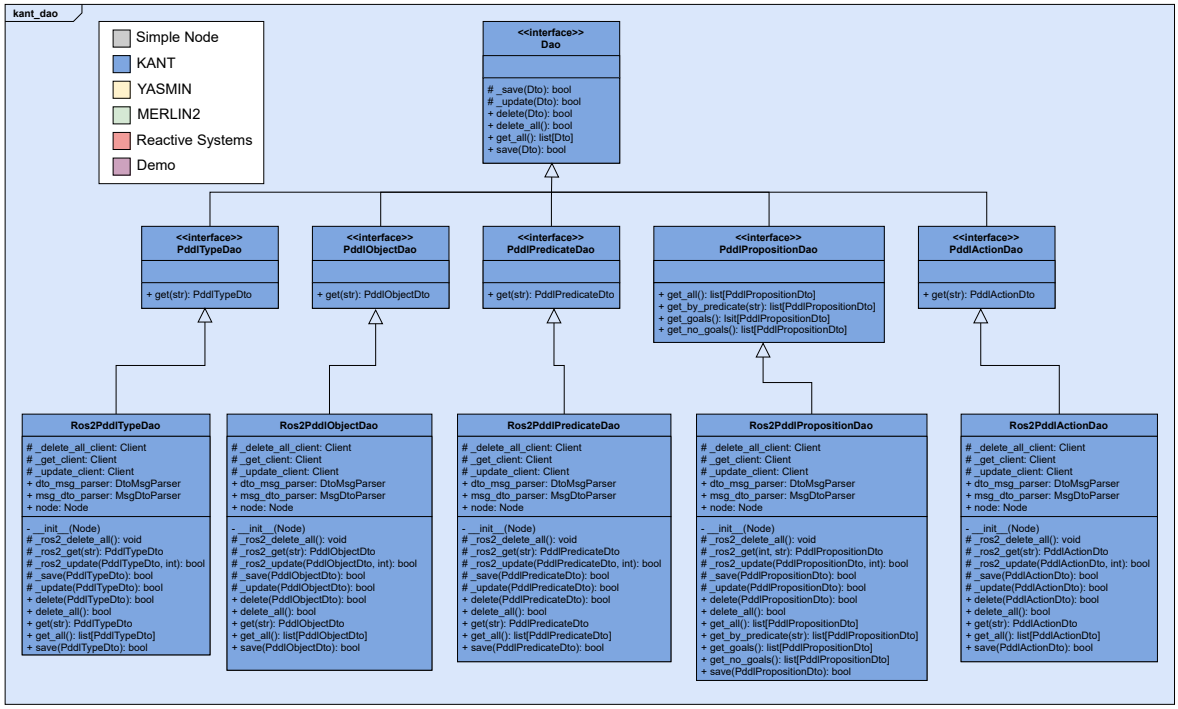


Figura C.6: Diagrama de clases del paquete de ROS 2 ros2_dao de KANT.

