UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE INGENIERÍA ELÉCTRICA

INTERACCIÓN DE UN ROBOT MÓVIL CON UN OBJETO MÓVIL APLICADO AL FÚTBOL ROBÓTICO

MEMORIA PARA OPTAR AL TÍTULO DE
INGENIERO CIVIL ELÉCTRICO

WLADIMIR CELEDÓN AGUILERA

PROFESOR GUÍA:
JAVIER RUIZ DEL SOLAR SAN MARTÍN

MIEMBROS DE LA COMISIÓN:
JOSÉ MIGUEL YÁÑEZ ARANCIBIA.
HECTOR AGUSTO ALEGRÍA.

SANTIAGO DE CHILE
2014

RESUMEN DE LA MEMORIA
PARA OPTAR AL TÍTULO DE
INGENIERO CIVIL ELÉCTRICO
POR: WLADIMIR CELEDÓN AGUILERA
FECHA: 2014
PROFESOR GUÍA: JAVIER RUIZ DEL SOLAR SAN MARTÍN

INTERACCIÓN DE UN ROBOT MÓVIL CON UN OBJETO MÓVIL APLICADO AL FÚTBOL ROBÓTICO

El trabajo desarrollado tiene la finalidad de encontrar una nueva herramienta para desarrollar la toma de decisiones por parte de un robot que juega fútbol. En el transcurso de los avances en robótica, han sido muchos los intentos por encontrar una técnica de programación que permita emular el comportamiento humano en su mayor complejidad. Mientras más complejas sean las acciones, es más difícil generar un comportamiento, debido a que se necesita disponer de la plataforma que sea capaz de entregar las herramientas necesarias para ello.

El lenguaje de programación XABSL utiliza el concepto de la máquina de estados finita para desarrollar un comportamiento. Establece archivos donde se pueden crear los comportamientos y los símbolos por separados y anexarlos al software del robot. Además existe un editor para este lenguaje que muestra el grafo de estados y la jerarquía de estos, lo cual facilita el diseño. El desarrollo del comportamiento del jugador (Striker) comprende buscar la pelota, ir hacia la pelota, alinearse y patearla. Se diseña una máquina de estados que a su vez llama a otras para realizar cada una de las etapas mencionadas. Para utilizar el potencial de desarrollo de esta herramienta, se trabaja en el control de los sonares, seleccionando el modo de operación y la creación de los símbolos para realizar las transiciones en la máquina de estados para la evasión.

Con la mejora realizada en la adquisición de datos de los sonares y la calibración de estos para su funcionamiento, mejora la respuesta ante la aparición de obstáculos. Por otro lado el comportamiento de evasión, recibe como parámetro de entrada la posición de la pelota para ejecutar la transición de estados, lo cual es un aspecto importante que se presenta en XABSL. Patear la pelota constituye una máquina de estados con los tipos de golpes a realizar sin necesidad de establecer transiciones entre ellos. Además el jugador reduce las colisiones con las mejoras en el control de los sonares y la decisión de evadir considerando la información de la pelota.

Dedicado a mis padres Ely (mamá), Wlady (papá) y a mi hermano César, que tantas horas de estudios juntos *tú* en lo tuyo (Química y Farmacia) y *yo* en lo mío (Ingeniería Civil Eléctrica). Nunca llegamos atrasados a despertar la madrugada.

Al terminar esta etapa de mi vida, agradecer a quién me protegió por siempre ***María y José*** donde encontraré el éxito y satisfacciones en la nueva etapa que iniciaré.

Agradecimientos

Este trabajo fue parcialmente financiado por el proyecto Fondecyt 1090250.

Quiero agradecer a mis profesores guías que con ellos en el tercer piso del Laboratorio de Robótica de la Universidad de Chile trabajé en mi vocación, la Robótica. Anhelo fervientemente que sigan llegando jóvenes a ese Laboratorio para que con el tiempo seamos los más grandes en ese campo, que no tengo duda que así será.

A mis amigos: Juan Carlos (el mostro), Carlos Sebastian (taty) y Patricio (díscolo) a todos ellos y a muchos más gracias por las horas y años, que juntos se hizo fácil el sacrificio para obtener el Título (Ingeniero Civil Eléctrico) tan anhelado.

Agradecido de la mejor universidad, **La Universidad de Chile**, y de la mejor facultad de ingeniería, en Beauchef 850, Santiago de Chile. Gracias y a seguir trabajando, que a Chile hay que cuidar.

Tabla de contenido

Índice de tablas

Índice de figuras

Capítulo 1

Introducción

El presente trabajo se enmarca en el contexto del fútbol robótico, y se desarrolla dentro del equipo de robótica de la Universidad de Chile, del departamento de Ingeniería Civil Eléctrica. A continuación se presenta un nuevo enfoque para resolver la forma en que un robot puede interactuar con los objetos móviles, en particular con una pelota y otros robots. Entiéndase con interacción al hecho de que un robot pueda realizar acciones específicas como patear una pelota, aproximarse o evadir un objeto.

1.1. Motivación

En el fútbol robótico es importante la interacción que el robot puede tener con una pelota, la cual varía su posición en el campo de juego. Se pueden distinguir tres tipos de acciones importantes que se realizan durante un juego, las que son *localizar*, *aproximarse* y *patear*. Las tres acciones definen el éxito que tendrá el jugador para realizar un gol. El trabajo se centra en crear un comportamiento para el jugador dentro de un lenguaje de desarrollo de máquinas de estado finito. Este lenguaje es parte de la investigación para determinar cuál se ajusta más a las características que posee la librería UChRoboticsTeam y que consiga desarrollar un comportamiento más simple y ordenado del existente.

La interacción esperada corresponde a las decisiones que toma respecto de la información de la pelota y de la información de los obstáculos que detecte. La respuesta esperada ante un obstáculo es que el robot pueda evadirlo, para ello debe tener un esquema en el cual pueda caracterizar el objeto y determinar la dirección en donde dejaría de obstaculizar el paso. Por otra parte es importante también el desarrollo del comportamiento que se diseñe para generar la acción más rápida y sin posibles colisiones. En este contexto la solución que se proponga no es la definitiva ni la más óptima, dado que sigue siendo

un problema abierto, el desarrollo de los comportamientos para interactuar dentro de un partido de fútbol. El ideal de toda solución es que sea modular, simple en ejecución y que sea reactiva.

1.2. Estado del Arte

La plataforma de trabajo es el robot humanoide Nao (figura 1.1), el cual posee 21 grados de libertad y posee un sistema operativo basado en Linux. El funcionamiento de los sensores y actuadores del robot, así como la autonomía para ejecutar acciones son controladas por un programa que se ejecuta en su sistema operativo y está desarrollado en el lenguaje de programación C++.

Figura 1.1: Robot Nao utilizado en el fútbol robótico

El robot Nao utiliza una cámara que capta imágenes a color, para detectar la pelota, los arcos y las líneas que definen el campo de juego. Los dos sonares instalados en el torso permiten detectar la existencia de objetos alrededor de este y la distancia a la que se encuentran. En cada pie hay un sensor de presión que se utiliza para la caminata y un interruptor de dos estados (ON/OFF), que se utiliza para asignar el color del equipo e indicar el equipo que debe patear primero la pelota. Los sensores descritos entregan información del ambiente en el que se encuentra el robot (cámara y sonar) y del estado del robot(sensor de presión, encoders, giroscopio, acelerómetro) que son utilizados por el programa principal. Este programa está dividido en cinco módulos: **actuación**, **localización**, **comunicación**, **toma de decisiones** y **percepción**.

Basado en el diagrama de la figura 1.2, la *percepción* realiza la lectura de los sensores, esta información pasa a los *módulos de actuación*, *toma de decisiones*, *comunicación* y *localización*. La **actuación** realiza los movimientos del robot, los que pueden ser preestablecidos (patear la pelota) u otros dinámicos (caminata). Estas son ordenes que vienen de la **toma de decisiones** y es importante que sepa el estado actual de los actuadores involucrados en la locomoción. Toda tarea a realizar requiere como base que el robot

conozca su localización, por ejemplo en el fútbol robótico es importante determinar la pose para patear la pelota. Por último, el módulo de **comunicación** entrega información de las acciones que está ejecutando, el estado en el partido, su posición en el campo de juego y lo que percibe con los sensores al resto de los jugadores de su equipo, dado que es útil para tomar una decisión durante el juego, por ejemplo el robot que está más cerca es el que patea la pelota.

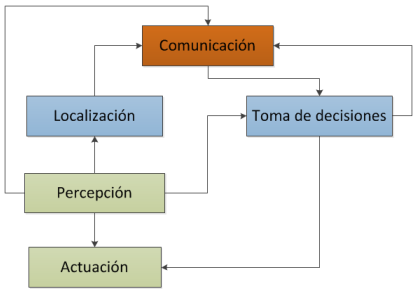


Figura 1.2: Esquema del programa principal del robot Nao.

1.3. Alcances y Objetivo General

Como objetivo de la memoria se pretende realizar el comportamiento de un jugador que sea capaz de llevar una pelota usando los pies hasta el arco contrario, evada obstáculos y realice un gol. Esta etapa corresponde al alto nivel dentro de las acciones de un robot. También dado que en el presente trabajo se requiere el uso de un sonar para detectar obstáculos, se desarrollará una sistema de control para los sonares del robot, cuya finalidad es mejorar las decisiones para caminar hacia la pelota o patearla.

El desarrollo que se pretende esta en el marco del control de la ejecución de las acciones. Por ello es importante que la estrategia a utilizar no se enmarque en una aplicación particular, es decir, considerar otras posibles situaciones durante el juego. Por ejemplo evadir varios obstáculos que aparezcan en todo momento, patear hacia el lado cuando no sea posible patear hacia el arco, escoger otra pose para patear hacia el arco en función de la orientación hacia este, etc.

1.4. Objetivos Específicos

Desarrollar el tema de la llega del robot involucra varias etapas de trabajo las cuales deben primero tomar en cuenta cual es el estado en que se encuentra actualmente el

funcionamiento del robot y la solución que se usa. Por lo que los objetivos específicos son:

- Analizar y comparar dos lenguajes XABSL y SMACH para el desarrollo de máquinas de estado finito.
- Definir el lenguaje a utilizar conforme a las necesidades que impone el fútbol robótico y la plataforma de desarrollo que ya existe, de acuerdo a la información dada por el punto anterior.
- Proponer un nuevo esquema del comportamiento del robot en base al lenguaje seleccionado.
- Implementar un sistema de control del sonar, para integrar las mediciones a los algoritmos previamente existentes para procesar la información.
- Evaluar la detección de obstáculos y el módulo de evasión.

1.5. Estructura de la memoria

La estructura utilizada en este documento para exponer el trabajo realizado es la siguiente:

- **Capítulo 1. Introducción:** Corresponde a la descripción del tema, la motivación de éste y los alcances y objetivos del trabajo realizado.
- **Capítulo 2. Revisión Bibliográfica:** Corresponde a la revisión bibliográfica o antecedentes. En este capítulo se explican los conceptos necesarios para la comprensión y contextualización del trabajo de investigación realizado.
- **Capítulo 3. Librerías para el Desarrollo de Máquinas de Estados Finita:** Corresponde al trabajo de investigación sobre los lenguajes para el desarrollo de máquinas de estados finitos.
- **Capítulo 4. Diseño e Implementación de Control de Sonar:** Se realiza la construcción de un control por software del funcionamiento de los sonares, cuya información acerca de los obstáculos sea usada por una grilla de obstáculos ya existente.
- **Capítulo 5. Diseño General de la Máquina de Estados:** Se describe el comportamiento realizado para que un robot pueda jugar fútbol.

- **Capítulo 6. Discusión de resultados, simulaciones y pruebas en el robot:** Se exponen los resultados y se discuten los alcances de la solución propuesta.
- **Capítulo 7. Conclusiones:** Se enumeran las conclusiones del trabajo realizado y se proponen trabajos a realizar en el futuro.

