ESCUELA POLITÉCNICA

UNIVERSIDAD DE EXTREMADURA

ESCUELA POLITÉCNICA
INGENIERÍA INFORMÁTICA EN INGENIERÍA DE COMPUTADORES

TRABAJO FIN DE GRADO

Localización de robots en tiempo real y simulación de comportamientos biológicos

UNIVERSIDAD DE EXTREMADURA

ESCUELA POLITÉCNICA
INGENIERÍA INFORMÁTICA EN INGENIERÍA DE COMPUTADORES

TRABAJO FIN DE GRADO

Localización de robots en tiempo real y simulación de comportamientos biológicos

Autor: Ismael Sánchez García
Tutor: Pablo Bustos García de Castro

Resumen

El objetivo principal de este proyecto es conseguir crear cinco robots[2], los cuales simulen cuatro de los comportamientos básicos de los corderos. Este estudio es parte de un proyecto en el que actualmente está trabajando Robolab, grupo de investigación de robótica[1] de la Universidad de Extremadura.

En este documento se explicará cómo se ha realizado el proyecto paso a paso y como este se divide en tres partes principales.

La primera de ellas son los capítulos dos, tres y cuatro. La segunda, la cual está formada solo por el capítulo cinco y la tercera y última que la forman desde el capítulo seis al ocho.

La primera parte se basa en todos los elementos que se han utilizado durante el proyecto tanto hardware como software, y algunas alternativas que han sido estudiadas pero finalmente no han sido utilizadas.

En la segunda parte se explicará todo el desarrollo del proyecto, desde el problema que se quiere solucionar inicialmente hasta las últimas funciones implementadas en el proyecto.

También se verán las diferentes variantes que se han utilizado en la resolución los problemas planteados, y cual ha sido la solución final.

Finalmente, en la última parte se explicarán las conclusiones y opciones futuras que se

podrían añadir al proyecto y así ampliarlo y mejorarlo. También se incluirá un glosario en el que se explicarán los términos más utilizados en el proyecto, para que sea más sencillo su comprensión.

Índice general

Índice de figuras

Capítulo 1

Introducción

En las ultimas décadas, venimos presenciando como evoluciona internet y las nuevas tecnologías y como estas se van integrando rápidamente en la mayoría de los aspectos de la vida. Durante el desarrollo de este trabajo, se verá como la tecnología se pueden integrar en parte del sector primario. Este trabajo forma parte de un proyecto llamado 'Caracterización etológica de corderos con tecnologías robóticas', en el cual está trabajando actualmente el grupo de investigación de la Universidad de Extremadura, Robolab.

Este proyecto trata de intentar minimizar el coste de producción de una empresa que se encarga de comercializar alrededor de unos setecientos mil corderos al año.
Esta empresa alimenta a los corderos hasta conseguir un peso apropximado de seis kilogramos. Cada kilogramo le cuesta a la empresa entre uno con veinte y uno con cuarenta euros por kilogramo, por lo que conseguir reducir este coste sería una gran ventaja para la compañia.

Para ello, el proyecto en el que está trabajando Robolab consta de varias fases de estudio e investigación y este proyecto es una parte de la fase en la que se creará un ecosistema de robots reales que replicarán las variantes de comportamiento de los corderos y así poder analizar más rapido y con menos recursos las hipótesis que se quieran evaluar.

Por lo tanto, este trabajo consistirá en realizar una simulación en la cual los robots virtuales recreen a los corderos y se puedan analizar los datos antes de recrear el mismo comportamiento en robots reales.

Capítulo 2

Objetivo general del proyecto

El objetivo principal es conseguir que los cinco robots simulen algunos de los comportamientos principales de los corderos y que publiquen cada cierto tiempo su posición en tiempo real.
Para conseguir este objetivo, hay que realizar varias tareas, que se definirán a continuación:

- Conseguir que un robot realice las cuatro tareas básicas del comportamiento de los corderos (andar, comer, beber y dormir) mediante el modelo de la máquina de estados.

- Una vez que el robot esté funcionando correctamente mediante la máquina de estados, investigar y estudiar el funcionamiento de los árboles de comportamiento.

- Implementar el funcionamiento que hemos conseguido recrear mediante la máquina de estados en un árbol de comportamiento.

- Una vez que el robot esté funcionando mediante el árbol de comportamiento, se añadirá un nuevo robot a la ejecución.

- Cuando el segundo robot esté implementado, estudiaremos los errores que se producen durante la interacción de ambos robots.

- Tras corregir los errores de interacción encontrados entre ambos, se implementará para las funciones de Andar, IrComer e IrBeber, un método en el cual si dos robots se cruzan, se esquiven y no acaben cruzandose.
- Cuando los dos robots estén funcionando correctamente, se añadirán los tres robots restantes para realizar la ejecución del estudio con cinco robots.
- Una vez se encuentre todo funcionando correctamente, se introducirá la simulación con las radios MDEK1001.
- Cuando la simulación de las radios esté implementada, se creará la función de publicación de la posición de los robots en tiempo real.

Capítulo 3

Estado de la técnica

3.1. Frameworks

3.1.1. Robocomp

Robocomp es un framework libre de desarrollo software para robots el cual sigue el paradigma de programación POC, programación orientada a componentes.

Este paradigma de programación se basa en que los usuarios construyen un mercado de componentes software para que las aplicaciones puedan reutilizar estos componentes que han sido creados anteriormente y se encuentran testeados. De esta manera, cuando se quiera construir una nueva aplicación, esta será creada de forma más rápida y robusta.

Además este paradigma de programación mejora algunos de los grandes problemas del desarrollo software, como son la encapsulación, seguridad y polimorfismo.

Un componente es una unidad de un programa software formado por interfaces y dependencias las cuales le permiten comunicarse con otros componentes.
Para que dos componentes se comuniquen mediante sus interfaces es necesario el uso de un middleware y en este proyecto el utilizado se llama Ice, ya que es el que utiliza Robocomp.

Como se ha explicado anteriormente, Robocomp es un framework libre de desarrollo software en el cual se pueden crear componentes de una manera sencilla e intuitiva. Para crear un componente se utiliza un archivo específico el cual tiene la extensión .cdsl[6]. En otro capítulo se explicará como es el archivo .cdsl de este proyecto.

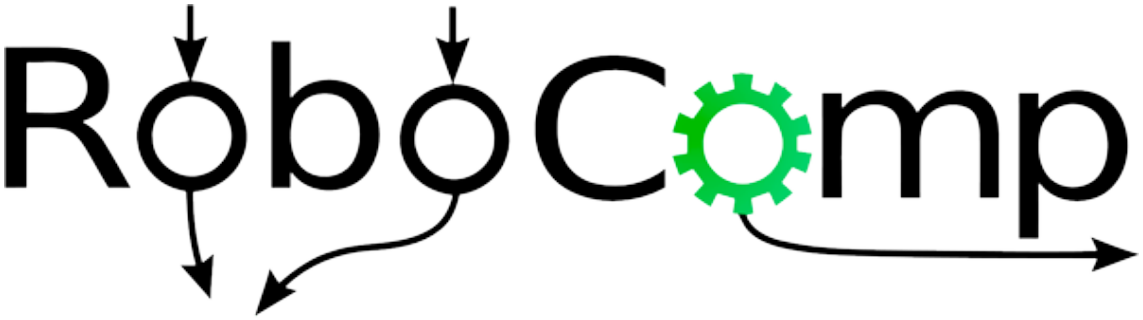

Figura 3.1: Robocomp

3.1.2. Robot Operating System

Robot Operating System, también conocido como ROS, es un framework para el desarollo software específico de robótica.

La filosofía de este framework es que todos los programas sean de libre uso, escalables e integrables a cualquier software de robótica, por ello sigue utilizando plataformas de software libre como KDL, OpenCV, etc.