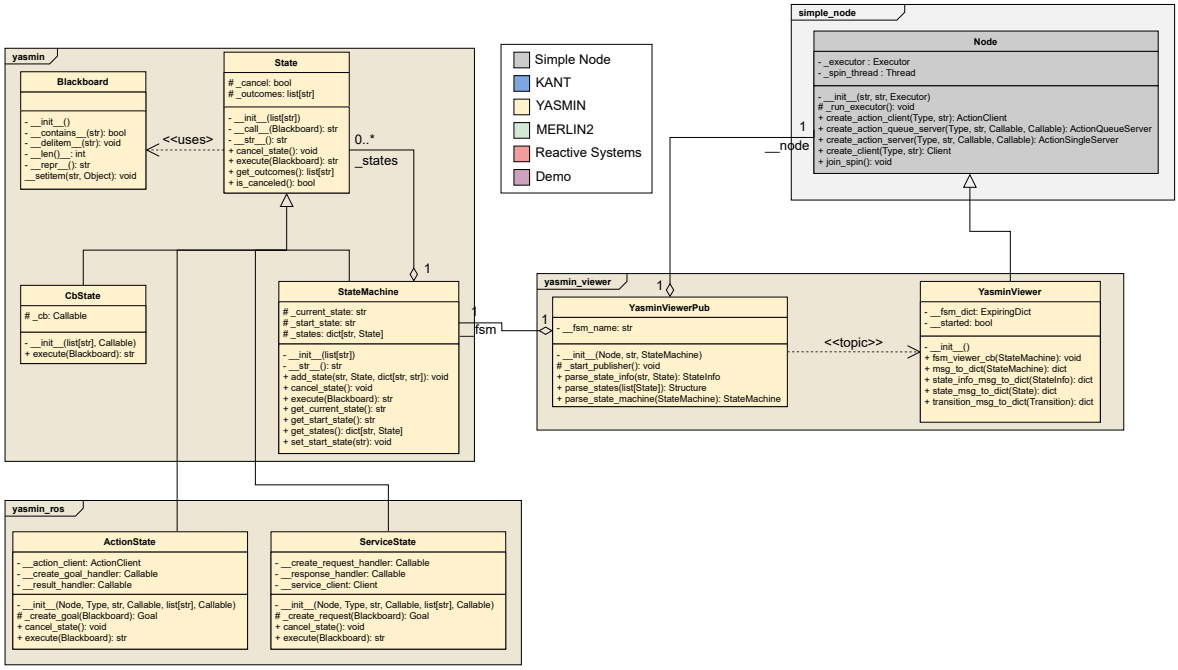


Figura C.7: Diagrama de clases de YASMIN.