Capítulo 2

Revisión Bibliográfica

2.1. Máquina de Estados Finita

Este concepto se suele referir al término **autómata finito** [5], el cual es un modelo matemático que se usa para diseñar programas de computadora o para el diseño de circuitos secuenciales. El modelo corresponde a la representación de un sistema que posee entradas y salidas discretas, por otra parte el sistema puede estar en alguna de las configuraciones finitas que posee, lo cual se conoce como *estado*. El estado de un sistema resume la información concerniente de las entradas pasadas necesarias para determinar el comportamiento de éste sobre las subsecuentes entradas. La figura 2.1 se presenta un ejemplo del diagrama de un autómata básico que contiene un estado inicial etiquetado como q_0.

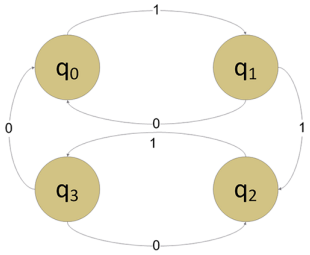


Figura 2.1: Diagrama de transición de estados de un autómata.

Un *autómata finito* se define como un conjunto de estados finitos y un conjunto de transiciones desde un estado a otro que ocurre dado un símbolo de entrada elegido desde un alfabeto Σ. Para cada símbolo de entrada hay una transición de salida de cada estado, en ciertas ocasiones esa salida puede llevar al mismo estado. El estado inicial denotado

por q_0 es en el cual el autómata comienza. Por otra parte puede existir un estado final. La representación de un autómata se realiza por medio de grafo, que establece en sus vértices los respectivos estados del sistema y los arcos representan las entradas que generan las transiciones entre los estados.

Matemáticamente un autómata finito es una 5-tupla $(Q, \Sigma, \delta, q_0, F)$, donde Q es un conjunto finito de estados, Σ es el alfabeto finito de entradas, $q_0 \in Q$ es el estado inicial, $F \subseteq Q$ es el conjunto final de estados y δ es la función de transición que genera el mapa de $Q \times \Sigma aQ$. Los símbolos de los autómatas son comunes en **alfabetos** y **strings** en su mayoría, los cuales son:

Símbolo: En muchos lenguajes de programación corresponde a una variable que se utiliza para crear las transiciones de estado.

Alfabeto: Conjunto finito de símbolos. El alfabeto se simboliza Σ, el cual es el conjunto de letras en un alfabeto.

Clausura de Kleene: Lenguaje que puede considerarse un subconjunto de todas las posibles palabras. Y a su vez este conjunto de palabras, puede ser, considerado como el conjunto de todas las posibles concatenaciones de cadenas. En inglés a este conjunto de cadenas se les denomina *free monoid*, y se indica como Σ^* y el superíndice se conoce como estrella de Kleene.

2.1.1. Tipos de Máquinas de Estado

Existen cuatro tipos de autómatas finitos:

Autómata finito determinista (AFD)

Pueden o no tener una transición por cada símbolo del alfabeto.

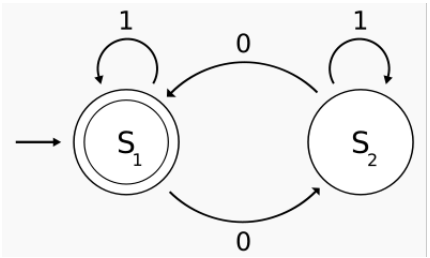


Figura 2.2: Ejemplo de autómata determinista.

Autómata finito no determinista (AFND)

El de este tipo de autómata pueden, o no, tener una o más transiciones por cada símbolo del alfabeto. Acepta una palabra si existe al menos un camino desde el estado q_0 a una estado final F etiquetado con la palabra de entrada. La palabra es rechazada si una transición no está definida, por lo tanto el autómata no puede saber como continuar leyendo la entrada.

Autómata finito no determinista con transiciones ε (AFND-ε)

Capaz de alcanzar más estados leyendo un símbolo, permite alcanzarlos sin leer ningún símbolo. Si un estado tiene transiciones etiquetadas con ε, entonces el AFND puede encontrarse en cualquiera de los estados alcanzables por las transiciones ε, directamente o a través de otros estados con transiciones ε. El conjunto de estados que pueden ser alcanzados a través de este método desde un estado q, se denomina la clausura ε de q. Se puede construir un AFD que acepte el mismo lenguaje que el dado por un AFND (por lo tanto estos tipos de autómatas pueden aceptar los mismos lenguajes).

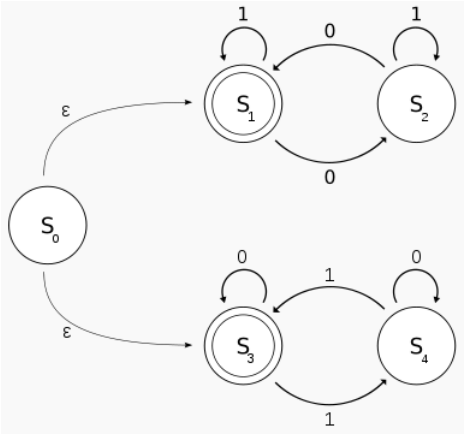


Figura 2.3: Ejemplo de autómata con transiciones ε.

Autómata con pila

Máquinas idénticas a los AFD (o AFI), exceptuando de que disponen de una memoria adicional, haciendo uso de una pila. La función de transición δ ahora dependerá también de los símbolos que se encuentren al principio de la pila. Esta función determinará como cambia la pila en cada transición. Este tipo de autómatas aceptan los lenguajes independientes del contexto.

2.2. Fútbol Robótico

El fútbol robótico es una competencia de 5 robots por lado que juegan un partido de fútbol bajo ciertas condiciones que son:

- Se dispone de una cancha de 9 metros de largo por 6 de ancho. Se utiliza una carpeta verde en la cual se trazan líneas de 50 milímetros de espesor a 70 centímetros de los bordes de la cancha, para trazar el área de juego.
- Hay dos arcos amarillos, que miden 1.5 metros de ancho por 80 centímetros de alto y una profundidad de 50 centímetros.
- Cada equipo tiene un arquero que es el único que puede estar dentro del área. Lleva el número 1 puesto en la espalda.
- Se utiliza una pelota naranja de 65 milímetros de diámetro.
- Todos los equipos consisten en robots humanoides Nao, fabricado por la empresa Aldebaran Robotics. No deben presentar modificaciones en cuanto al hardware.
- Cada equipo se identifica con un color, están permitidos el azul y rojo. Para ello se utiliza una polera que va en el torso del robot, junto con un número de identificación.
- El número uno siempre corresponde al robot arquero.
- Poseen conexión wifi para que los robots se comuniquen con el **GameController** que es un programa que contiene la información del estado del juego, por ejemplo el marcador, los jugadores penalizados y el tiempo de juego.

La figura 2.4 muestra un bosquejo de la cancha de fútbol que se utiliza en la RoboCup.

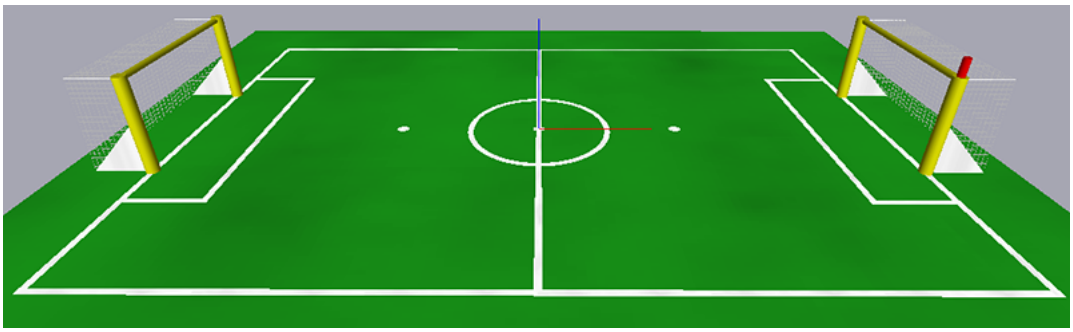


Figura 2.4: Bosquejo del campo de juego.

Los partidos constan de dos tiempos de 10 minutos cada uno. En caso de que exista empate se realizan penales. La figura 2.5 representa los estados del juego, es decir, las etapas que se realizan en el primer y segundo tiempo.

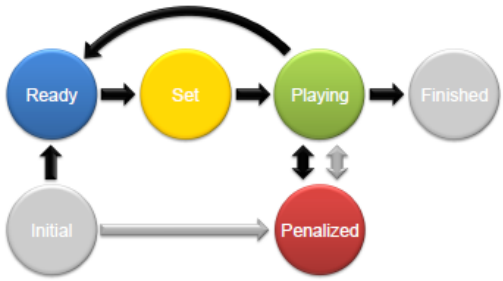


Figura 2.5: Estados del juego para el fútbol robótico.

Initial: Este estado mantiene al robot en pie sin realizar ninguna acción. El robot está en espera, en el borde de la cancha.

Ready: El robot se localiza en el campo de juego y posteriormente se posiciona en el puesto que se le es asignado. Tiene una duración de 45 segundos y el botón del pecho está en color azul.

Set: Después de que se están posicionados, los robot visualizan la pelota y el arco enemigo. El botón está de color amarillo.

Play: En este estado se da inicio al juego, su duración es de 10 minutos por lado. El botón del pecho está en color verde.

Finish: Después de cumplidos los 10 minutos, los robots se detienen y quedan a la espera de ser retirados de la cancha.

Penalized: En este estado el robot es sacado del campo de juego y puesto a un costado sin mirar donde esta la pelota. La duración es de 45 segundos y el botón del pecho está en rojo.

El posicionamiento es importante dentro del partido, en caso que un robot no se posicione bien, este es puesto cerca del área del arco a modo de penalización para el equipo. En la figura 2.6, el equipo en rojo corresponde al que realiza el *kick off* o patada inicial, si no se ejecuta dentro de los 10 primeros segundos de comenzado el partido, el equipo contrario puede patear la pelota.

Un robot puede ser penalizado por las siguientes razones:

1. Colisionar con otro robot por más de 3 segundos.
2. Dejar el campo de juego.
3. Ingresar al área del arquero de su mismo equipo.
4. Retener la pelota en los pies por mas de 3 segundos.

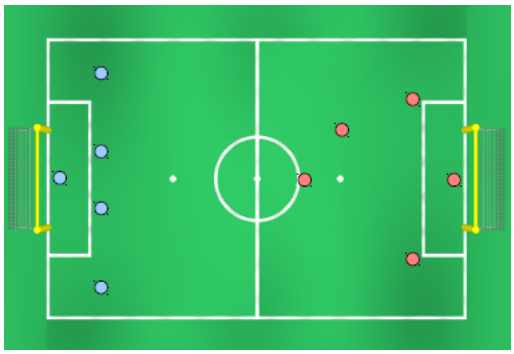


Figura 2.6: Ejemplo del posicionamiento de los equipos.

2.3. Sonar

El sonar (acrónimo de Sound Navigation And Ranging, *navegación por sonido*) es una técnica que usa la propagación del sonido para navegar, comunicarse o detectar objetos. El sonar puede usarse como medio de localización acústica, funcionando de forma similar al radar, con la diferencia de que en lugar de emitir señales electromagnéticas, se emplean pulsos sonoros. De hecho, la localización acústica se usó en aire antes que el GPS, siendo aún de aplicación el SODAR (la exploración vertical aérea con sonar) para la investigación atmosférica. La señal acústica puede ser generada por piezoelectricidad o por magnetostricción.

El término *sonar* se usa también para aludir al equipo empleado para generar y recibir el sonido de carácter infrasonoro. Las frecuencias usadas en los sistemas de sonar pueden ser intrasonica (bajo los 20 Hz) o ultrasónica (sobre los 20000 Hz). Existen otros sonares que no abarcan el espectro del oído humano, (cazaminas); pueden comprender varios espectros de alta frecuencia, 80 kHz o 350 kHz, por ejemplo. Ganan en precisión a la hora de determinar el objeto, pero pierden en alcance.