La estructura de este framework es modular, ya que cada programa es denominado nodo, que a la vez se ejecuta dentro de un ejecutable que funciona como núcleo del sistema.

Figura 3.2: Robot Operating System

3.2. Simuladores

3.2.1. RCIS

El simulador RCIS es el simulador de robots de Robocomp, desarrollado por Robolab. Para ejecutar un mapa en este simulador será necesario un archivo de extensión .xml[5] con toda la información del mapa que queremos simular en su interior. Para ejecutar el simulador, será necesario escribir en la línea de comandos 'rcis' seguido del nombre del mapa que queremos abrir. En la siguiente imagen se verá un mapa simple con el simulador RCIS, sin ningún robot u obstáculo en él.

Figura 3.3: Mapa básico simulador RCIS

3.2.2. Gazebo

El simulador gazebo es un simulador 3D, dinámico y multi-robot, el cual permite evaluar el comportamiento de un robot en un mundo virtual, crear mundos virtuales usando herramientas sencillas, importar modelos ya creados y personalizar el diseño de los robots.
Gazebo es el simulador compatible con el framework ROS.

Figura 3.4: Gazebo

3.2.3. V-Rep

El simulador V-Rep es un simulador 3D, disponible para los principales sistemas operativos y de licencia libre. Está basado en una arquitectura de control distribuida, es decir, los scripts de control, pueden conectarse de forma directa a los objetos que existen en la escena y pueden ejecutarse de forma paralela.
Este simulador también tiene la ventaja de que te permite programar en muchos de los principales lenguajes como Python, C++, Java, Lua, etc. Además tiene una amplia documentación online de su uso para que sea más sencillo el aprendizaje de la herramienta.

Figura 3.5: V-Rep

3.3. Estructuras de control

3.3.1. Máquina de estados

Definición

Una máquina de estados es una abstracción matemática utilizada para diseñar algoritmos, compuesta por estados que hacen de intermediarios entre las entradas y las salidas.

Elementos que lo forman

Las máquinas de estados están formadas por tres elementos principales que son las entradas, las salidas y los estados.

Funcionamiento

La ejecución de los estados de la máquina van cambiando según las entradas que lleguen. Una vez que una entrada llega y comienza a ejecutarse un estado, este devuelve una salida con todos los cambios implementados. Tras esto, la máquina de estados vuelve a su estado inicial a la espera de que llegue otra entrada o bien la salida del último estado se convierte en una entrada.

3.3.2. Árboles de comportamiento

Durante el desarrollo del proyecto, se buscaron distintas tecnologías que pudiesen mejorar el funcionamiento de las máquinas de estados. Finalmente se decidió utilizar una tecnología muy usada en el mundo de los videojuegos y que poco a poco se va introduciendo en la robótica. Esta tecnología son los árboles de comportamiento.

Definición

Un árbol de comportamiento es una estructura de nodos jerárquicos que controlan la toma de decisiones de una entidad mediante inteligencia artificial.

Elementos que lo forman

Los árboles de comportamiento están formados por nodos y existen tres tipos principales de nodos, que son los siguientes:

- Compuesto: puede tener uno o más hijos. Estos hijos pueden ejecutarse en orden o de una manera aleatoria según se configure el nodo. Hay cuatro tipos de nodos compuestos que son los siguientes:
    - Secuencia: es un nodo el cual visita cada uno de los hijos en orden de izquierda a derecha. Si nada falla devuelve éxito al padre. Si cualquiera de los hijo falla, devuelve fallo. Funciona como una puerta lógica AND.
    - Selector: es un nodo el cual ejecuta el primer hijo, si este falla pasa al siguiente hasta que alguno de los hijos devuelve éxito.En caso de no encontrar ningín hijo que retorne éxitosamente, devolverá fallo. Funciona como una puerta lógica OR.
    - Secuencia aleatoria: funciona exactamente igual que la secuencia explicada anteriormente excepto que el orden que sigue es completamente aleatorio.
    - Selector aleatorio:funciona exactamente igual que el selector explicado anteriormente excepto que el orden que sigue es completamente aleatorio.

- Decorator (simple): este nodo solo puede tener un hijo. Su función principal es cambiar el resultado que le devuelve el hijo. Hay cuatro tipos de nodos decorator, que son los siguientes:
  - Inversor: es un nodo que cambiara el resultado. Es decir, si es éxito lo cambiará a fallo y si es fallo lo cambiará a éxito.
  - Exitoso: este nodo siempre devolverá éxito.
  - Repetidor: este nodo hace que se repita hasta que devuelva un resultado.
  - Repetidor hasta que falle: este nodo hace que se repita hasta que devuelva fallo, y entonces el nodo decorator devolverá éxito a su padre.
- Hoja: son nodos que no tienen ningún nodo hijo. Son los más importantes ya que definen todas las tareas que realizará el árbol,pero a su vez son los de nivel más bajo en el árbol de comportamiento.

Funcionamiento

El árbol de comportamiento funciona a traves de ticks[4], es decir, el root del árbol manda continuamente ticks sobre el nodo en el que se encuentra. Los nodos se ejecutan y devuelven el estado en el que se encuentran. Existen tres estados diferentes, que son los siguientes:

- Éxito: todo ha ido bien y se puede pasar al siguiente nodo.
- Fallo: si algún nodo devuelve este estado, el árbol termina la ejecución.
- Ejecutando: este estado indica que el nodo todavía no ha terminado de ejecutarse.

Capítulo 4

Equipo utilizado

4.1. Hardware

En la parte final de proyecto, una vez las simulación de varios robots funcionaban correctamente se han utilizado las radios MDEK1001 para obtener un conjunto de datos más realista simulando el ruido que estas generan.

4.1.1. Radios MDEK1001

Las radios MDEK1001 poseen una placa base formada por antenas y sensores integrados. Estas radios se caracterizan por que poseen varias tecnologías como son bluetooth y UWB(ultra wide-band).

Gracias a estas tecnologías, las radios tienen una gran precisión. Una de las ventajas de estas, es que para configurarlas y monitorizarlas tienen una aplicación móvil (solo válida para android superior a android 6.0) y un cliente web.

Estas han sido utilizadas para generar datos reales, con los cuales se hizo un estudio estadístico para calcular el valor del ruido que generan y así poder introducirlo en la simulación.

Figura 4.1: Rádio MDEK1001

Capítulo 5

Desarrollo del proyecto

En este capítulo se explicarán las distintas fases en las que se ha desarrollado el proyecto.

5.1. Problema a resolver

En este proyecto se han presentado dos problemas principales a resolver. El primero de ellos, es la localización de robots en tiempo real. Este se debe a que un objetivo futuro de este proyecto es la localización de animales en granjas en tiempo real.

El segundo problema presentado es la simulación de comportamientos biológicos de animales en los robots. Un robot siempre va a seguir una rutina, como cualquier ser vivo pero en el caso del robot será secuencial o aleatoria lo que hace que no sea real y no se asemeje al comportamiento biológico de un animal.

5.2. Propuesta de solución al problema presentado

Ya conocidos los dos principales problemas a resolver en este proyecto, se procederá a explicar las soluciones propuestas para su resolución.

5.2.1. Localización en tiempo real de robots

Esta fase estará compuestos por dos etapas. La primera es una simulación de los robots mediante el simulador rcis. En la segunda etapa se simulará el ruido generados por las radios MDEK1001 y así se podrá crear un modelo más realista.

Simulación de robots

Para la simulación se usará el simulador RCIS ya nombrado anteriormente.
Durante la simulación se puede acceder directamente a las coordenadas exactas del robot en todo momento, pero esto no sería posible en un modelo en la vida real, por tanto para hacer que la simulación sea lo más precisa posible a un experimento real, se simulará el ruido que generaría una radio.