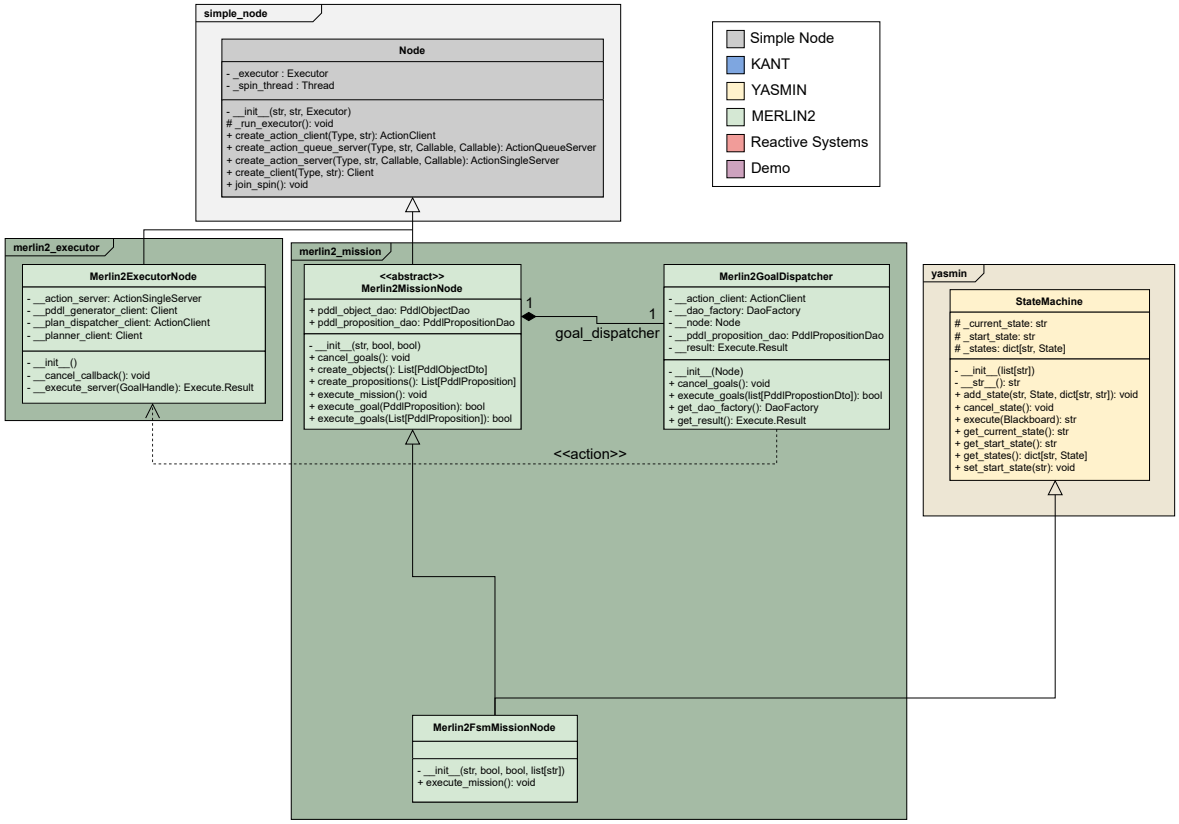


Figura C.8: Diagrama de clases de la Mission Layer de MERLIN2.

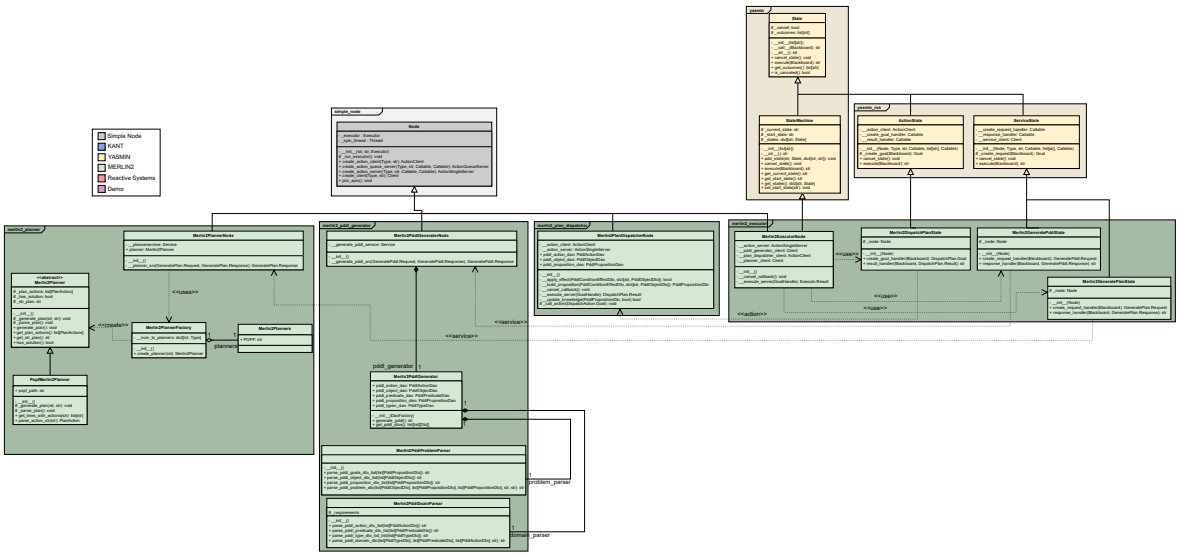


Figura C.9: Diagrama de clases de la Planning Layer de MERLIN2.