El sonar activo usa un emisor de sonido y un receptor. Cuando los dos están en el mismo lugar se habla de funcionamiento monoestático. Cuando el emisor y el receptor están separados, de funcionamiento biestático. Cuando se usan más emisores o receptores espacialmente separados, se refiere a funcionamiento multiestático. La mayoría de los equipos de sonar son monoestático, usándose la misma matriz para emisión y recepción, aunque cuando la plataforma está en movimiento puede ser necesario considerar que esta disposición funciona biestáticamente. Los campos de sonoboyas activas pueden funcionar multiestáticamente. El sonar activo crea un pulso de sonido, llamado a menudo *ping*, y entonces oyen la reflexión o eco del mismo. Este pulso de sonido suele crearse mediante electrónica, usando un generador de señal, un amplificador de potencia y un transductor o matriz electroacústica y posiblemente un conformador de haces.

Para calcular la distancia a un objeto se mide el tiempo desde la emisión del pulso a la recepción de su eco y se convierte a una longitud conociendo la velocidad del sonido. Para medir el rumbo se usan varios hidrófonos, midiendo en conjunto el tiempo de llegada relativo a cada uno, o bien una matriz de hidrófonos, midiendo la amplitud relativa de los haces formados mediante un proceso llamado conformación de haz. El uso de una matriz reduce la respuesta espacial de forma que para lograr una amplia cobertura se emplean sistemas multihaces. La señal del blanco (si existe) junto con el ruido se somete entonces a un procesado de señal, que para los equipos simples puede ser sólo una medida de la potencia. Se presenta entonces el resultado a algún tipo de dispositivo de decisión que califica la salida como señal o ruido. Este dispositivo puede ser un operador con auriculares o una pantalla, en los equipos más sofisticados es un software específico. Pueden realizarse operaciones adicionales para clasificar el blanco y localizarlo, así como para medir su velocidad.

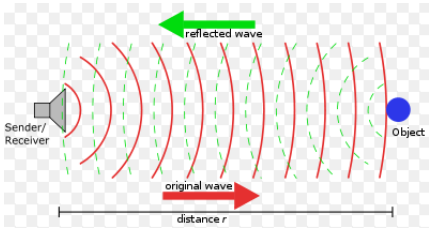


Figura 2.7: Esquema de funcionamiento del Sonar.

En la figura 2.7 se muestra el concepto detrás del funcionamiento del sonar. Una onda que tiene una frecuencia f y velocidad v es emitida. Cuando ésta regresa se obtiene la siguiente ecuación

$$t = \frac{2r}{v}$$

Midiendo el tiempo de recepción se puede determinar la distancia al objeto.

Capítulo 3

Librerías para el Desarrollo de Máquinas de Estados Finita

3.1. XABSL

3.1.1. Descripción del Lenguaje

Extensible Agent Behavior Specification Language [2] es un lenguaje que describe el comportamiento para robots autónomos basado en máquinas de estados finita que se organizan de forma jerárquica, independiente de la plataforma y arquitectura en la cual se utilice. Este consiste de los siguientes archivos:

- Archivos *option* en el cual se define una máquina de estados finita, cada archivo tiene la extensión *".xabsl"*.
- Archivo de comportamiento básico que contiene todos los comportamientos básicos y sus parámetros. Estos son referenciados desde los estados.
- Archivos de símbolos, cuya extensión es *".xabsl"*. Un símbolo en XABSL es una representación de una variable o constante que corresponde a una entrada para algún estado dentro de la máquina de estados finita definida en el archivo *option*.
- **agents.xabsl**, este incluye la lista de todos los archivos *option* definidos para un comportamiento.

Los símbolos se definen en documentos de instancia XABSL para formalizar la interacción con el entorno del software, se clasifican como entrada o salida, dependiendo si se lee un dato desde el software o se carga un dato hacia este. La interacción significa

acceder a las variables y funciones de entrada, y para las funciones de salida (por ejemplo, para establecer solicitudes a otros módulos en el procesamiento de la información). Para cada variable o función que se quiera utilizar en ciertas condiciones, se debe definir un símbolo. Esto hace que XABSL sea independiente del entorno específico del software y la plataforma. El programador puede decidir si se debe expresar condiciones complejas en XABSL combinando diferentes símbolos de entrada con operadores lógicos y/o numéricos o mediante la implementación de la condición como un función de análisis en C++ y su referencia a esta función a través de un símbolo de entrada.

Option es un concepto dentro de XABSL que corresponde a una máquina de estados finita, un conjunto de *option* se conoce como *option graph*, el cual es un grafo de la relación que existe entre todos los estados que se utilicen en un autómata finito. Las condiciones que deben cumplirse para entrar a un estado o salir de este, dependen de los *símbolos de entrada*. Por otro lado un *símbolo de salida* comunica una acción que debe ejecutar el software del robot. Durante el funcionamiento del robot, se cargan los datos procesados que provienen de los sensores en los símbolos de entrada que se hayan creado, y dentro del estado se actualizan los símbolos de salida que son leídos para ejecutar la acción que en ello se especifique.

XABSL tiene dos formas de controlar las acciones en un robot: comportamientos básicos y símbolos de salida. Un **comportamiento básico** es una acción parametrizada que puede ser activada en el último estado, aquel que no tiene más estados subsiguientes y no es parte de XABSL. Un **símbolo de salida** puede ser un valor lógico, decimal o enumerado. Cada estado puede modificar un conjunto o todos los símbolos de salida; un estado puede modificar un símbolo de salida que haya sido previamente modificado en un estado anterior. Los símbolos de salida se pueden utilizar para controlar procesos de percepción o actuadores adicionales.

Organización Jerárquica

Un *option* simple se conforma de un conjunto de estados y solamente un estado puede estar marcado como el *estado inicial*. Cada estado define una acción que significa utilizar símbolos de salida o comportamientos básicos definidos en el software del robot. La figura 3.1 es un ejemplo de una máquina de estados finita en XABSL. Se define como estado principal a **initial_Position** lo cual se expresa en el grafo por las dos líneas horizontales dentro del círculo. Todos los estados son definidos como círculos en el grafo. En este caso se tiene un estado inicial y uno final, por lo cual no es cíclico. Un estado dentro de un *option* puede ser otro *option* o una llamada y así sucesivamente, además varios estados pueden estar conectados a un mismo estado o comportamiento básico.

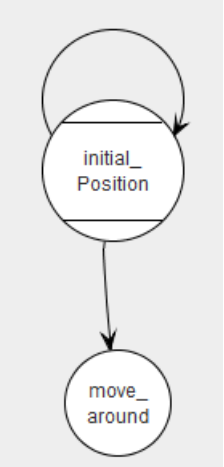


Figura 3.1: Ejemplo de un *option* básico que se define como una máquina de estados finita.

En el ejemplo de la figura 3.2 se define un *option* principal que tiene como estado inicial a **left**, el cual ejecuta el *option* **look_left** que viene a ser un *option* simple como el de la figura 3.1. Análogamente se tiene la misma situación para el estado **right** que ejecuta el *option* **look_right**. En este caso se define un *option* de bajo nivel, lo que representa más bien un comportamiento más básico, como por ejemplo mirar hacia la derecha o izquierda. Entonces cuando se desarrolla un comportamiento que permita mirar hacia la derecha e izquierda, ya sea alternadamente se puede utilizar las máquinas de estado que definen cada comportamiento anidadas de manera que una máquina de estados en un nivel más alto se encargue de ejecutarlos cuando entre al estado respectivo en el cual se llame a cada uno para su ejecución.

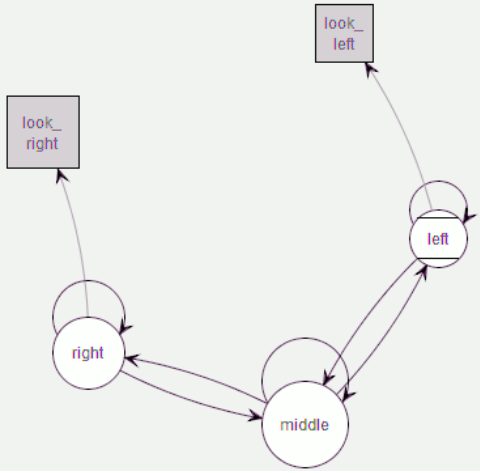


Figura 3.2: Ejemplo de un *option* que contiene otros *option*.

El *option graph* muestra la jerarquía entre las máquinas de estados. Para el ejemplo de la figura 3.3 se definen tres comportamientos: `look_left`, `look_right` y `look_left_and_right`. Donde el *option* principal es **look_left_and_right** y por lo tanto, para ejecutar el comportamiento de mirar hacia la derecha, se debe ejecutar la siguiente ruta de activación: **left**→**middle**→**right**→*look_right*. Entonces pueden existir máquinas de estados de bajo nivel que tienen un propósito simple de ejecutar órdenes básicas (mover la cabeza), los cuales son utilizadas dentro de otra máquina de estados para definir un comportamiento más complejo, por ende se crea una jerarquía en la cual los *option* que se encuentran en el nivel más bajo se ejecutan dependiendo de las transiciones de estados que sucedan en el *option* que los precede. Para una acción como caminar hacia un objeto (pelota), los comportamientos que se necesiten pueden ser integrados en varios niveles de *option*, no hay limitación en la complejidad de la máquina de estados.

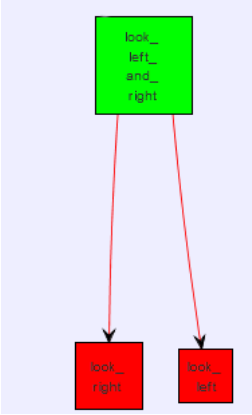


Figura 3.3: Ejemplo de la jerarquía de los *option*.

Activar un *Option* y Ejecutar una Acción

Durante la ejecución de un comportamiento, se puede definir una *ruta de activación* que se inicia en el estado inicial del *option* raíz. Para definir las transiciones entre estados, cada estado tiene un **árbol de decisiones**, el cual selecciona una transición a otro para el mismo estado. Las decisiones se toman en base a símbolos de entrada que pueden ser información de los sensores, localización o mensajes de otros robots. El subsiguiente *option* asociado con el estado actual en ejecución, puede no haber estado activo en el paso anterior, el estado actual de este subsiguiente *option* es establecido en el estado inicial. Entonces el árbol de decisiones de este nuevo estado dará paso a un siguiente estado que es anexado en la *ruta de activación*, este proceso se repite hasta llegar al estado de nivel más bajo que ejecuta una acción fuera de XABSL.

Compilador

La clase xabsl:Engine es una plataforma y aplicación independiente escrita en C++. Esto permite implementar XABSL en cualquier plataforma robótica. Para ejecutar XABSL en un software específico, solamente dos clases deben ser derivadas desde la clase abstracta. Esta clase analiza y ejecuta el código intermedio de extensión *.dat*. Este vincula los símbolos creados con las funciones y variables pertenecientes a la plataforma, por lo tanto para cada símbolo utilizado una entidad en el software del robot debe ser registrado en XabslEngine. Los comportamientos básicos son escritos en C++, los que deben ser derivados desde una clase base común y registrados en el XabslEngine.

Desde XABSL se pueden generar 4 documentos mediante un compilador escrito en el lenguaje Ruby[1]:

1. Un *código intermedio* el cual es ejecutado por la clase **Xabsl:Engine**.
2. Símbolos de depuración que contiene los nombres de todos los *option*, estados, comportamientos básicos y símbolos que son útiles para implementar herramientas de depuración de código.
3. Un *keyword list* para usar en editores.
4. Un documentos en HTML que contiene un gráfico SVG para cada máquina de estados.

3.1.2. Sintaxis y Visualización Gráfica de Máquinas de Estados

Para crear un comportamiento en XABSL se debe definir el archivo **agents.xabsl**, archivos para los símbolos y máquinas de estado finita.