El proceso que se realizará será coger datos reales de una radio usada en un experimento que ha sido colocada en una posición conocida del mapa durante una hora.

Una vez se han conseguido todos esos datos, se generará una gráfica de la cual se podrá saber la media de error que tendría una radio, así como la varianza, el mayor y el menor error.

Teniendo en cuenta estos datos, se puede generar siempre un número aleatorio, dentro de los umbrales de error calculados previamente usando las radios, mediante una distribución normal, para sumarle a nuestras coordenadas exactas y así generar dentro de la simulación el ruido generado por el funcionamiento de las radios bluetooth.

Con esta solución se conseguirá que la simulación con robots virtuales sea un experimento similar sobre como funcionaría en robots físicos midiendo sus coordenadas con esta radios.

5.3. Iniciación del robot al entorno

Para crear el robot, el primer paso es instalar Robocomp en la máquina donde se realizará el trabajo.

Una vez que Robocomp se ha instalado correctamente, el siguiente paso es crear un componente.

Para realizar esta acción, el primer paso es crear una carpeta y dentro de dicha carpeta ejecutaremos el siguiente comando

```
robocompdsl cordero.cdsl
```

Este comando generará el archivo .cdsl que contrendrá la configuración de las comunicaciones del robot. Este archivo se tendrá que modificar para que el robot funcione correctamente.

El siguiente código es el resultado del fihero cordero.cdsl utilizado en este proyecto.

```
import "DifferentialRobot.idsl";
import "Laser.idsl";
Component MyFirstComp
{
    Communications
    {
        requires DifferentialRobot, Laser;
    };
language Cpp;
gui Qt(QWidget);
};
```

Como se puede observar en el anterior fragmento de código, el robot utiliza la librería 'laser.idsl'[8]. Esto significa que el robot hace uso de objeto de la librería 'laser.idsl' con el cual este será capaz de identificar los obstáculos que se encuentre por el mapa.

Para representar el robot en el simulador hay que crear un archivo .xml con la configuración sobre la posición del robot y del laser con el mapa y el ejemplo de la configuración para el primer robot creado en el proyecto es la siguiente:

```
<differentialrobot id="base1" port="11004">
<mesh id="base_robex1"
file="/home/robocomp/robocomp/files/osgModels/robex/robex.ive"
    tx="0" ty="0" tz="-0" scale="1000" collide="1"/>
        <translation id="laserPose1" tx="0" ty="140" tz="200">
                <laser id="laser1" port="11003" measures="1000"
                min="100" max="30000" angle="3" ifconfig="10000" />
                <plane id="sensorL1" nz="1" pz="-200" size="100"
                repeat="1" texture="#ff0000" />
    </translation>
<translation id="robotGeometricCenter1" tx="0" ty="0" tz="0">
</translation>
</differentialrobot>
```

A continuación se adjunta una imagen en la cual se puede observar el robot con el láser en el mapa.

Figura 5.1: Ejemplo robot con láser

Una vez que el archivo cordero.cdsl está correctamente creado y configurado, se

ejecutará el siguiente comando para que se creen todos los archivos necesarios para poder lanzar el robot.

```
robocompdsl cordero.cdsl .
```

Este comando creará todos los archivos C++ en este caso, los cuales se modificarán para que el robot se comporte de la manera más parecida posible a un animal real.

5.4. Creación y desarollo de la máquina de estados

Para que el robot tenga un comportamiento lo más similar posible a un cordero, se ha creado una máquina de estados en la que se simulan los comportamientos básicos más importantes de los corderos.

La máquina de estados está compuesta por los estados por los que pasará el robot durante la ejecución y por las interacciones entre ellos, ya que un estado puede pasar a uno u otro distinto dependiendo del valor de la variable que controla la máquina de estados denominada 'Estado'. Esta variable contendrá el valor del nombre del estado en el que se encuentre en cada momento el robot y se irá actualizando una vez se realicen las acciones definidas para cada estado y siempre dependiendo del resultado de esa acción.

A su vez, la máquina de estados trabaja con otra variable denominada 'EstadoEnUso' la cual almacena la acción en la que se encuentra el robot.

La diferencia entre las variables 'Estado' y 'EstadoEnUso' es que la primera puede almacenar todos los estados que se explicarán en el siguiente punto, mientras que la segunda únicamente podra almacenar las acciones que realiza el animal (comer, beber, dormir y andar) y que se intentan simular mediante los robots.

A continuación se adjunta una imagen en la cual se verá la estructura de la máquina de estados definida inicialmente para este proyecto.

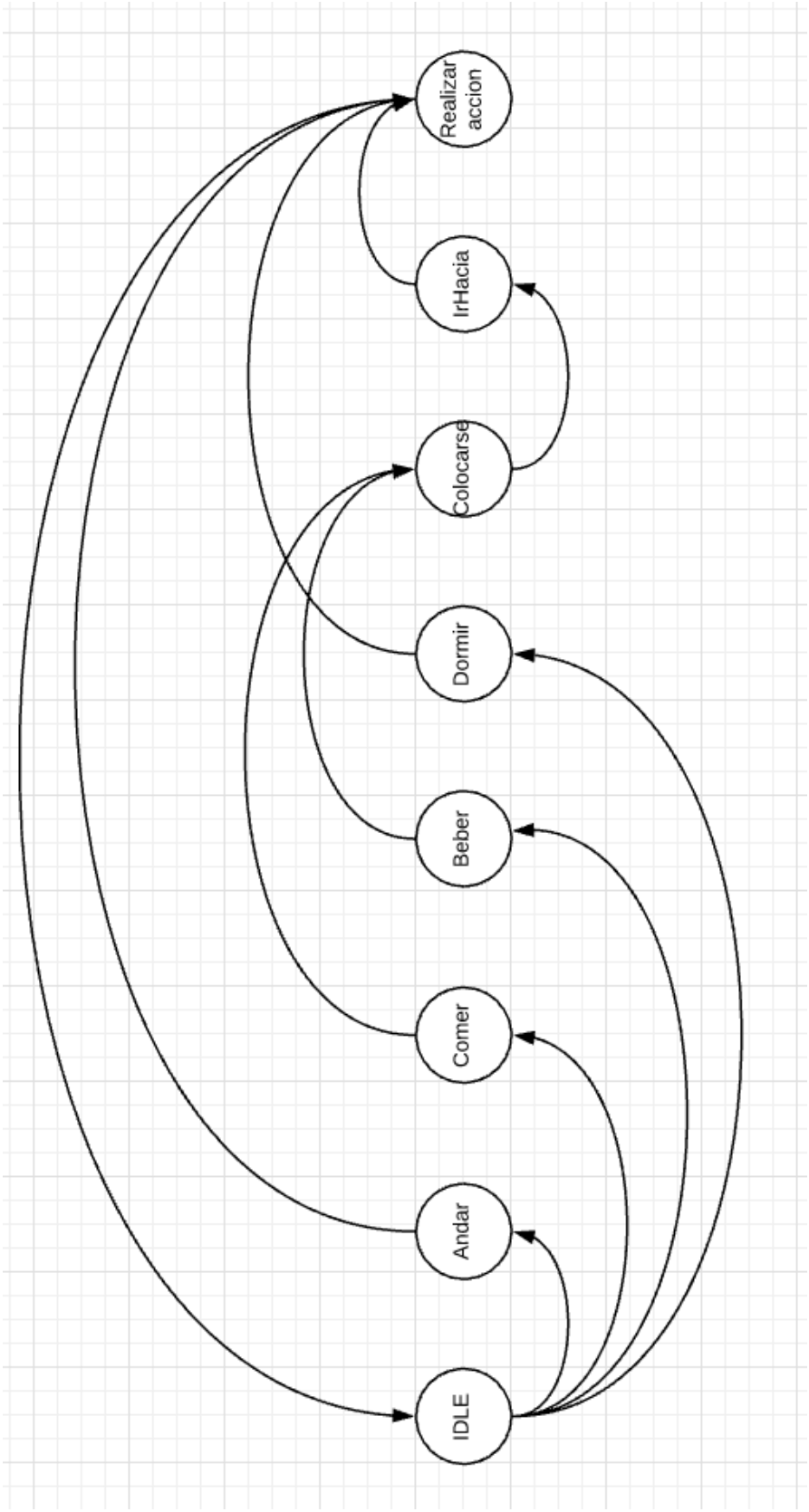

Figura 5.2: Máquina de estados.