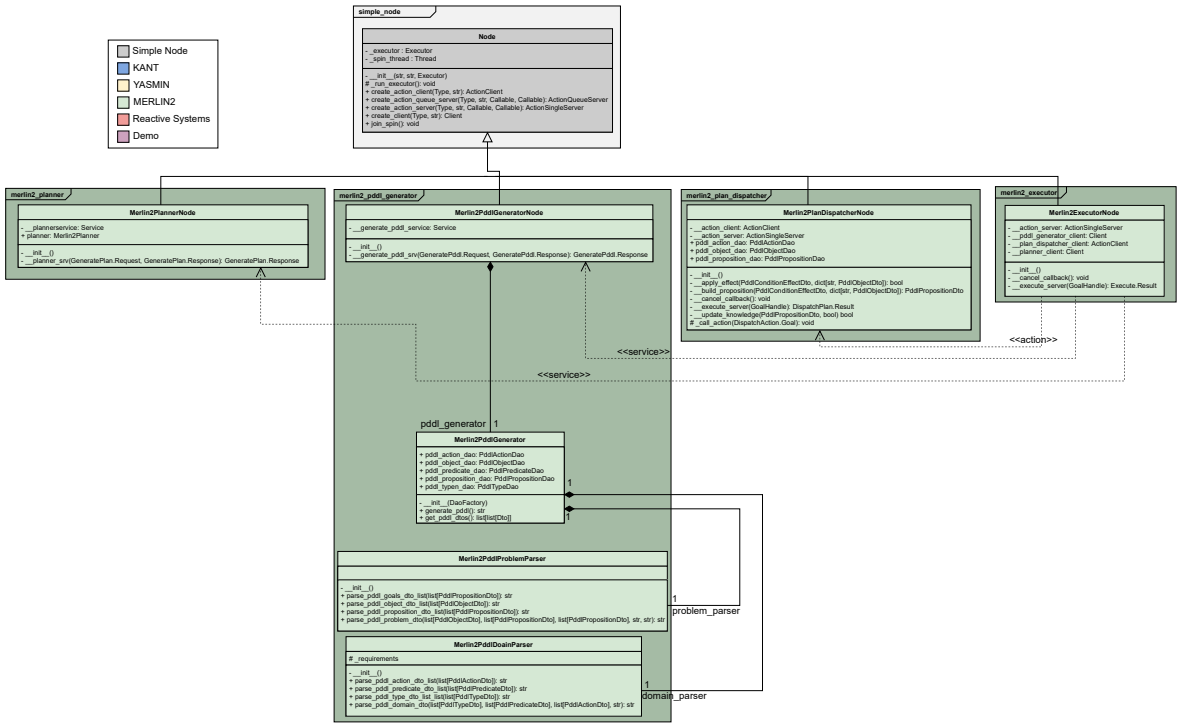


Figura C.10: Diagrama de clases de la Planning Layer de MERLIN2 (paquete merlin2_pddl_generator).

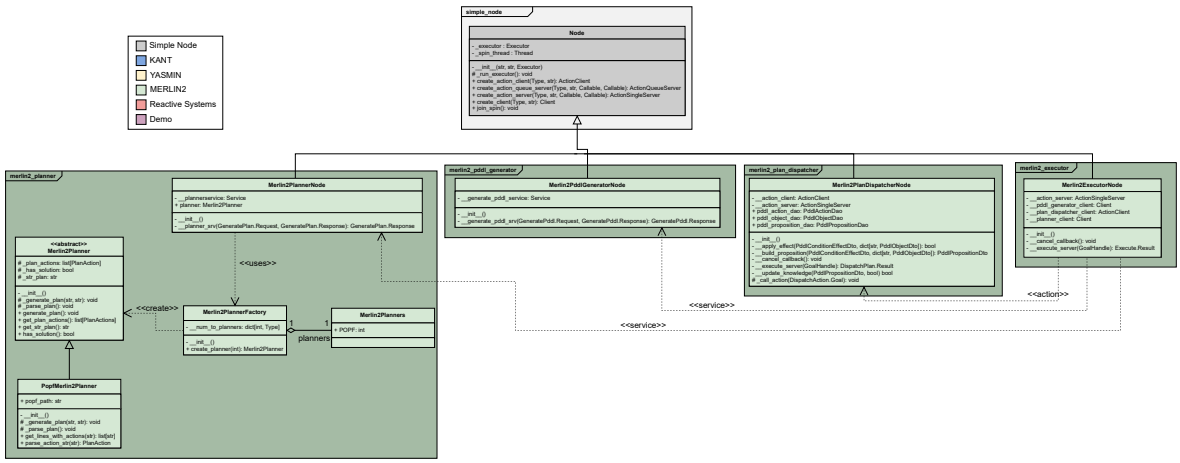


Figura C.11: Diagrama de clases de la Planning Layer de MERLIN2 (paquete merlin2_planner).