Agents

Es el documento principal de un comportamiento específico, este incluye todos los *option* necesarios. En un comportamiento de XABSL, el grafo de máquinas de estado no necesita estar completamente conectado. Por lo tanto, no es posible determinar el *option*

[1]Ruby es un lenguaje de programación interpretado, reflexivo y orientado a objetos, creado por el programador japonés Yukihiro Matsumoto, y presentado públicamente en 1995. Combina una sintaxis inspirada en Python y Perl con características de programación orientada a objetos similares a Smalltalk. Comparte también funcionalidad con otros lenguajes de programación como Lisp, Lua, Dylan y CLU. Ruby es distribuido bajo una licencia de software libre.

raíz del grafo. Por ejemplo un sub-grafo que está extendido por un *option* y todos sus *option* posteriores y comportamientos básicos puede ser declarado como un agente. Por lo tanto, un agent define un punto de partida en el grafo. A continuación se presenta la sintaxis para declarar una máquina de estados en el archivo **agents.xabsl**.

```
include "<include file>";
agent <id>("<agent-title>", <root-option>);
```

Se incluyen los archivos de todos los **option** a compilar, para ello se usa la sentencia `include` y posteriormente se ingresa la ruta donde está el archivo. Después el `<id>` indica el agent que debe seleccionar XABSL:Engine, se coloca un título para la documentación y luego el option que será la raíz. El siguiente ejemplo muestra un archivo donde se definen los símbolos de entrada y salida, y dos archivos que definen las máquinas de estado. Se ingresa como título **Soccer** y el **option** raíz es **body_control**.

```
include "Symbols/ball_symbols.xabsl";
include "Options/BodyControl/body_control.xabsl";
include "Options/BodyControl/States/Play/player.xabsl";

agent soccer("Soccer", body_control);
```

Cada vez que se cree un nuevo **option**, es necesario agregarlo en el **agents.xabsl** para que sea llamado desde cualquier otro *option* que esté indexado. De lo contrario el compilador no generará el archivo intermedio e indicará que existe una llamada a un archivo que no está definido.

Option

Corresponde al archivo en el cual se construyen las máquinas de estado finita y por ende es la base para la construcción del comportamiento general. Se define la sintaxis general para declarar una máquina de estados simple:

```
include "<include file>";
[<doc comment>] option <name>
{
  { <parameter> }
  [ <common decision tree> ]
  <state>
  { <state> }
}
```

<name> corresponde al nombre del archivo, es decir, si el *option* a crear es ”soccer”, entonces el archivo debe ser **soccer.xabsl**. Por otra parte se pueden incluir símbolos, comportamientos básicos u otro *option* haciendo referencia al archivo en <include file>. La máquina de estados que se diseñe puede aceptar datos de entrada que se pasan en la llamada a ésta. Por lo cual se definen los parámetros, variables que pueden utilizar los estados dentro de una máquina de estados. Este se define de la misma manera como un símbolo de entrada, el nombre comienza con el símbolo @ para hacer la distinción con un símbolo de entrada normal. La sintaxis para definir un parámetro dentro de un *option* es:

```
float @<name> | bool @<name> | enum <enumeration> @<name>;
```

<name> es el nombre de la variable y float, bool y enumeration es el tipo de parámetro. Un ejemplo simple es definir un parámetro de tipo *float* y utilizarlo en un *option*:

```
option walk{
    float @x;
    [<common decision tree>]
    <state>
    { <state>}
```

States

Representa un estado simple que compone a una máquina de estados finita. Este se define como:

```
[initial] [target] state <name> {
  decision
  {
    [ else ] <decision tree>
  }
  action
  {
    { <action definition> }
  }
}
```

Cuando se crea un estado, se puede catalogar como initial, target o estado simple, en cuyo caso solo se escribe state <name>. Cuando se crea un comportamiento, siempre existe un estado *inicial*, al cual se entra cada vez que se llame al comportamiento;

estados simples y como opcional un estado *target*. Dentro de *decision* se escribe el árbol de decisiones para las transiciones a otro estado. Finalmente en *action* se describen las acciones que se pueden realizar, las cuales pueden ser llamar a otro **option** o asignarle un valor a un símbolo de salida. Las transiciones de un estado a otro se expresan de la siguiente forma:

```
if(proposicion_1)
  goto Next_State;
else if(proposicion_2)
  goto Other_State;
else
  stay;
```

Symbols

Los símbolos se definen en un archivo aparte, la sintaxis es la siguiente:

```
include "<include file>";
namespace <id>("<title>")
{
    <enumeration>     |
    <input symbol>    |
    <output symbol>   |
    <internal symbol> |
    <constant>
}
```

El **id** corresponde al nombre del archivo. También se pueden incluir archivos anexos. Hay diferentes definiciones de símbolos a usar. Un símbolo puede ser de entrada, salida o interno, este último implica que no se debe generar una instancia en el software del robot. Estos se definen como:

```
<type symbol> [input | output | internal] <name symbol>
```

`<type symbol>` indica el tipo de símbolo a usar. La siguiente sintaxis define el símbolo **enumeration**:

```
enum <name> [internal]
{
  <enum-element>,
```

```
    <enum-element>
  };
```

Se define un nombre cualquiera **name**, los elementos van seguidos de una coma a excepción del último elemento. Para usar el símbolo se debe crear un símbolo tipo *enumeration*, la sintaxis muestra el tipo y el nombre que se le asigne.

```
  enum <name> [input | output | internal] <name_symbol>;
```

Para el caso de un símbolo de tipo `float` su declaración es:

```
  float <name> [[<range>]] ["<measure>"];
```

En este caso se puede indicar un intervalo al que pertenece el símbolo `<name>` mediante `[[<range>]]` y `<measure>` es la unidad de medición, que puede ser milímetro, centímetro, etc. También se puede definir una constante, la cual siempre es de tipo `float`, su declaración es:

```
  float const <name> = <value> ["<measure>"];
```

Para los símbolos de tipo *bool*, se admiten las siguientes expresiones:

```
<boolean expression> ::=
    ( <boolean expression> )
    !<boolean expression>
    <boolean expression> && <boolean expression>
    <boolean expression> || <boolean expression>
    <enumerated expression> == <enumerated expression>
    <decimal expression> == <decimal expression>
    <enumerated expression> != <enumerated expression>
    <decimal expression> != <decimal expression>
    <decimal expression> < <decimal expression>
    <decimal expression> <= <decimal expression>
    <decimal expression> > <decimal expression>
    <decimal expression> >= <decimal expression>
    true
    false
    action_done
```

action_done se convierte en *true* cuando el estado actual ejecuta un *option* y el estado activo dentro de éste está etiquetado como *target state*.

Para los símbolos de tipo decimal, se admiten las siguientes expresiones:

```
<decimal expression> ::=
  ( <decimal expression> )
  <decimal expression> + <decimal expression>
  <decimal expression> - <decimal expression>
  <decimal expression> * <decimal expression>
  <decimal expression> / <decimal expression>
  <decimal expression> % <decimal expression>
  <boolean expression> ? <decimal> : <decimal>
  <decimal input symbol> [ <parameter list> ]
  state_time
  option_time
```

- *state_time* indica la duración en segundos para la cual el estado actual de una máquina de estados está activa.
- *option_time* expresa el tiempo en segundos para la máquina de estados activa.

Editor XABSL

Para desarrollar los comportamientos en este lenguaje, se dispone de un editor implementado en el lenguaje Java, el cual funciona en cualquier plataforma que soporte Java(Windows, Linux, Mac). Este programa provee un visor gráfico de la máquina de estados que se construye en tiempo real y el grafo de todos los *option* que se anexan en el **agents.xabsl**. Permite ir viendo el código de cada estado o máquina de estados manipulando la representación gráfica en el visor.

El editor destaca las palabras claves del lenguaje dentro de la sintaxis. Para actualizar la pantalla, el compilador XABSL se invoca directamente desde el editor. El compilador genera representaciones de tipo XML que se utilizan para generar los gráficos. Errores notificados por el compilador también se muestran directamente en el editor.

El editor de XABSL proporciona una manera conveniente de hacerlo. Especialmente para usuarios sin experiencia, es útil tener una pantalla gráfica que permite visualizar de inmediato los cambios en la máquina de estados. La figura 3.4 muestra el esquema de una máquina de estados simple junto con el código, notar que las palabras claves como **option**, **state**, **initial state**, **decision**, **action**, **goto**, **if - else** y **stay** están destacadas. También se destacan los comportamientos básicos que se ejecuten dentro de un estado o funciones simples que se creen.

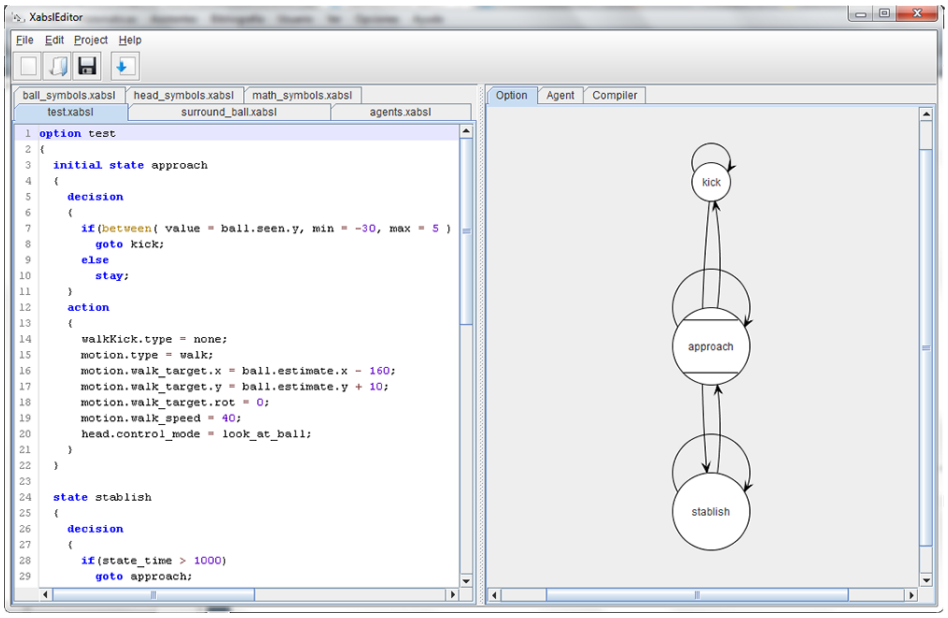


Figura 3.4: Editor de XABSL en Windows.

3.1.3. Formas de Depuración del Código

La clase XabslEngine incluye herramientas para depurar el código, estas son utilizadas por el simulador SimRobot[2], cuya plataforma de desarrollo es C++. Cuando se ejecuta el simulador, se activa un visor que muestra la información de los símbolos y el *option activation graph*. Para ello se necesita compilar el código para generar el archivo intermedio *.dat*.

En la figura 3.5 se muestra como ejemplo la ejecución del comportamiento **start_soccer**. En la parte superior se define el título que se escribió en el agents.xabsl, después se define la solicitud de movimiento del robot, que para este ejemplo es estar parado, posteriormente se puede ver los valores que tienen los símbolos de entrada y salida durante la ejecución del comportamiento. Y por último el *option activation graph* muestra el estado actual junto con los estados predecesores, el tiempo que aparece a la derecha corresponde al *state_time* y el *option_time*. Mientras más niveles tenga el comportamiento, mayor será la profundidad del *option activation graph* que se muestra en el visor. Además el visor sólo muestra los símbolos de entrada y salida, los internos no se pueden ver dado que no tienen una representación dentro de la plataforma de desarrollo del software del robot.

[2]Programa que visualiza gráficamente la ejecución de un comportamiento en XABSL, está desarrollado en C++.