5.5. Funcionamiento de la máquina de estados

Como se puede observar en la imagen anterior, la máquina de estados consta de ocho estados diferentes. A continuación se analizará el funcionamiento de cada uno de ellos.

- IDLE: el estado IDLE es el estado inicial de la máquina de estados. Este estado se encarga de elegir cual será la siguiente acción que realizará el robot. Esto se realizará de manera aleatoria sin poder repetir un mismo movimiento dos veces seguidas.

- Andar: el estado andar es el que se encarga de hacer que se mueva el robot por el mapa. Lo primero será controlar el tiempo que el robot se está moviendo de forma aleatoria por el mapa. Este estado, tiene tres opciones internas, que son las siguientes:

    - Sin obstáculos: en este caso el robot comprueba mediante su propio láser comprueba que no tiene ningún objeto delante y puede avanzar sin problema.

    - Caso objeto: en este caso el robot detecta que tiene muy cerca un objeto y antes de chocar con él, comprueba por qué lado esquivaría antes el objeto girando hacia izquierda o derecha y después sigue desplazándose por el mapa. Este mismo caso, es el que usa el robot cuando ve que va a chocar con una pared. Además, este caso también será utilizado cuando el robot se cruce con otro que se encuentre en el mapa, ya que lo tratará como si fuera un objeto.

    - Caso esquina: en este caso el robot detecta que se ha quedado bloqueado en una esquina y no tiene una salida para ninguno de los dos lados, por tanto cuando detecta que está bloqueado, se queda parado y comienza a girar sobre su propio eje hasta encontrar la primera salida posible. Una vez la ha encontrado, sale de la esquina y vuelve a caminar por el mapa.

A continuación se muestran varias imagenes en las que se pueden observar los distintos casos que se puede encontrar el robot en el mapa durante el estado andar.

(a) Caso esquina (b) Caso pared (c) Caso objeto

Figura 5.3: Estado andar

- Comer: este estado se divide en dos acciones. La primera es cambiar el valor de la variable 'Estado' a colocarse. La segunda es cambiar el valor de la variable 'EstadoEnUso' a comer, que será utilizada más adelante por otros estados.
- Beber: este estado se comporta igual que el estado comer. En el caso de la variable 'Estado' también lo cambia a colocarse pero en el caso de la variable de 'EstadoEnUso' lo pone al valor beber.
- Dormir: en este estado se realizan dos acciones como en los dos estados anteriormente explicados. El primer caso es cambiar el valor de la variable 'Estado' a realizarAccion. El segundo caso es cambiar el valor de la variable 'EstadoEnUso' a dormir.
- Colocarse: este estado lo primero que hace es comprobar cual es el valor de la variable 'EstadoEnUso'. Después crea una variable del tipo QPointF en la cual, si el 'EstadoEnUso' es comer guarda las coordenadas de donde se encuentra el comedero y si el estado en uso es beber, guardará las coordenadas del bebedero. Una vez que ya tiene las coordenadas correctamente guardadas, el robot

comienza a girar hacia al punto al que quiere dirigirse.
Cuando el ángulo entre el robot y el punto al que quiere dirigirse es menor de 0.001 radianes, el robot se para y cambia el valor de la variable 'Estado' a IrHacia.

- IrHacia: en este estado, lo primero que hace el robot es comprobar cual es el valor de la variable 'EstadoEnUso', para guardar las variables del comedero o bebedero dependiendo donde vaya a desplazarse el robot. Tras esto, este estado pone al robot en movimiento, avanzando en línea recta, y va comprobando mediante las coordenadas guardadas del objetivo donde queremos llegar y una vez que hemos llegado, para al robot, y cambia el valor de la variable 'Estado' a realizarAccion. En este punto hay que destacar que si durante este estado, el robot se encuentra con algún obtaculo, es este caso otro robot que pueda estar ejecutándose, este lo esquivará y recalculará de nuevo la posición del comedero ó bebedero para continuar avanzando hasta su posición.

- Realizar acción: en este estado, utiliza la variable 'EstadoEnUso' para decidir cual de todas las acciones tiene que llevar a cabo. Esto lo hará mediante un switch/case, y dentro de cada caso cargará el tiempo que tiene que estar realizando la acción pertinente. Tras ello, quedará esperando en un bucle el tiempo necesario. Una vez ha terminado, mostrará un mensaje por pantalla del tiempo que ha estado realizando la acción. Para finalizar, cambia el valor de la variable estado a IDLE y así el robot seguirá realizando acciones.

5.6. Creación y desarrollo del árbol de comportamiento

Una vez que el robot se encuentra funcionando correctamente con la máquina de estados, se pasará a implementar dicho robot utilizando un árbol de comportamiento ya que como se ha explicado anteriormente, esta tecnología mejora el rendimiento en

el funcionamiento del robot. En la siguiente imagen se muestra la definición del árbol de comportamiento utilizado en este trabajo. La definición, creación e implementación del árbol de comportamiento realizado en este proyecto se encuentra en el anexo 1.0.

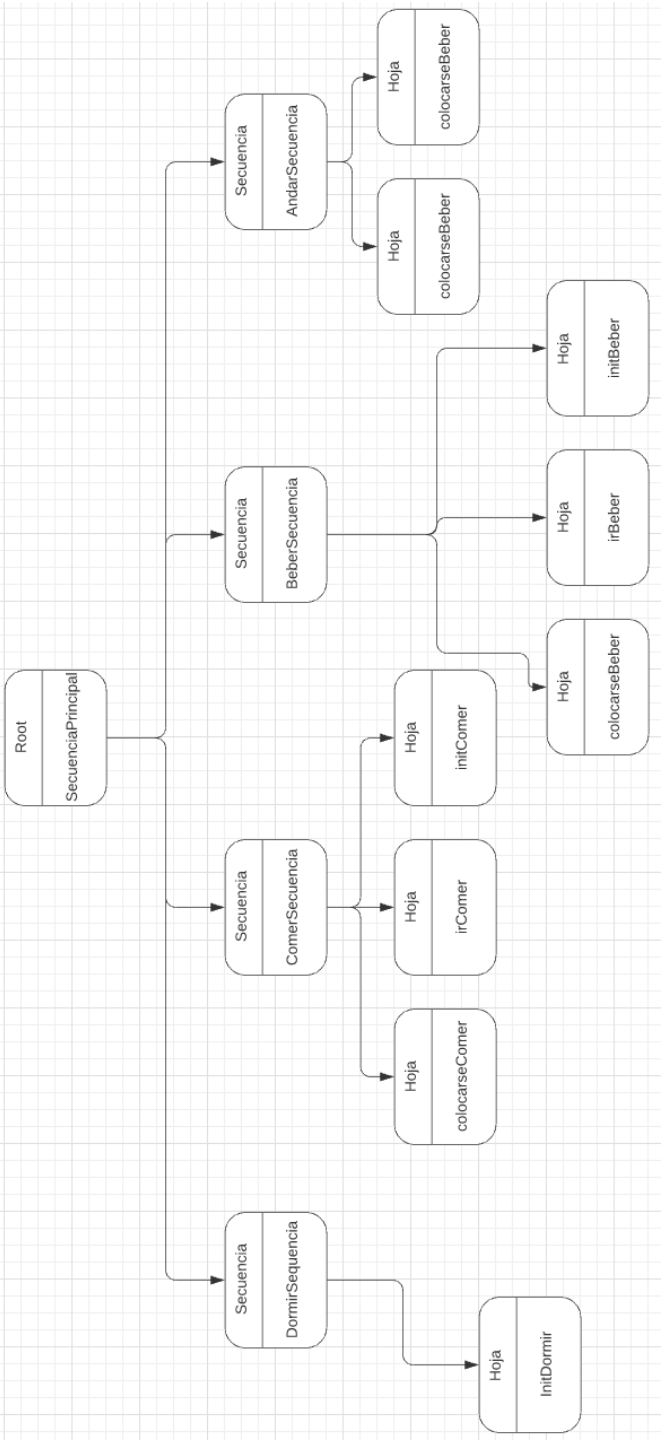

Figura 5.4: Árbol de comportamiento.