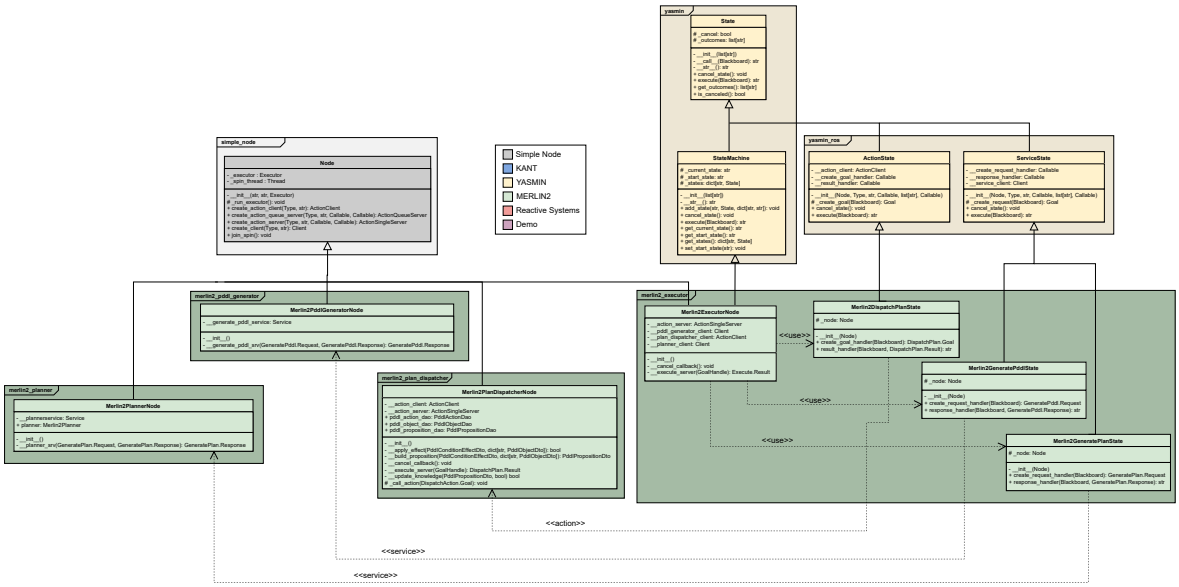


Figura C.12: Diagrama de clases de la Planning Layer de MERLIN2 (paquete merlin2_executor).