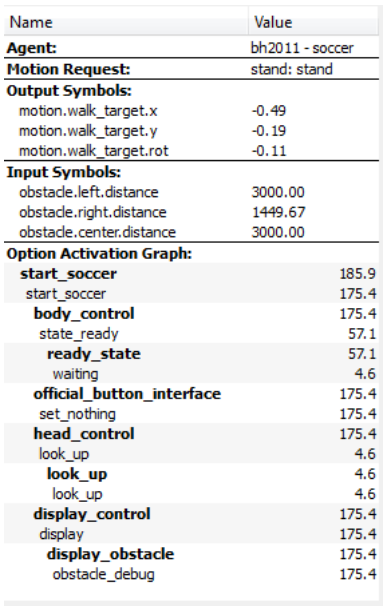


Figura 3.5: Programa en el simulador para visualizar el funcionamiento de un comportamiento.

3.1.4. Ventajas y Desventajas Técnicas

La sintaxis y la lógica de programación hacen de XABSL un lenguaje de rápido aprendizaje. La clase **XabslEngine** permite que se utilice en cualquier sistema basado en C++, lo que demuestra que es independiente del programa del robot. EL análisis de las ventajas o desventajas que tenga su utilización están relacionadas con las prestaciones y las condiciones que se deben cumplir para ser implementado. Esto significa que el entorno de desarrollo debe ser adaptado para realizar la comunicación de los módulos de actuación, cognición y localización.

Ventajas

- Visualización en tiempo real de la máquina de estados.
- Capacidad para generar comportamientos complejos.
- Visualizar símbolos en ejecución.
- Compilación externa a la plataforma.
- Editor interactivo.

Desventajas

- Generar símbolos de entrada y salida, se deben crear las funciones o variables en el software del robot, y asociarlos con los símbolos creados dentro de XABSL, no es el caso para los símbolos internos.
- Instalación relativamente más compleja, dado que se debe integrar Ruby para compilar los comportamientos creados.

3.2. SMACH

3.2.1. Descripción del Lenguaje

SMACH es una arquitectura para la rápida creación de comportamientos complejos para un robot. En esencia, es una biblioteca de Python[3] independiente de ROS[4] para construir máquinas de estados jerárquicas. Es una nueva biblioteca que se aprovecha de conceptos muy viejos con el fin de crear rápidamente comportamientos robustos para el robot mediante un código modular.

SMACH es útil cuando se desea que un robot ejecute un rutina compleja, donde todos los posibles estados y las transiciones de estado se puedan describir de forma explícita. Se puede construir una máquina de estados finita utilizando SMACH, pero este puede hacer mucho más. SMACH es una biblioteca para la ejecución y la coordinación a nivel de tareas y proporciona varios tipos de contenedores para un estado. Un contenedor es una máquina de estados finita, pero éste puede estar contenido dentro de otro contenedor. Hay varios tipos de contenedores que son proporcionados por la biblioteca SMACH. Diferentes contenedores proporcionan diferentes semánticas de ejecución, pero todos ellos pueden ser tratados como estados en otros contenedores. Los contenedores pueden tener sus propias formas de especificar las transiciones de los estados contenidos, ya que la *transición* significa cosas distintas en diferentes contextos.

Para todos los contenedores, la interfaz para los estados es definida vía **state outcomes**. Las salidas potenciales de un estado son una propiedad de éste y deben ser declaradas antes

[3]Python es un lenguaje de programación multiparadigma, ya que soporta orientación a objetos, programación imperativa y, en menor medida, programación funcional. Es un lenguaje interpretado, usa tipado dinámico y es multiplataforma.

[4]El Sistema Operativo Robot (ROS) es un marco flexible para la escritura de software para robots. Se trata de una colección de herramientas y bibliotecas que tienen como objetivo simplificar la tarea de crear un comportamiento complejo y robusto en una amplia variedad de plataformas.

de ser ejecutado. Los *state outcomes* pueden causar diferentes salidas en diferentes tipos de contenedores. Pero lo que sucede después de que una salida es emitida es irrelevante desde la perspectiva del estado. Por ejemplo, un estado puede proveer salidas tales como *succeded*, *aborted* o *preempted* y hasta donde el estado es incluido. En una máquina de estados, éstas salidas podrían ser relacionadas con otros estados, por lo tanto formando una transición. El concepto de **userdata** que corresponde a una estructura dentro del contenedor para pasar información entre los diferentes estados. Esto es útil cuando un estado retorna un valor de información sensorial. Lo cual permite que la información sea mantenida en la ejecución y este disponible para otras tareas.

Todos los contenedores tienen las siguiente propiedades:

- Contienen un diccionario de los estados donde los objetos que implementan la interfaz **Smach**.State están marcados con etiquetas de cadena.
- Contienen una estructura de datos, la cual puede ser accedida por todos los hijos de un contenedor.
- Para modificar la estructura de un contenedor, éste debe ser abierto.

Para abrir un contenedor se puede utilizar el comando de Python **with** o llamando al método *open()* y *close()* sobre el objeto contenedor.

3.2.2. Sintaxis y Visualización Gráfica de Máquinas de Estados

Para definir una máquina de estados en SMACH, se debe crear cada uno de los estados que la compone mediante la siguiente sintáxis:

```
# define state Bar
class Bar(smach.State):
    def __init__(self):
        smach.State.__init__(self, outcomes=['outcome2'])

    def execute(self, userdata):
        rospy.loginfo('Executing state BAR')
        if xxxx:
            return 'outcome1'
        else:
            return 'outcome2'
```

Se definen dos estados como ejemplo, lo primero es definir una clase con el nombre del estado, por ejemplo Bar, la cual hereda la clase `State` de Smach.

- El método **init** inicializa el estado.
- En el método **execute** se llevan a cabo las transciones de estados. Para ello se utiliza la variable de entrada `userdata` que contiene la información de entrada al estado.
- Cada estado tiene asociadas salidas, las cuales se definen en el método **init**.

La máquina de estados es un contenedor con un número finito de estados, se deben especificar las transiciones entre estados cuando se agregan. La sintaxis es la siguiente:

```
sm = smach.StateMachine(outcomes=['outcome4','outcome5'])
with sm:
   smach.StateMachine.add('FOO', Foo(),
                          transitions={'outcome1':'BAR',
                                       'outcome2':'outcome4'})
```

Lo que se hace es agregar el estado creado al contenedor, el cual se designa por **FOO**, el siguiente argumento corresponde a la clase creada. El tercer argumento describe las transiciones, en la cual se definen de la siguiente forma:

- Si la salida del estado FOO es **outcome1** se le indica que el estado al que pasa es BAR.
- En caso que la salida sea **outcome2**, la transición será a **outcome4**.

Por último cuando se crea el contenedor, también se pueden definir salidas. En el caso del código anterior se definen las salidas **outcome4** y **outcome5**, dado que todo contenedor es también un estado.

Entonces dentro de un archivo de extensión **.py** se definen las clases correspondientes a los estados y dentro del método principal **main** se crea el objeto contenedor y se agregan los estados. Por último, para ejecutar la máquina de estados, se debe agregar la siguiente línea, después de anexar los estados.

```
outcome = sm.execute()
```

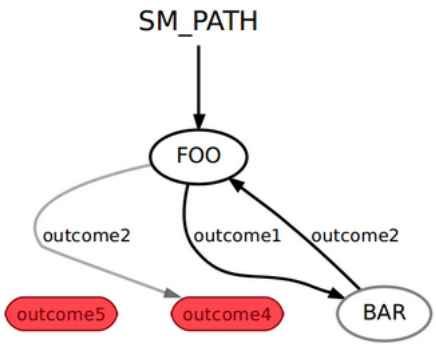


Figura 3.6: Ejemplo de una máquina de estados en SMACH.

La figura 3.6 se muestra un ejemplo de una máquina de estados simple que contiene dos salidas, por lo cual esta máquina puede ser utilizada como otro estado dentro de otra máquina de estados. Esto conlleva a lo que se conoce como jerarquía y por lo cual se construyen comportamientos complejos.

Para todo archivo de SMACH, se deben importar las clases `rospy`, `smach`, `smach_ros` para ejecutar la máquina de estados dentro de un nodo de ROS. La sintaxis es la siguiente:

```
#!/usr/bin/env python

import roslib; roslib.load_manifest('smach_tutorials')
import rospy
import smach
import smach_ros
```

SMACH provee un nodo (smach_viewer) dentro de ROS que permite visualizar la ejecución en tiempo real de la máquina de estados, la información que está pasando dentro de un estado los cuales se representan por medio de mensajes dentro de cada estado. Para visualizar la información, en el archivo que define la máquina de estados, se deben agregar las siguientes líneas después de haber sido agregados los estados.

```
sis = smach_ros.IntrospectionServer('server_name', sm,
                                    '/SM_ROOT')
sis.start()
```

Donde **server_name** corresponde al nombre con que se le da la instancia a la visualización de SMACH. Y **SM_ROOT** es el nombre que aparecerá cuando se muestre la máquina de estados, el cual puede ser cualquier nombre.

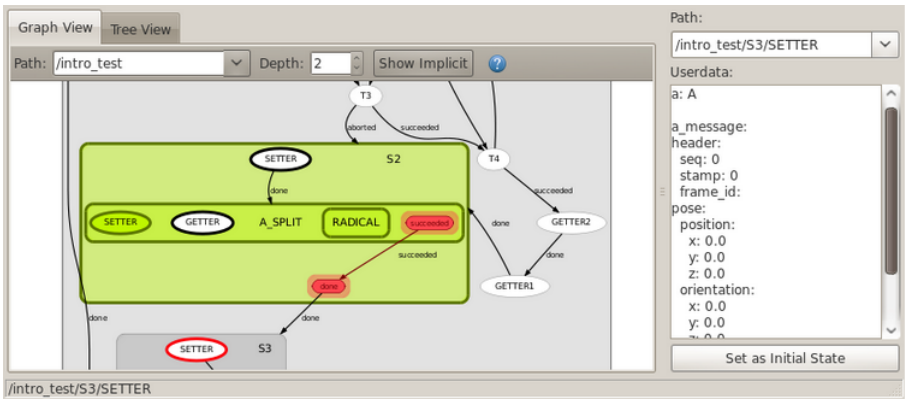


Figura 3.7: Ejemplo de la visualización y depuración de una máquina de estados en SMACH.

En la figura 3.7 se presenta la ejecución de una máquina de estados, el recuadro en verde corresponde al estado actual en ejecución. A la derecha en el *userdata* se puede ver la información que contiene el contenedor, de esa forma se puede verificar los cambios que se dan conforme a lo que diseñó.

3.2.3. Formas de Depurar el Código

La depuración del código consiste en construir un estado que puede entregar la información de un contenedor. La visualización de la información se puede realizar imprimiendo en consola las variables internas de cada estado, dado que el programa para depurar el código de SMACH no está completamente documentado.

3.2.4. Ventajas y Desventajas Técnicas

Ventajas

- Visualización en tiempo real de las transiciones que ocurren en una máquina de estados mediante smach_viewer.
- Se pueden anidar las máquinas de estados, esto quiere decir que una máquina de estados se puede considerar como un estado y ser anexado en el mismo archivo donde se define el contenedor general.
- Se puede transferir información de un estado a otro, para ello se debe definir en el método `__init__()` las entradas y salidas.

- Cuando se visualiza el estado en ejecución se puede ir viendo estado por estado y si este estado es otra máquina de estados, entonces se puede ver en particular como se ejecutan los sub-estados definidos en ella. Dispone de una visual que muestra la jerarquía de la máquina de estados como un árbol.

Desventajas

- El software del robot disponible a la fecha, está desarrollado en C++. Implementar SMACH significa realizar una interfaz de comunicación entre ambos lenguajes.
- SMACH es una librería independiente de ROS, pero los tutoriales publicados en el sitio web de ROS no abarcan todos los aspectos de esta herramienta. Se requiere investigar.