5.7. Funcionamiento del árbol de comportamiento

Como se ha explicado anteriormente, el árbol funciona mediante ticks, los cuales son mandados a las secuencias y a su vez a la hoja[7] correspondiente para que esta se ejecuta. Cuando el nodo devuelve éxito pasa al siguiente. Seguidamente, se explicará como funciona cada una de las partes del árbol.

5.7.1. Secuencia principal

La secuencia principal es la que se encarga de mandar los ticks a las demás secuencias. Esta tiene como hijos las otras cuatro secuencias principales, que son las que se encargan de realizar las cuatro acciones que realiza el robot simulando un animal real. Se encarga de ejecutar en orden las secuencias de dormir, comer, beber y andar. Una vez que termina una ejecución, vuelve a comenzar y así el robot nunca se encuentra sin realizar ninguna acción.

5.7.2. Secuencia dormir

Esta secuencia es la que se encarga de realizar la acción de dormir y tiene un nodo hoja, que es el que ejecuta la acción.

- initDormir: esta hoja se encarga de simular el comportamiento de dormir. Para ello, para el robot, y se queda parado durante siete segundos. Durante este tiempo, todos los ticks que le llegan los devuelve con el valor de 'ejecutando'. Una vez terminan esos siete segundos, el siguiente tick que le llegue le devolverá como éxito y así el siguiente tick llegará al siguiente nodo.

5.7.3. Secuencia comer

Esta secuencia es la que se encarga de realizar la acción de comer. Para realizar esta acción completa se requiere de diferentes acciones , por lo tanto esta secuencia la forman tres hojas, que son las siguientes.

- colocarseComer: esta hoja es la encargada de que el robot se posicione de frente al comedero. Esto lo hace parándose y girando sobre su eje hasta que el ángulo que forma con el comedero es menor de 0.001 radianes. Durante todo este tiempo que está calculando el ángulo, cada tick que le llega le devuelve con el valor de ejecutando, y hasta que no comprueba que el ángulo es menor a un valor proporcionado por el usuario, no manda el tick como éxito para que pueda ejecutarse el siguiente nodo.

- irComer: esta hoja se encarga de poner en movimiento al robot, una vez que ya está colocado, hacia el comedero. En el momento que empieza a moverse todos los ticks los devuelve como ejecutando. Cuando comprueba que ha llegado al comedero, para el robot y manda el siguiente tick como éxito. Si por el camino hacia el comedero, el robot se cruza con otro robot, este le esquivará y así evitarán una colisión.

- initComer: esta hoja se encarga de realizar la simulación de la acción comer. Una vez el robot está en el comedero, este se estará quieto durante cuatro segundos. Cuando termine este tiempo, el siguiente tick lo devolverá como éxito y el nodo de la secuencia principal podrá ejecutar la siguiente secuencia, ya que la accion 'comer' ha terminado correctamente.

5.7.4. Secuencia beber

Esta secuencia es la que se encarga de realizar el método beber. Al igual que la secuencia de comer, esta secuencia la forman varias hojas ya que el robot para beber tiene que desplazarse hacia el bebedero y no siempre está ni cerca, ni centrado respecto al bebedero. Estas tres hojas son las que forman esta secuencia:

- colocarseBeber: esta hoja es la encargada de que el robot gire sobre su eje y quede colocado enfrente del bebedero. Para ello, estando parado, el robot gira sobre su eje y cuando el ángulo que forma el robot respecto al bebedero es

menor que 0.001 radianes, en el siguiente tick devolverá el valor éxito y pasará a ejecutarse el siguiente nodo.

- irBeber: una vez que el robot está colocado, esta hoja se encarga de que el robot avance hacia el bebedero. Mientras avanza hacia este, va comprobando si ya ha llegado mediante las coordenadas, y todos los ticks los devuelve como ejecutando. El momento que el robot detecta que ha llegado al bebedero, el robot se para y devuelve el siguiente tick que le llega como éxito. Si por el camino hacia el bebedero, el robot se cruza con otro , este le esquivará utilizando el mismo proceso que en el caso de la acción 'comer'.

- initBeber: esta hoja se encarga de realizar la simulación de la acción beber. Para ello, queda parado el robot durante cinco segundos. Una vez que pasa ese tiempo, termina y manda el tick con el valor de éxito y así el árbol puede pasar a la siguiente secuencia ya que la secuencia beber ha terminado satisfactoriamente.

5.7.5. Secuencia andar

Esta secuencia es la que se encarga de que el robot se mueva por el mapa siguiendo una ruta aleatoria durante un periodo de tiempo determinado, sin tener una posición final determinada. Para realizar esta secuencia completa, tienen que ejecutarse correctamente dos hojas. Estas hojas son las siguientes.

- initAndar: esta hoja es la encargada de iniciar el contador de tiempo que el robot tiene que estar caminando por el mapa. Una vez que resetea esa variable, el siguiente tick que le llega le devolverá como éxito.

- actionAndar: esta hoja es la encargada de poner en movimiento al robot por el mapa. Para ello controla el tiempo que lleva moviéndose y cuando llega a los quince segundos , el siguiente tick que reciba será devuelto como éxito. Además, mediante el láser del robot, controla que en todo momento no choque con ningún objeto, ni con ningún otro robot. Cualquier objeto que se encuentro lo rodeará y seguirá su camino.

5.8. Introducción final de cinco robots en el entorno

Una vez que un robot se encuentra funcionando correctamente mediante el árbol de comportamiento, el siguiente paso es incorporar un nuevo robot para comprobar los posibles problemas que pueden aparecer cuando hay más de una instancia de robot ejecutándose en el mismo mapa . Se decide comenzar esta prueba únicamente con dos robot porque es la manera más simple de identificar los errores que pueden aparecer. Una vez que estos errores se localicen y se solucionen, se procederá a añadir todos los necesarios para llevar a cabo la finalidad de este proyecto. Los errores que se han encontrado son lso siguientes:

- Choques cuando los robots se encontraban en la función andar: estudiando el comportamiento de los robot, se observó que no se había tenido en cuenta la posibilidad de que dos robots chocasen entre ellos.
  Para solucionar este error, se implementó una función en la cual si un robot detectaba algún objeto en su láser y estaba dentro de un pequeño umbral, el robot giraría y tomaría otra dirección.

- Error en las funciones en las cuales se llamaba al robot mediante su id: este error se encontró cuando mse observó que el segundo robot actualizaba las coordenadas con las del primero ya que tenían el mismo identificador en el archivo de configuración de los robots.
  Para solucionar este problema, se creó una variable dentro del fichero de configuraciónes en la que se indica el id/nombre del robot y será esa variable la utilizada por las distintas funciones para que los robots sean independientes unos de otros y no compartan variables ni estados.