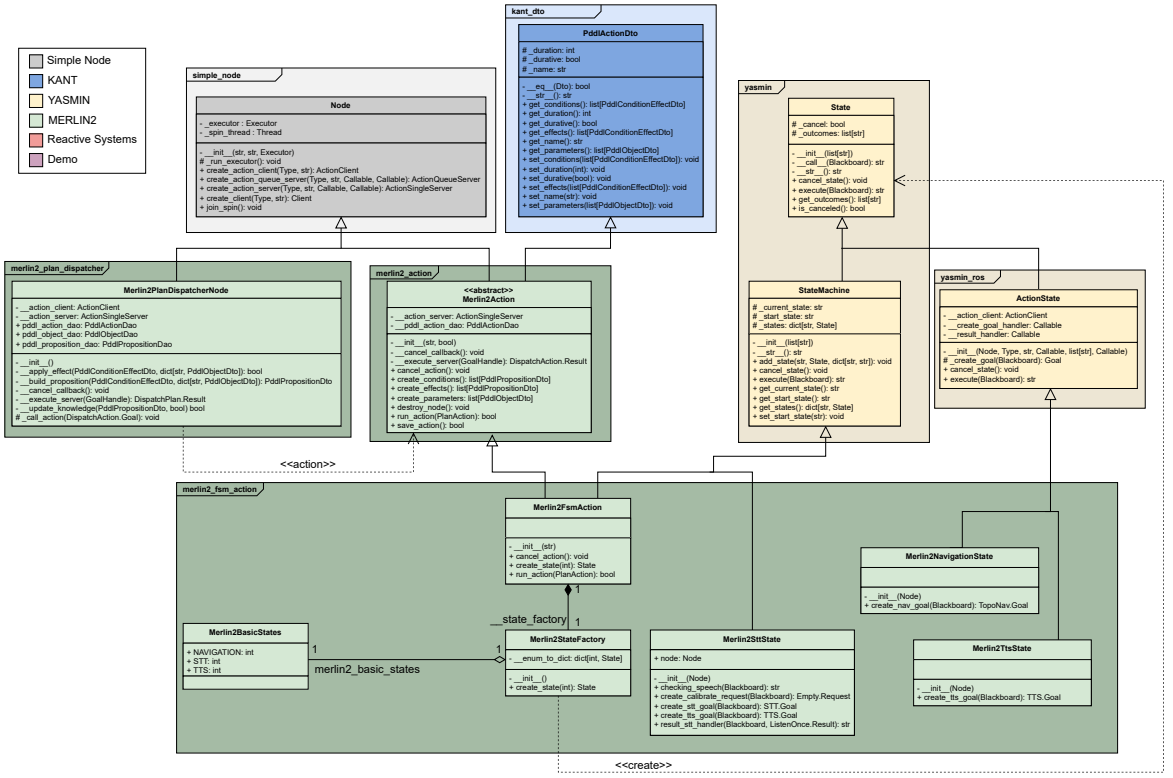


Figura C.13: Diagrama de clases de la Executive Layer de MERLIN2.

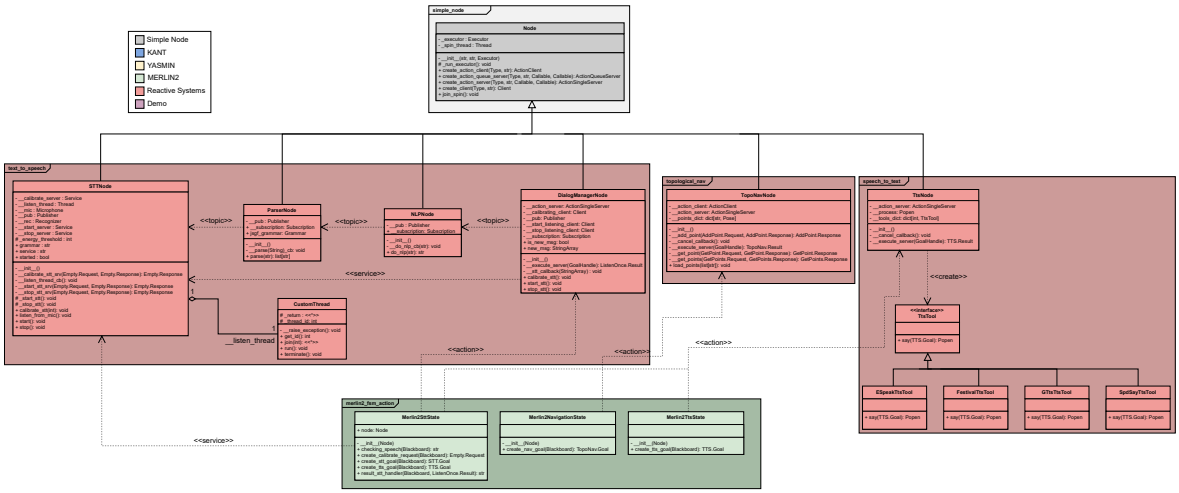


Figura C.14: Diagrama de clases de la Reactive Layer de MERLIN2.