3.3. Selección de Librería para Generar Máquinas de Estados

3.3.1. Comparación de Ventajas y Desventajas

Los dos lenguajes proveen una interfaz para verificar el estado de ejecución de una máquina de estados. Pero difieren en la forma en que se debe realizar la depuración del código, por un lado XABSL dispone de las clases que se encuentran en XabslEngine.cpp para mostrar la información de los símbolos de salida y entrada. Por el contrario en SMACH no se puede visualizar la información de los símbolos mediante una interfaz adecuada, debido a que no se tiene la documentación completa para obtener los métodos que lo hagan posible.

Considerando que la plataforma de desarrollo del software del robot está en C++, se debe buscar una manera para ejecutar la máquina de estados y use la información que provee el software del robot. Una posibilidad es utilizar el entorno de desarrollo ROS, pero ello implica adaptar en nodos de ROS los diferentes módulos que componen a *Motion* y *Cognition*, lo cual escapa de los alcances de este trabajo. Una tercera opción es desarrollar todo en Python, pero ello implica que para los procesos de adquisición y procesamiento de imágenes serán más lentos, lo que significa un funcionamiento lento y torpe del robot.

Por otra parte la sintaxis de XABSL es más directa que la de SMACH, dado que en este último existen varios tipos de contendedores que pueden definir una máquina de estados y ello implica investigar cuales se adaptan los requerimientos de la complejidad del

comportamiento del robot. Finalmente es más complejo realizar una máquina de estados en SMACH a diferencia de XABSL que su sintaxis es simple. En ambos casos se pueden tener máquinas de estados concurrentes.

3.3.2. Discusión de uso en el Fútbol Robótico

EL desarrollo de comportamientos para el fútbol robótico no impone condiciones para encontrar la solución al comportamiento que defina un jugador lo más real posible. Lo que se requiere es tener comportamientos que puedan ser adaptables a las diversas situaciones que se presenten, con ello a medida que se complejizan más y más las decisiones, el lenguaje en el cual se decida desarrollar debe permitir la concurrencia y generar comportamientos más complejos. Por lo cual ambos lenguajes pueden ser útiles para dar una solución a lo que se necesita para que el robot tenga un módulo de toma de decisiones.

Por una parte XABSL es un lenguaje que ya ha sido utilizado en el fútbol robótico desde el año 2004 por el *German Team* y se ha ido mejorando con el tiempo, es independiente del robot que se usa, siendo utilizado en los robots Aibo de la compañía Sony y para automóviles autónomos. No así SMACH que es un lenguaje reciente, que tiene pocos años de uso y se está en investigación su uso, así como la creación de una documentación. Ya ha sido probado en robots de tipo diferencial como por ejemplo Hokubi, Husky, TurtleBot y PR2. Existen poca información de su uso en el robot Nao y es parte de la investigación verificar la compatibilidad con este.

3.3.3. Elección para el Desarrollo del *Behavior* en el Fútbol Robótico

Dado que el código del equipo UChRoboticsTeam está desarrollado en C++, es preferible optar por utilizar XABSL para desarrollar los comportamientos del robot, en desmedro de SMACH que significaría transportar todo lo realizado a ROS y sería un trabajo de un año.

Capítulo 4

Diseño e Implementación de Control de Sonar

El sensor le da la capacidad al robot de medir cantidades físicas. De esta forma se puede recibir y percibir información del ambiente que lo rodea; con el debido procesamiento de la información se logra crear una representación del mundo y con ello el robot puede interactuar con este. Las señales o estímulos percibidas por el sensor son procesadas, es decir, dentro del dispositivo existe un transductor que transforma la señal que puede ser energía mecánica, química, magnética, eléctrica, etc. a una señal eléctrica, cuya intensidad está en función de la intensidad del estímulo registrado. En la figura 4.1 se establece de forma sencilla como el robot adquiere la información desde el ambiente y la procesa para su uso en el programa principal.

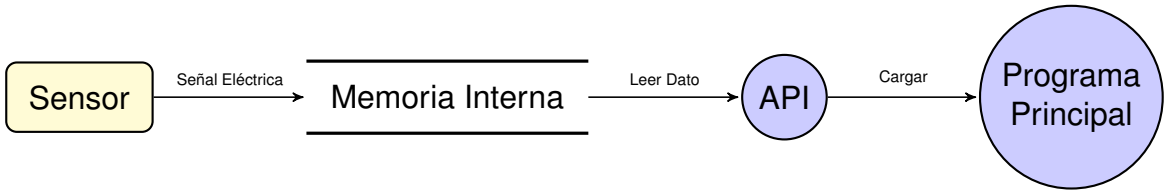


Figura 4.1: Esquema básico de la adquisición e interpretación del ambiente por parte de un robot.

La señal eléctrica se obtiene de la transducción de la cantidad física medida, esta es convertida por medio de un conversor análogo-digital en un valor hexadecimal que representa la cuantificación de la variable en observación, por ejemplo distancia. El valor obtenido se almacena en la memoria interna del robot, de este modo se puede acceder a ella desde el programa principal. La **API** controla al sensor y permite acceder a valores que

mide, para cargarlos en una representación que utiliza el programa principal, para tomar una decisión y posteriormente realizar la debida acción.

4.1. Configuración del Sonar en el Robot Nao

El robot está equipado con dos sonares de ultrasonido en su parte frontal(figura 4.2), como se muestra en la figura cada sonar se compone de un transmisor en la parte superior y de un receptor en la parte inferior para el lado derecho e izquierdo. Debido a la geometría del robot en su pecho, los sonares están rotados entorno a los ejes **y** y **z**.

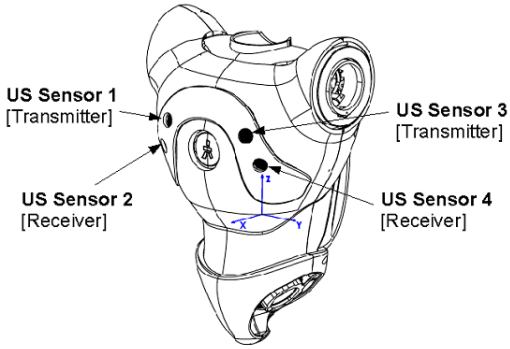


Figura 4.2: Diagrama físico de la localización de los sonares en el robot.

Cada emisor tiene un cono efectivo de 60° según el fabricante, este cono se expresa en el plano cartesiano **XY**. El rango de detección va desde los 25 cm hasta los 255 cm. Bajo los 25 cm, el sonar no puede entregar un valor de la distancia hacia el objeto. La figura 4.3 representa un ejemplo de la forma del campo de detección mediante ultrasonido, los arcos se definen por medio de un ángulo de apertura y el radio de alcance. No se muestra el campo efectivo en el eje **Z**, pero éste también es cercano a los 60° de apertura. El cono rojo representa al sonar derecho y el cono azul el sonar izquierdo.

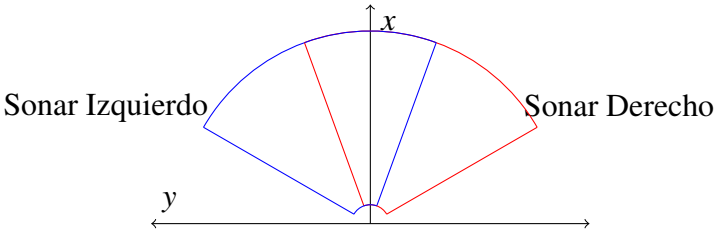


Figura 4.3: Rango efectivo de los sonares del Nao.

El sonar envía una onda, que puede chocar con muchos objetos durante su propagación, lo que significa que se generarán varias ondas reflejadas que serán detectadas por el receptor, el cual solamente reconocerá 10 ondas entrantes. El valor retornado por el dispositivo de ultrasonido es guardado en la siguiente ruta:

```
Device/SubDeviceList/US/Sensor/Value
```

El valor almacenado corresponde al primer eco o el objeto más cercano detectado por el dispositivo. Entonces para los 9 ecos restantes se tendrán las rutas que van desde **Value1** a **Value9**, los cuales van de menor a mayor, por lo que el último eco siempre corresponderá al objeto más lejano que esté en el campo de detección del sonar. Considerando que existen dos sonares en el robot, se dispone de 20 rutas de acceso para consultar la distancia de los objetos en torno éste, dichas rutas son:

Sonar derecho:

```
Device/SubDeviceList/US/Right/Sensor/Value
⋮
Device/SubDeviceList/US/Right/Sensor/Value9
```

Sonar izquierdo:

```
Device/SubDeviceList/US/Left/Sensor/Value
⋮
Device/SubDeviceList/US/Left/Sensor/Value9
```

4.1.1. Modo de Operación de los Sonares

Los sonares instalados en el robot disponen de varios modos de uso. Estos modos determinan la combinación transmisor-receptor que se utilizará. El DCM[1] es el encargado de gestionar el funcionamiento del dispositivo, por lo tanto el sonar enviará una onda conforme el modo que se le indique, para ello se entrega un entero de 7 bits que se describe a continuación:

Bit 6: Se usa para registrar una captura periódica de los valores del sonar. La frecuencia de actualización es de 100 ms. El comando se envía una sola vez al DCM.

[1]DCM (Device Communication Manager) es un módulo del sistema operativo NAOqi, encargado de la comunicación con los dispositivos eléctricos del robot, a excepción del sonido y la cámara.

Bit 3: Indica usar ambos transmisores al mismo tiempo.

Bit 2: Las dos capturas se realizan con un solo comando. El orden es izquierdo-izquierdo y derecho-derecho.

Bit 1: Selecciona el transmisor a usar. Para el izquierdo se indica por 0 y para el derecho por 1.

Bit 0: Establece el receptor a usar. Para el izquierdo es 0 y el derecho es 1.

Para el caso del bit 5 se considera como *don't care*. Entonces se pueden generar los siguientes valores que indicarán algunos de los modos de operar el dispositivo de ultrasonido:

Comando DCM	Binario	**Transmisores**	**Receptores**
0	0000000	Izquierdo	Izquierdo
1	0000001	Izquierdo	Derecho
2	0000010	Derecho	Izquierdo
3	0000101	Derecho	Derecho
4	0000100	Derecho e Izquierdo	Derecho e Izquierdo
6	0000110	Derecho e Izquierdo	Izquierdo y Derecho
12	0001010	Derecho e Izquierdo (al mismo tiempo)	Derecho e Izquierdo
68	1000100	Derecho e Izquierdo (periódico)	Derecho e Izquierdo
72	1001000	Derecho e Izquierdo (periódico)	Izquierdo y Derecho

Tabla 4.1: Valor para indicar el modo de operación de los sonares.

Los comandos presentados en la tabla 4.1 corresponden al valor en decimal que se pasa al DCM. En los casos en que se utilizan ambos transmisores, se pueden tener casos como *Derecho-Izquierdo* $\longrightarrow$ *Izquierdo-Derecho*, esto significa que los ecos del transmisor derecho serán registradas por el receptor izquierdo, es análogo para el transmisor izquierdo y el receptor derecho.

4.2. Estrategia de Control de los Sonares

Cada uno de los modos presentados en la tabla 4.1 definen un campo de detección único. Por lo tanto para efectos de este trabajo se considerará los modos en los cuales se usen ambos sonares al mismo tiempo, estos modos son: 4, 12, 68 y 70. Los pasos a seguir para adquirir la información de ambos sonares son:

1. Al iniciar el programa, indicar el modo de uso del dispositivo de ultrasonido.

2. Cuando los transmisores envían una onda, toma alrededor de 10 milisegundos en que vuelva la onda reflejada. Por lo tanto se debe esperar un tiempo desde que se envía la orden inicial, durante ese tiempo no se lee nada de los receptores.

3. Cuando pasa el tiempo establecido para asegurar que existe un valor coherente en los receptores, se indica que se puede leer en las respectivas direcciones de memoria asignadas para cada dispositivo.

4. Para los modos que tiene el bit 6 en 1, se deben leer los valores registrados por los receptores. Por otra parte si no se tiene el bit 6 en 1, entonces es necesario apagar los sonares para evitar posibles errores de lectura, de esa forma se asegura que no habrá un cambio inesperado de los datos por posible ruido de los sonares.