- Choque cuando se encontraban en las funciones IrComer e IrBeber: en este caso nos encontramos con el mismo problema que se ha comentado en la función andar, pero este problema hay que resolverlo de problema diferente ya que el robot va camino al comedero/bebedero entonces no puede girar simplemente en otra dirección sino que, una vez ha esquivado el obstáculo, debe recalcular la

posición del comedero ó bebedero para terminar de realizar la acción en la que se encontraba.

Para ello se ha implementado una función en la cual se define que si el robot se encuentra con un obstáculo, este lo rodea por la izquierda y vuelva a colocarse en dirección al comedero/bebedero. Esta solución es válida se encuentre a uno o más robots por el camino.

- Error del tiempo realizando una acción: al estudiar la ejecución de los cinco robots, se comprobó que el robot se quedaba realizando una acción durante un tiempo fijo, algo que no concuerda con el comportamiento de un animal real.

  Para ello, se sustituyó el número fijo de segundos que el robot tenía que estar realizando una acción, por un número aleatorio con una distribución normal.

A continuacion se mostrará una imagen reproducciendo la colocación de varios de los robots en el mapa durante la misma ejecución.

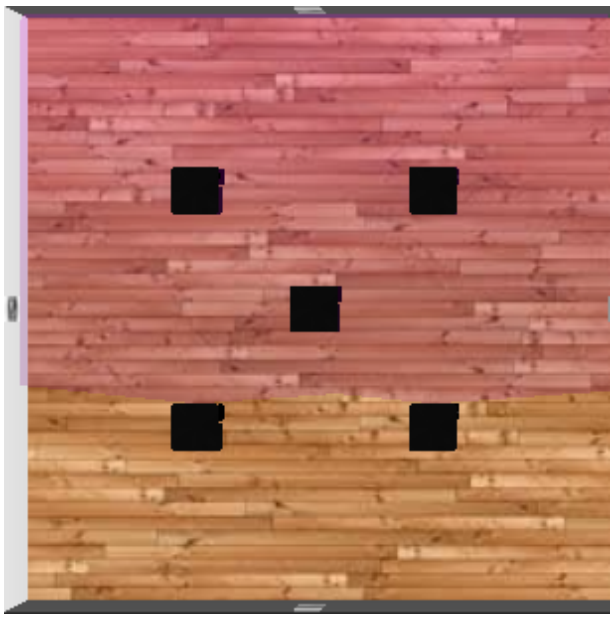

Figura 5.5: Los cinco robots en el mapa.

5.9. Publicación de las coordenadas

La última función que se ha implementado en el proyecto ha sido la publicación de la posicion de los robots en tiempo real. En la simulación, en cualquier momento es posible acceder a la posición exacta del robot y eso no es eficiente para la finalidad del proyecto. Por lo que se ha decidido simular el comportamiento utilizando las radios MDEK1001, las cuales han sido mencionadas anteriormente.

Para ello, se colocó una radio durante una hora en un mapa en el cual se conocen las coordenadas exactas donde se encuentra. Tras ello, se obtienen los datos y se calculan datos estadísticos como la media, la varianza, la mediana, la varianza, etc.

Una vez se tienen esos datos, se genera un número aleatorio basándose en ellos, y se le suma a las coordenadas que se quieren publicar. Esta publicación se hace cada 3 segundos, con el siguiente formato, en un archivo ”txt”.

```
x#y
```

Capítulo 6

Resultados y líneas futuras

Una vez que el proyecto se ha desarrollado al completo, hay que comprobar si los objetivos impuestos inicialmente se han cumplido y cuales han sido los resultados.

6.1. Resultados

Inicialmente se estableció como objetivo principal conseguir que cinco robots recrearan el comportamiento fisiológico de un cordero y que cada uno publicara su localización en tiempo real cada cierto umbral de tiempo en un fichero.

Para ello, existían varias tareas principales a desarrollar que eran las siguientes:

- Implementar las funciones básicas que realiza un cordero en un robot mediante máquinas de estado.
- Estudiar funcionamiento y creación de árboles de comportamiento. Crear el mismo sistema implementado mediante máquinas de estado pero con árboles de comportamiento.
- Introducir un segundo robot en la simulación y estudiar qué errores pueden aparecer y la iteracción que existe entre ambos robots.
- Introducir los cinco robots.

- Publicación de la posición de los robots en tiempo real.

En general, todos estos objetivos que fueron marcados han sido desarrollados y probados para certificar su correcto funcionamiento. Cabe decir, que se trata de una versión inicial la cual deberá ser actualizada a medida que el sistema vaya aumentando.

6.2. Líneas futuras

Como hemos mencionado anteriormente, este sistema que ha sido desarrollado se trata de una versión inicial. Esta versión puede ser mejorada y ampliada. Algunas de estas ideas de ampliación surgieron durante el proceso del desarrollo de estey ayudaran a mejorar el rendimiento y los resultados de este proyecto en el futuro. Algunas de estas mejoras son:

- Modificación de la librería utilizada de los árboles de comportamiento: con un problema que nos encontramos a la hora de utilizar el árbol fue que al cruzarse dos robots en las acciones de irComer y irBeber, no podíamos volver al nodo padre(secuencia) para volver a realizar la acción de colocarnos hacia nuestro objetivo.

  Esto se debe a que mediante los tres valores con lo que se devuelven los ticks solo se puede avanzar hacia el siguiente nodo, quedarnos ejecutando el nodo en el que nos encontramos o bien parar la ejecución del árbol.

  Por lo que una opción que se pensó fue modificar la librería del árbol de comportamiento y crear un nuevo valor para devolver en los ticks, el cual pueda volver la ejecución al nodo padre y así poder comenzar de nuevo la secuencia.

- Añadir funcionalidades menos básicas que realicen los robots y así hacer que el modelo sea más real. Algunas de estas funcionalidades pueden ser por ejemplo jugar, en la que dos o más robots interactúen entre ellos.

- Crear una variable que sea tenerSed y otra que sea tenerHambre las cuales se carguen cuando se esté en el comedero y el bebedero pero se vayan descargando

mientras el robot va realizando otras acciones por el mapa y así el robot vaya a comer/beber solo cuando esta variable tenga un nivel bajo y necesite recargarla. De esta manera podríamos gestionar los ticks mediante algunos eventos.

- Implementar un botón en el simulador el cual al pulsarlo te exporte un fichero comprimido con los archivos en los cuales se están almacenando las coordenadas del recorrido de los robots.

- Implementar un botón en el simulador el cual al pulsarlo te genere mediante los archivos de las coordenadas de los robots una gráfica en la que se puedan ver pintadas el recorrido de cada robot.

- Implementar este sistema en robots reales a través de LearnBlock[3]. De esta manera se daría un paso más hacia la implementación en animales reales.

Capítulo 7

Conclusiones

El proyecto en el que se engloba este trabajo se llama 'Caracterizacion etológica de corderos con tecnologías roboticas', en el cual se encuentra trabajando actualmente Robolab. Como se ha comentado en el apartado anterior, el resultado del proyecto parece satisfactorio ya que se ha conseguido emular las funciones básicas de un animal como son comer, beber, andar y dormir mediante robots virtuales, utilizando las herramientas explicadas en este documento, que era uno de los objetivos principales que se propuso al inicio del proyecto, junto al posicionamiento en tiempo real del animal, que también se ha simulado utilizando las radios MDEK1001.

El resultado obtenido parece un buen punto de partida para llegar al objetivo final el cual es implantar este desarrollo en animales reales, aunque para eso queda bastante recorrido, y el siguiente paso a realizar sería la simulación utilizando robots físicos, el cual parece posible implementar utilizando el desarrollo realizado en este trabajo.

Personalmente este proyecto me ha ayudado bastante desde un punto de vista académico debido a que he podido aprender y trabajar con nuevas tecnologías y sofware como son las máquinas de estados, árboles de comportamiento, Robocom, etc.