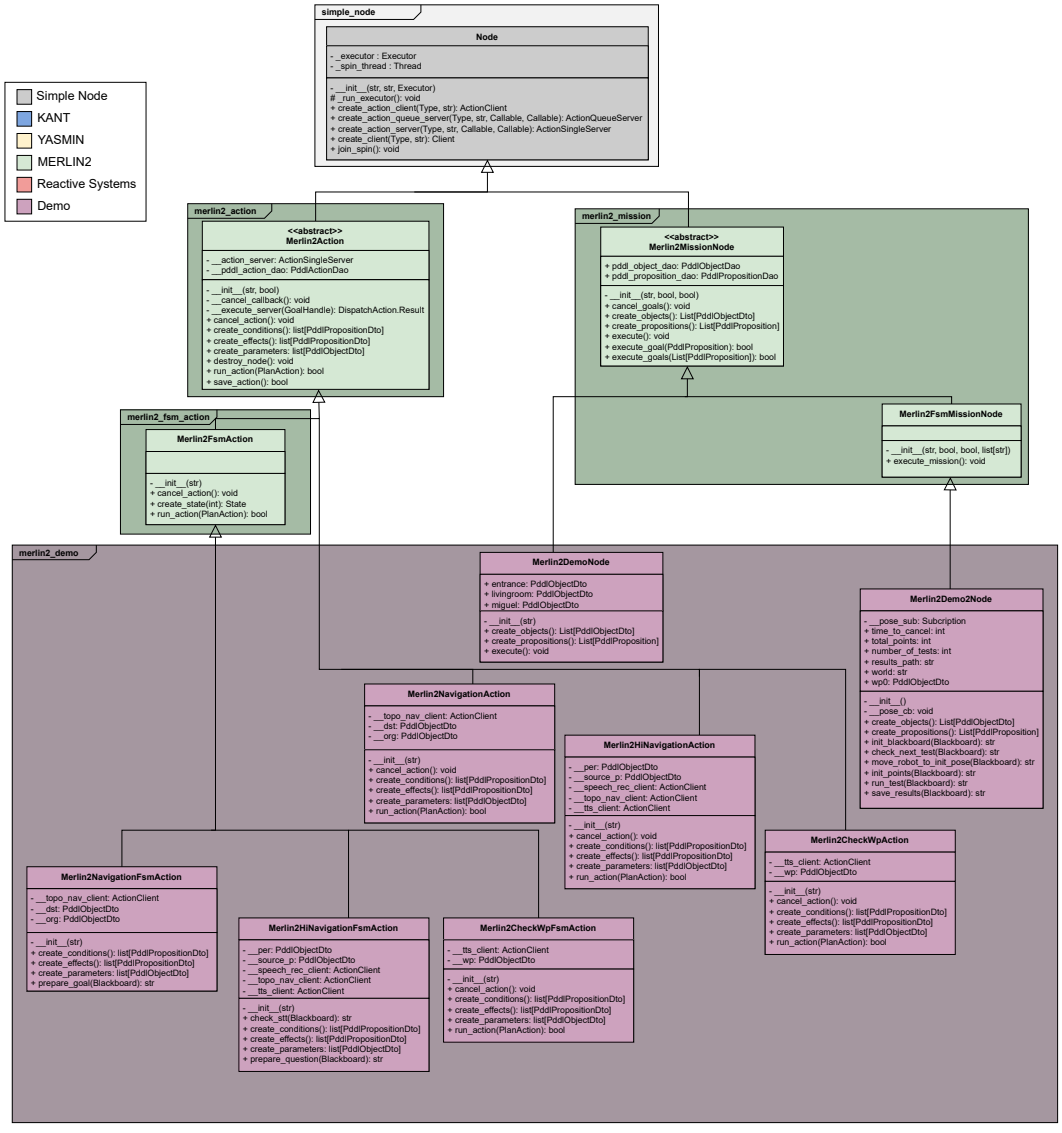


Figura C.15: Diagrama de clases de la demo de MERLIN2.