5. Cuando se cumpla el tiempo de lectura, se vuelve a enviar el comando de uso de los sonares o bien se espera 100 milisegundos en caso que sean los modos que usen el bit 6 en 1.

Basados en el esquema presentado en la figura 4.1, el programa principal dispone de la clase NaoProvider que es la que entrega los valores de los sensores y a su vez envía ordenes a los actuadores, es decir, es la que se encarga a bajo nivel de controlar las acciones que se deciden en el alto nivel, para ello hace uso de la librería o API del robot. La clase que gestiona los pasos mencionados anteriormente corresponde a USControl, la cual se encarga de ir actualizando los parámetros que se definan en la representación USRequest que es una clase de bajo nivel que permite comunicar NaoProvider con USControl.

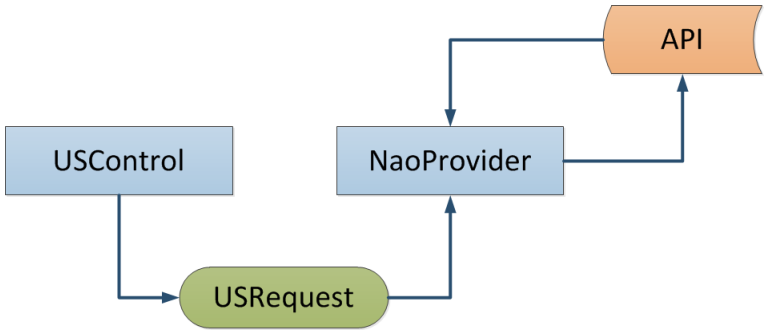


Figura 4.4: Esquema de control de los sonares.

En el esquema presentado en la figura 4.4, USControl se ejecuta periódicamente de modo que el método principal de este módulo se encarga de actualizar los parámetros de la clase USRequest cada cierto período. Por otra parte NaoProvider realiza la comunicación

con la **API**, la cual consiste en enviar instrucciones a los actuadores o bien leer los datos de los sensores y encoders del robot. Entonces, para el caso de los sonares NaoProvider lee los datos de USRequest y los envía a la API que se encarga de entregarlos al hardware del robot para su posterior ejecución. La figura 4.5 la señal en el tiempo del cambio de los modos de ejecución del sonar.

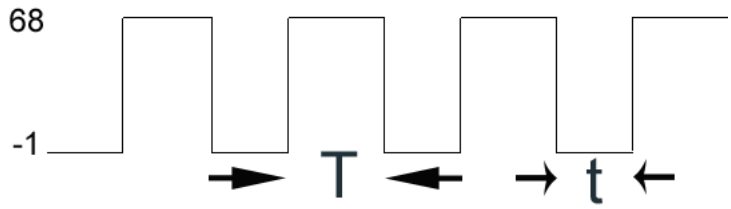


Figura 4.5: Diagrama de tiempo del control de los modos del sonar.

La representación **USRequest**, contiene dos variables:

sendMode: corresponde al modo de operación de ambos sonares. Es el entero de 7 bits que recibe el hardware de los sonares.

receiveMode: indica el modo de operación utilizado por los sonares. Se usa para indicar cómo se debe leer los datos de cada sonar dentro de la clase `NaoProvider`.

De la figura 4.5 se definen dos tiempos, el primero **T** es la duración en la cual los sonares están funcionando con el comando 68 de la tabla 4.1. Por otro lado, **t** es el lapso de tiempo en que los sonares no emiten ondas de ultrasonido. Los respectivos valores son 400 y 200 microsegundos.

4.2.1. Módulo USControl

Los sonares cada cierto tiempo estarán enviando una onda y captando los ecos. Dependiendo del modo de uso de los sonares es si se debe periódicamente variar **sendMode**. Este módulo se constituye de una clase que contiene los siguientes métodos:

USControl(): el constructor de esta clase se encarga de abrir y cargar los parámetros definidos en el archivo de configuración *USControl.cfg*.

update(*USRequest& usRequest*): gestiona la actualización de los parámetros de la clase USRequest. Se ejecuta periódicamente en el módulo de *Motion*.

Por otra parte los parámetros definidos para esta clase son:

- **lastSendTime**: esta variable lleva el tiempo de la última vez que se envió la orden de activar los sonares.
- **sendInterval**: esta constante define el tiempo de espera para enviar nuevamente una onda de ultrasonido.
- **ignoreAfterSwitchMode**: define el tiempo que se debe esperar entre el envío de la onda de ultrasonido y la lectura desde el DCM.

Para el caso de **sendInterval** y **ignoreAfterSwitchMode**, permiten que se pueda variar la reactividad de los sonares, por ejemplo reduciendo el intervalo de espera para leer los sonares, permite detectar objetos que se encuentren en movimiento dentro del rango de detección del robot. No así para un tiempo de espera más largo, que implica situaciones en las cuales se está frente a obstáculos fijos.

Algoritmo 1 Método **update** de la clase USControl.

```
1:  La variable t es el tiempo que ha pasado desde lastSendTime.
2:  if t > sendInterval then
3:      sendMode ← −1                          ▷ Apagar los sonares.
4:      recieveMode ← −1                       ▷ No leer datos.
5:      lastSendTime ← actualizar con el tiempo actual
6:  else if t > ignoreAfterSwitchMode then
7:      sendMode ← c    ▷ Encender los sonares con el modo de operación seleccionado.
8:      receiveMode ← c                        ▷ Indicar el tipo de lectura a realizar.
9:      lastSendTime ← actualizar con el tiempo actual
10: end if
```

El algoritmo 1 establece una estrategia simple para controlar el funcionamiento de los sonares, esto implica una situación que se presenta durante el juego en el cual el robot puede captar la onda de ultrasonido proveniente desde otro robot dentro de la cancha de juego. Esto se considera como interferencia, puesto que la onda que llega al receptor tiene la misma velocidad que la onda enviada, ello implica asumir una sola posibilidad que es que se crea que existe un objeto tan cerca del robot que simule un tiempo de recepción pequeño. Por lo cual el código anterior describe lo siguiente:

- La línea 2 verifica si se deben apagar los sonares, se envía -1. Esto se realiza como una solución básica para evitar la interferencia con otras ondas de ultrasonido. Al hacer intermitente el modo de operación de los sonares, permite evitar un error constante de lo que se registra con los receptores, sin embargo, existirán pequeños intervalos donde la medición de un objeto se vea afectada por esta interferencia.

- La línea 5 se encarga de verificar si es tiempo para volver a indicarle a los sonares el modo de operación definido en *USControl.cfg* y leer los datos que se obtengan, por último se actualiza la variable **lastSendTime** y cuando se ejecute nuevamente el método *update* se evaluará si apagar o leer datos del sonar.

Este algoritmo será periódico, cuya frecuencia es de 100Hz, la cual corresponde al ciclo de trabajo que tiene **Motion** que es el módulo encargado de llevar a cabo el control de la actuación y los sensores dentro del robot.

4.2.2. Adquisición de Datos con NaoProvider

El módulo definido por la clase NaoProvider se encarga de enviar la instrucción para operar los sonares y posteriormente leer los datos que se obtengan. El método encargado de realizar esto es **update(JointData& jointData)** que se ejecuta a 100Hz. Los datos que se obtenga de los sonares se ingresan a la clase SensorData que es una representación para la información proveniente de los sonares, sensores y encoders de los motores. Los parámetros importante de esta clase son:

- **usActuatorMode** define el modo de operación utilizado por los sonares, no se requiere que sea un el valor que se usa para el DCM, dado que es una codificación para el resto del código.

- **data** es un vector que contiene la información de todos los sensores del robot.

- **sensor** es una constante de tipo ENUM que contiene los identificadores de cada uno de los datos que se almacenan en el vector **data**. Para el sonar izquierdo los *id* se simbolizan por usL, usL1 hasta usL9 y para el sonar derecho usR, usR1 hasta usR9.

- **usTimeStamp** es el tiempo de registro para cuando se cargan datos en SensorData.

El algoritmo 2 describe la forma en que se encarga de almacenar los datos obtenidos mediante la API en un esquema de información común para el resto de los módulos presentes en *Motion*. La línea 1 permite tener un control de la lectura de los datos, por lo cual se leerá sólo cuando *receiveMode* sea distinto de -1. Para la lectura, se deben extraer los datos del vector de entrada *sensors* que contiene la información enviada por la API. La constante *lUsSensor* corresponde al *id* o índice de donde comienza la información de los sonares dentro del arreglo *sensors*. La línea 2 a la 5 realizan la tarea de guardar los ecos en **SensorData**, por otro parte la línea 4 asegura que no exista un valor nulo en el vector

Algoritmo 2 Rutina para cargar la información proveniente de la API en SonarData.

```
1:  if receiveMode != −1 then                    ▷ ¿leer los datos provenientes de la API?
2:      for i ∈ {1,...,20} do      ▷ Se leen las 20 mediciones provenientes de los sonares.
3:          data ← sensors[i+lUsSensor]
4:          sensorData.data[i] ← data ? data∗1000 : 2500
5:      end for
6:
7:      switch receiveMode
8:      case 68 :
9:          mode ← leftRightToLeftRightP
10:     case 70:
11:         mode ← leftRightToRightLeftP
12:     default:
13:         ASSERT(false)
14:     end switch
15: else
16:     mode ← none
17: end if
18: usActuatorMode ← mode
```

data, esto se presenta cuando no hay retorno de la onda enviada. Posteriormente las líneas 7 a 13 se encargan de indicar el modo de operación del sonar que se usó para obtener las mediciones. Si se está en el caso en que no se debe leer, la línea 16 indica que el modo utilizado es nulo. La variable **usActuatorMode** sirve para comunicar el modo de operación de los sonares a los demás módulos que usan la representación *SensorData*, esto queda expresado en la línea 18.

El código presentado es el nexo entre la información que se puede obtener con los diferentes sensores y la ejecución de la rutina principal que da vida al robot. Esta tarea se realiza en todo momento y representa la parte más importante para poder realizar las acciones y decisiones durante el juego. La figura 4.6 muestra el proceso completo de los módulos y representaciones que son necesarias para utilizar en alto nivel la información de la distancia de los objetos próximos al robot. A partir de SensorData los siguientes módulos se encargan de crear representaciones útiles del entorno que rodea al robot y transformar la información para la rutina de toma de decisiones.

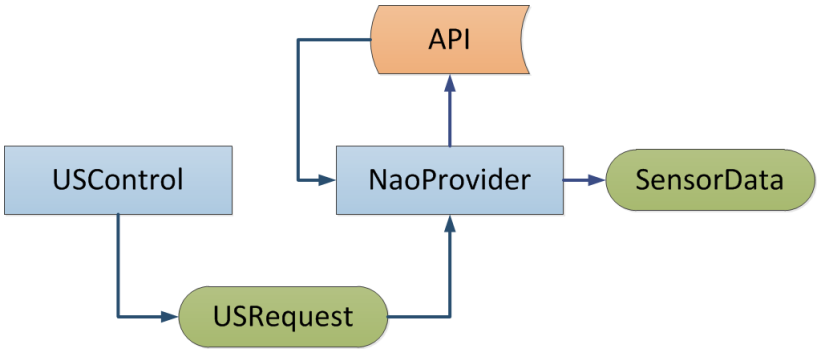


Figura 4.6: Esquema de los módulos y representaciones que conforman la adquisición de datos de los sonares.

4.3. Grilla de Obstáculos

La grilla corresponde a una representación bi-dimensional de cualquier objeto que se encuentra alrededor del robot. Durante un partido es común que el robot se encuentre obstaculizado por otros jugadores que se dirigen hacia la pelota, esto representa un tipo de interacción con un objeto que también se encuentra en constante movimiento, las posibles situaciones que pueden aparecer son las siguientes:

1. El camino del robot hacia la pelota está obstaculizado por otro robot, es decir, existirá una colisión que se penaliza como *pushing* dentro del juego y el jugador es retirado del campo.