Me ha parecido un trabajo bastante interesante debido a que he podido comprobar

como el uso de la tecnología puede ayudar al sector primario y simular, en este caso, el comportamiento de animales de la forma más realista posible con las herramientas utilizadas y espero que en un futuro pueda ayudar a que este proyecto se acabe desarrollando en su totalidad.

Anexos

Apéndice A

Anexo 1.0

En este anexo se adjuntará el código de la definición, creación e implementación del árbol de comportamiento realizado en este proyecto.

A.1. Creación del árbol

A.1.1. Definición del árbol

```
void SpecificWorker::createTree(BrainTree::BehaviorTree &btree)
{
    auto mainSequence = std::make_shared<BrainTree::Sequence>();

    //Dormir
    auto sleepSequence = std::make_shared<BrainTree::Sequence>();
    auto initSleep = std::make_shared<ActionInitSleep>();

    //Comer
    auto eatSequence = std::make_shared<BrainTree::Sequence>();
    auto colocarseComer = std::make_shared<ActionStandToEat>(this);
    auto irComer = std::make_shared<ActionGoToEat>(this);
```

```
auto initEat = std::make_shared<ActionInitEat>();

//Beber
auto drinkSequence = std::make_shared<BrainTree::Sequence>();
auto colocarseBeber = std::make_shared<ActionStandToDrink>(this);
auto irBeber = std::make_shared<ActionGoToDrink>(this);
auto initDrink = std::make_shared<ActionInitDrink>();

//Andar
auto walkSequence = std::make_shared<BrainTree::Sequence>();
auto initWalk = std::make_shared<ActionInitWalk>(this);
auto andar = std::make_shared<ActionWalk>(this);

mainSequence->addChild(sleepSequence);
mainSequence->addChild(eatSequence);
mainSequence->addChild(drinkSequence);
mainSequence->addChild(walkSequence);

sleepSequence->addChild(initSleep);

eatSequence->addChild(colocarseComer);
eatSequence->addChild(irComer);
eatSequence->addChild(initEat);

drinkSequence->addChild(colocarseBeber);
drinkSequence->addChild(irBeber);
drinkSequence->addChild(initDrink);

walkSequence->addChild(initWalk);
```

```
    walkSequence->addChild(andar);


    btree.setRoot(mainSequence);
    btree.update();
}
```

A.2. Implementación de los nodos hoja

A.3. Implementación de los nodos hoja

A.3.1. Dormir

```
class ActionInitSleep : public BrainTree::Node
{
    public:
        ActionInitSleep()
        {
        }
        Status update() override
        {
            if(first_epoch)
            {
                reloj.restart();
                first_epoch = false;
                return Node::Status::Running;
            }
            else
            {
                if(reloj.elapsed() > 4000)
```

```
            {
                first_epoch = true;
                return Node::Status::Success;
            }
            else
            {
                return Node::Status::Running;
            }
        }
    }
private:
    bool first_epoch = true;
    QTime reloj;
};
```

A.3.2. Colocarse para comer

```
class ActionStandToEat : public BrainTree::Node
{
    public:
        ActionStandToEat(SpecificWorker *x)
        {
            this->sp = x;
        }
        Status update() override
        {
            QPointF t;
            t = sp->getFoodDispenser();
            float angle = 0;
```

```
            //Paso el punto, de coord del mundo al robot
            QVec p = sp->innerModel->transform(sp->robotName.c_str()
            ,QVec::vec3(t.x(),0,t.y()), "world");
            angle = qAtan2(p.x(),p.z()); // calculo angulo en rads
            if(fabs(angle) < 0.001)
            {
                sp->differentialrobot_proxy -> setSpeedBase(0,0);
                return Node::Status::Success;
            }else
            {
                sp->differentialrobot_proxy -> setSpeedBase(0,angle);
                return Node::Status::Running;
            }
        }
    private:
        SpecificWorker* sp;
};
```

A.3.3. IrComer

```
class ActionGoToEat : public BrainTree::Node
{
    public:
        ActionGoToEat(SpecificWorker *x)
        {
            this->sp = x;
            esquivar = false;
        }
        Status update() override
```

```
{
    float coordX;
    float coordY;
    coordX = sp->getCoordXFood();
    coordY = sp->getCoordYFood();
    if((((coordX - sp->bState.x) < 20) &&
    (coordX - sp->bState.x) > -20) &&
    (((coordY - sp->bState.z) < 20) &&
    (coordY - sp->bState.z) > -20))
        {
                sp->differentialrobot_proxy ->
                setSpeedBase(0,0);
                return Node::Status::Success;
        }
        else
        {
        RoboCompLaser::TLaserData ldata
        = sp->laser_proxy->getLaserData();
        float center = ldata[ldata.size()/2].dist;
        float right = ldata.front().dist;
        //sort laser data from small to large
        distances using a lambda function.
        std::sort( ldata.begin(), ldata.end(),
        [](RoboCompLaser::TData a, RoboCompLaser::TData b)
        { return a.dist < b.dist; });
        if((ldata.front().dist < 350 || center < 350))
            {
                    sp->differentialrobot_proxy->
            setSpeedBase(0, -0.6);
```

```
            if(right > 450){
                esquivar = true;
            }
            return rodearObj(center,right);
            }

        if(esquivar){
            return colocarse();

        }else
        {
                    sp->differentialrobot_proxy ->
                    setSpeedBase(500,0);
            return Node::Status::Running;
        }
        }
}
Status colocarse()
{
    QPointF t;
    t = sp->getFoodDispenser();
    float angle = 0;
    //Paso el punto, de coord del mundo al robot
    QVec p = sp->innerModel->transform
    (sp->robotName.c_str(),
    QVec::vec3(t.x(),0,t.y()), "world");
    angle = qAtan2(p.x(),p.z()); // calculo angulo en rads
    qDebug() << "Angulo = " << angle;
    if( fabs(angle) < 0.001)
```

```
        {
            sp->differentialrobot_proxy -> setSpeedBase(0,0);
            esquivar = false;
            return Node::Status::Running;
        }else
        {
            sp->differentialrobot_proxy -> setSpeedBase(0,angle);
            return Node::Status::Running;
        }
    }

    Status rodearObj(float c, float r)
    {

        if(r > 400 && r < 500 && c > 450)
        {
            qDebug() << "rodear : avanzo ";
            sp->differentialrobot_proxy -> setSpeedBase(100,0);
            esquivar = true;
            return Node::Status::Running;
        }
        else
        {
            if(r < 400 || c < 350)
                    {
                                sp->differentialrobot_proxy ->
                                setSpeedBase(100,-0.6);
                return Node::Status::Running;
                    }
```

```
                    else if(r > 500)
                    {
                            sp->differentialrobot_proxy ->
                            setSpeedBase(100,0.25);
                return Node::Status::Running;
                    }
            }
        }
    private:
        SpecificWorker* sp;
        bool esquivar;
        //void colocarse();
};
```

A.3.4. IniciarComer

```
};
class ActionInitEat : public BrainTree::Node
{
    public:
        ActionInitEat()
        {
        }
        Status update() override
        {
            if(first_epoch)
            {
                reloj.restart();
                first_epoch = false;
```

```
            return Node::Status::Running;
        }
        else
        {
            if(reloj.elapsed() > 4000)
            {
                first_epoch = true;
                return Node::Status::Success;
            }
            else
            {
                return Node::Status::Running;
            }

        }
    }
private:
    bool first_epoch = true;
    QTime reloj;
};
```

A.3.5. Colocarse para beber

```
class ActionGoToDrink : public BrainTree::Node
{
    public:
        ActionGoToDrink(SpecificWorker* x)
        {
            this->sp = x;
```

```
        }
        Status update() override
        {
            float coordX;
            float coordY;
            coordX = sp->getCoordXWater();
            coordY = sp->getCoordYWater();
            if((((coordX - sp->bState.x) < 20) &&
            (coordX - sp->bState.x) > -20) &&
            (((coordY - sp->bState.z) < 20) &&
            (coordY - sp->bState.z) > -20))
                {
                        sp->differentialrobot_proxy ->
                        setSpeedBase(0,0);
                        return Node::Status::Success;
                }
                else
                {
                        sp->differentialrobot_proxy ->
                        setSpeedBase(500,0);
                return Node::Status::Running;
                }
        }
    private:
        SpecificWorker* sp;
};
```

A.3.6. IrBeber

```
class ActionGoToDrink : public BrainTree::Node
{
    public:
        ActionGoToDrink(SpecificWorker* x)
        {
            this->sp = x;
            esquivar = false;
        }
        Status update() override
        {
            float coordX;
            float coordY;
            coordX = sp->getCoordXWater();
            coordY = sp->getCoordYWater();
            if((((coordX - sp->bState.x) < 20) &&
            (coordX - sp->bState.x) > -20) &&
            (((coordY - sp->bState.z) < 20) &&
            (coordY - sp->bState.z) > -20))
                {
                        sp->differentialrobot_proxy ->
                        setSpeedBase(0,0);
                        return Node::Status::Success;
                }
                else
                {
                RoboCompLaser::TLaserData ldata = sp->
                laser_proxy->getLaserData();
                float center = ldata[ldata.size()/2].dist;
                float right = ldata.front().dist;
```

```
            //sort laser data from small to large
            distances using a lambda function.
            std::sort( ldata.begin(), ldata.end(),
            [](RoboCompLaser::TData a, RoboCompLaser::TData b)
            { return a.dist < b.dist; });
            if((ldata.front().dist < 350 || center < 350))
                {
                        sp->differentialrobot_proxy->
                        setSpeedBase(0, -0.6);
                if(right > 450){
                    esquivar = true;
                }
                return rodearObj(center,right);
                }

            if(esquivar){
                return colocarse();

            }else
            {
                        sp->differentialrobot_proxy ->
                        setSpeedBase(500,0);
                return Node::Status::Running;
            }
        }
    }
    Status colocarse()
    {
        QPointF t;
```

```
        t = sp->getWaterDispenser();
        float angle = 0;
        //Paso el punto, de coord del mundo al robot
        QVec p = sp->innerModel->transform
        (sp->robotName.c_str(),
        QVec::vec3(t.x(),0,t.y()), "world");
        angle = qAtan2(p.x(),p.z()); // calculo angulo en rads
        if( fabs(angle) < 0.001)
        {
            sp->differentialrobot_proxy -> setSpeedBase(0,0);
            esquivar = false;
            return Node::Status::Running;
        }else
        {
            sp->differentialrobot_proxy -> setSpeedBase(0,angle);
            return Node::Status::Running;
        }
    }
    Status rodearObj(float c, float r)
    {
        if(r > 400 && r < 500 && c > 450)
        {
            sp->differentialrobot_proxy ->
            setSpeedBase(100,0);
            esquivar = true;
            return Node::Status::Running;
        }
        else
        {
```

```
            if(r < 400 || c < 350)
                    {
                                sp->differentialrobot_proxy ->
                                setSpeedBase(100,-0.6);
                return Node::Status::Running;
                    }
                    else if(r > 500)
                    {
                            sp->differentialrobot_proxy ->
                            setSpeedBase(100,0.25);
                return Node::Status::Running;
                    }
            }
        }

    private:
        SpecificWorker* sp;
        bool esquivar;
};
```

A.3.7. IniciarBeber

```
class ActionInitDrink : public BrainTree::Node
{
    public:
        ActionInitDrink()
        {
        }
        Status update() override
```

```
        {
            if(first_epoch)
            {
                reloj.restart();
                first_epoch = false;
                return Node::Status::Running;
            }
            else
            {
                if(reloj.elapsed() > 4000)
                {
                    first_epoch = true;
                    return Node::Status::Success;
                }
                else
                {
                    return Node::Status::Running;
                }

            }
        }
    private:
        bool first_epoch = true;
        QTime reloj;
};
```

A.3.8. IniciarAndar

```
class ActionInitWalk : public BrainTree::Node
```

```
{
    public:
        ActionInitWalk(SpecificWorker* x)
        {
        }
        Status update() override
        {
            sp->timeAction.restart();
            return Node::Status::Success;
        }
    private:
        SpecificWorker* sp;
};
```

A.3.9. AccionAndar

```
class ActionWalk : public BrainTree::Node
{
    public:
        ActionWalk(SpecificWorker* x)
        {
            this->sp = x;
        }
        Status update() override
        {
            int waitingTime = 15000; // 15 seconds sleeping
            if(sp->timeAction.elapsed() > waitingTime){
                sp->differentialrobot_proxy->setSpeedBase(0,0);
                return Node::Status::Success;
```

```
        }
        else
        {
            RoboCompLaser::TLaserData ldata = sp->laser_proxy->
            getLaserData();
            //sort laser data from small to large distances using
            a lambda function.
            std::sort( ldata.begin(), ldata.end(),
            [](RoboCompLaser::TData a, RoboCompLaser::TData b
            ){ return a.dist < b.dist; });

            if( ldata.front().dist < 300)
                {
                        sp->differentialrobot_proxy->
                        setSpeedBase(5, 0.6);
                        usleep(rand()%(1500000-100000 + 1)
                        + 100000);
                        // random wait between 1.5s and 0.1sec
                }
            else
            {
                sp->differentialrobot_proxy->
                setSpeedBase(700, 0);
            }
            return Node::Status::Running;
        }
    }
private:
    SpecificWorker* sp;
```

```
};
```

Apéndice B

Glosario de términos

En este capítulo se explicarán algunos términos utilizados durante la documentación, para así hacer mas sencillo el entendimiento del proyecto.

[1]Robótica : Puede definirse robótica como una ciencia que junta varias ramas tecnológicas, con el objetivo de diseñar máquinas robotizadas (robots) las cuales sean capaces de realizar tareas automatizadas o bien de simular el comportamiento humano o animal, pudiendo llegar a recrear inteligencia, como es el caso de este proyecto.

[2]Robot : Un robot podemos definirlo como una entidad autómata compuesta por una mecánica artificial y un sistema electromecánico, cuya función y tipo inteligencia dependerá de los programas informáticos que tendrá integrados.

[3]LearnBlock: es una herramienta de programación educativa creado por Robolab.

[4]Tick: señal que manda la raíz del árbol de comportamiento hacia los diferentes nodos para que se ejecuten.

[5]XML: es un lenguaje de marcado el cual se utiliza para crear los ficheros de configuración para los mapas que se quieren simular con el simulador.

[6]CDSL: es el lenguaje de definición de componentes utilizado para crear y definir

componentes en Robocomp.

[7]Nodo hoja: es el nodo de nivel más bajo del árbol a pesar de ser el que realiza las funciones.

[8]IDSL: es el lenguaje de definición de interfaces utilizado para crear y definir interfaces en Robocomp.

[9] LaTeX: sistema que permite la creación de documentos, orientado a la creación de documentos de una alta calidad tipográfica.

Bibliografía

[1] **Robocomp** https://github.com/robocomp/robocomp

[2] **Árboles de comportamiento** https://gamasutra.com/

[3] **LearnBlock** https://github.com/robocomp/LearnBlock

[4] **Librería Árboles de comportamiento** https://github.com/arvidsson/BrainTree

[5] **Radios MDEK1001** https://www.digikey.es/product-detail/es/MDEK1001/

[6] **Gazebo** http://gazebosim.org/

[7] **ROS** https://www.ros.org/

[9] **Getting started with LaTeX** https://www.maths.tcd.ie/

[10] **Overleaf** https://www.overleaf.com/