2. En ciertos casos se puede no estar frente a una colisión directa, pero un robot puede obstaculizar la patada hacia el arco, por lo cual se debe patear la pelota hacia un área libre para volver a patear la pelota.

3. También es muy frecuente que ambos robot patean la pelota, lo cual implica que la pelota quede atrapada entre los pies de uno de los jugadores, en tal caso se penalizará al jugador y será sacado del campo de juego.

Puede que se den otras situaciones dentro del juego, aunque ello se deberá solucionar a nivel comportamientos que se diseñen y por lo tanto es necesario disponer de una buena representación de lo que sucede en el ambiente del robot. Por lo cual la grilla es un mapa local que se actualiza en el tiempo a través del cual se puede generar otro tipo de información que se comunica directamente con *Decision Making*, en otras palabras las máquinas de estado que en base a la información de los sensores disponibles del robot se puede realizar una acción que de en este caso una solución para evadir a otros robots.

4.3.1. Filtro de Mediana

Todo sensor contiene ruido, lo que significa obtener variaciones aleatorias en las mediciones, las cuales podrían indicar que existe un objeto más cerca o más lejos de lo que están en realidad o en otros casos indicar la presencia de objetos que en la práctica no están. Estas mediciones erróneas tienen un tiempo de duración que también es aleatorio, por lo cual es necesario procesar los datos con un filtro de mediana. Los valores que se almacenan en SensorData cambian cada 100 milisegundos, durante cada lectura es cuando puede aparecer ruido y en cuyo caso se debe escoger el tamaño de la ventana para el filtrado.

La solución se establece en el siguiente módulo perteneciente a ***Motion***, SensorFilter que toma como entrada las mediciones almacenadas en SensorData y la salida es guardada en FilteredSensorData que es una subclase de **SensorData**. La figura 4.7 muestra el proceso de filtrado de información, la salida ya contiene datos útiles para la construcción del mapa de obstáculos.

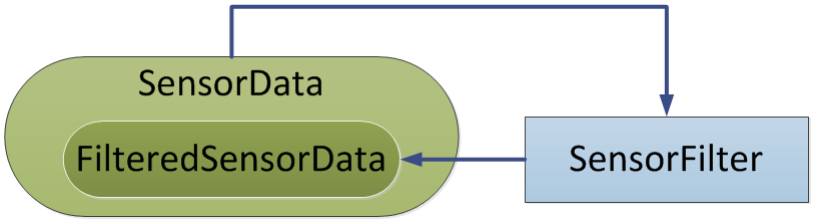


Figura 4.7: Lectura de las mediciones, filtrado y almacenamiento de la salida. El esquema describe el proceso que sigue la información para la creación de la grilla.

Algoritmo 3 Filtro de mediana de tamaño n.

Require: $V \in \mathbb{N}^m$ y $m \in \mathbb{N}$
Ensure: $v \in \mathbb{N}$

1: $buffer \leftarrow 0$ ▷ $buffer \in \mathbb{N}^n$ y $n \in \mathbb{N}$
2: **for** $i := 0$ e $i < n$ **do**
3: $\quad buffer_i \leftarrow V_i$
4: **end for**
5: Sort($buffer$)
6: **return** $v \leftarrow V_{\frac{n}{2}}$

El vector de entrada y la salida del algoritmo 4 se expresa en milímetros. El procedimiento que se realiza es tomar un vector de tamaño m y tomar solamente n valores que se guardan en el vector *buffer*. En la línea 5 se ordena el vector de menor a mayor y finalmente en la línea 6 se entrega el valor central del vector. El largo n del vector define el

tamaño de la ventana que se utiliza para realizar el filtrado y se asigna desde el archivo de configuración **usCalibration.cfg**.

Algoritmo 4 Proceso para el filtro de mediana y posterior asignación de datos en FilteredSensorData.

Require: $\vec{d_L} := \{d_{L_1}, ..., d_{L_{10}}\}$
Ensure: $\bar{d_L} \in [0, 2550]$ y $\bar{d_{L_1}} \in [0, 2550]$

1: $t > 0$
2: **for** $i := 0$ and $i < 3$ **do**
3: $\quad data_t \leftarrow \vec{d_{L_i}}$
4: $\quad llBuffer[i] \leftarrow data_t$
5: **end for**
6: $data_t \leftarrow mediana(llBuffer[0])$
7: $dataEco_t \leftarrow mediana(llBuffer[1])$
8: **if** $0 < data_t < D_{max}$ **then**
9: $\quad$ **if** abs($data_t$ - $data_{t-1}$) $< \eta$ **then**
10: $\quad\quad \bar{d_L} \leftarrow data_t$
11: $\quad$ **end if**
12: **else**
13: $\quad \bar{d_L} \leftarrow 2550$
14: **end if**
15: **if** $0 < dataEco_t < D_{max}$ **then**
16: $\quad \bar{d_{L_1}} \leftarrow dataEco_t$
17: **else**
18: $\quad \bar{d_{L_1}} \leftarrow 2550$
19: **end if**

El algoritmo 4 muestra solamente el filtrado y posterior asignación de la información para los datos obtenidos con el sonar izquierdo. Para el sonar derecho el procedimiento es el mismo donde solamente los datos filtrados se asignan a la respectiva ubicación dentro de FilteredSensorData.

La lógica del algoritmo se puede dividir en cuatro pasos:

1. En la línea 2 a la 5 se lee el vector de los 10 ecos registrados por el receptor izquierdo, en dicho caso solamente se toman los tres primeros ecos, los cuales son guardados en un vector temporal. La variable t indica que la medición es en el tiempo.

2. Después se obtiene la mediana para los dos primeros ecos, cabe destacar que esta mediana es en el tiempo, es decir, $llBuffer$ almacena datos en el tiempo para realizar el filtrado. Ver líneas 6 y 7.

3. La línea 8 a la 13 realizan la asignación del dato filtrado para el caso del primer eco. Esto consiste en verificar que el dato sea mayor a cero y sea menor a D_{max} que limita el rango de detección a nivel de software. En la línea 9 se verifica si la medición actual y la anterior difieren en más de 45 milímetros(η), entonces se asigna a FilteredSensorData como el valor $\bar{d}_L$, de lo contrario se mantiene el valor anterior($\bar{d}_{L_{t-1}}$). Esto se realiza para evitar variaciones pequeñas que puedan perturbar la generación de la grilla, pero al mismo tiempo no reducen la reactividad de las mediciones realizadas. Por otra parte si no se cumple la línea 8, entonces se pasa a la línea 13 donde se asigna 2550 milímetros, lo que significa que no hay objeto dentro del rango de detección del robot. Este límite impuesto por D_{max} es configurable desde el archivo **usCalibration.cfg** y su valor dependerá del comportamiento que se diseñe más adelante.

4. Las líneas 15 a la 19 asignan el siguiente eco en FilteredSensorData bajo una lógica similar, con la diferencia en que no se impone un umbral de comparación con las mediciones anteriores.

El segundo eco representa la detección de un segundo objeto por el mismo sonar, dado que durante el juego es importante considerar el obstáculo más cercano, es por ello que no se realiza una comparación de las mediciones anteriores con la actual para la asignación final del segundo eco, el filtro de mediana es suficiente.

4.3.2. Actualización de la Grilla

Con los datos de ambos sonares filtrados, se puede realizar la actualización de la grilla, que contiene cada de uno de los objetos detectados como un mapa del entorno cercano al robot. Está implementada en la clase MultiUSDataObstacleGridProvider, los métodos relevantes son:

- **update(USObstacleGrid & usObstacleGrid)**
- **checkUS()**
- **ageCellState()**
- **loadMeasurements(Measurements& outMeasurements)**
- **groupMeasurements(Measurements& measurements,GroupMeasurements& outGroups)**

El método principal es update que se ejecuta cíclicamente. La entrada es una referencia a un elemento de la clase usObstacleGrid que corresponde a la representación de la grilla de obstáculos. Su funcionamiento es simple, constantemente debe verificar si ha entrado al estado de jugar conocido como **Play** después de haber estado en **Set**, para lo cual limpia la grilla dado que la información que haya quedado anteriormente no sirve. En caso de que se ha penalizado al jugador, se suspende la actualización de la grilla, puesto que el robot debe permanecer en un costado del campo de juego y no necesita identificar obstáculos durante ese estado.

El algoritmo 5 describe la actualización de la grilla realizada durante el juego.

Algoritmo 5 Método principal de la clase MultiUSDataObstacleGridProvider

Require: $state \in \{Initial, Ready, Set, Play, Finish, Penalized\}$

1: **if** $lastState == Set$ y $state == Play$ **then**
2: Limpiar la grilla
3: **end if**
4: Actualizar el estado de las celdas de la grilla
5: **if** $state != Penalized$ **then**
6: Revisar la información de los sonares
7: **end if**

Los métodos *ageCellState* y *checkUS* utilizan directamente la grilla. Esta representación consiste de los siguientes parámetros:

Celdas: se crea un arreglo de 2116 celdas, cada una contiene la información de la última vez que fue actualizada, el **cluster** al que pertenece y el **estado** que es un entero que puede ser cero o mayor a cero. El tamaño de la celda es 60.

Umbral de celda ocupada: se define como el límite inferior del estado de una celda para ser clasificada como celda ocupada.

Máxima celda ocupada: este valor establece el límite máximo que puede tener el estado de una celda, cuando se alcanza este valor se deja de aumentar su estado.

La función *ageCellState* es la que se encarga de actualizar el estado de cada una de las celdas que componen a la grilla, este método se ejecuta en la línea 4 del algoritmo 5.

El método *checkUS* se encarga de verificar la información que llega desde los sonares e ingresarla en la grilla de obstáculos.

Algoritmo 6 Función *checkUS*

Require: $\bar{d_L}$, $\bar{d_R}$ pertenecientes a FilteredSensorData

```
1:  loadMeasurements(measurements)
2:  groupMeasurements(measurements, groups)
3:  for group ∈ groups do
4:      resetIntervals()
5:      usedModes[2] ← {false}
6:      for d ∈ groups do
7:          usedModes[d → mode] ← true
8:      end for
9:      for m ∈ {leftToLeft, rightToRight} do
10:         if usedModes[m] then
11:             increaseHitCount(sensorModeInterval[m])
12:         end if
13:     end for
14:     for m ∈ {leftToLeft, rightToRight} do
15:         if !usedModes[m] then
16:             resetHitCount(sensorModeInterval[m])
17:         end if
18:     end for
19:     i ∈ {interval_1, interval_2, interval_3}
20:     for i do
21:         if i → hitCount > 0 then
22:             grid ← i
23:         end if
24:     end for
25: end for
```

El algoritmo 6 corresponde al método que se encarga de ingresar la información proveniente de los sonares a la grilla, la complejidad de éste es mayor de lo que se presenta, a continuación se explica las partes principales de éste.

1. La línea 1 se encarga de leer la representación FilteredSensorData y guarda la información filtrada en un arreglo de tipo Measurement que es una subclase que tiene como parámetros *mode* que indica si es obtenida con el sensor derecho o izquierdo, y *distance* que es el valor obtenido con el sonar. El arreglo se carga con la información de ambos sonares y se le asigna con el sonar que se obtuvo.

2. La línea 2 agrupa las mediciones que difieran menos de una distancia umbral que se establece como un parámetro dentro del archivo **usObstacleGrid.cfg** con el nombre de *groupingDistance*.

3. A partir de la línea 3 se toma un grupo dentro los que están en el vector *groups*, entonces en las líneas 4 y 5 ponen a cero el *hitCount* de todos los intervalos y pone en falso los modos de adquisición de datos respectivamente.

4. En la línea 6 se toma una medición de un grupo, cada medición tiene definido su modo de adquisición, por lo cual en la línea 7 se asigna *true* dentro del vector *usedModes*.

5. La línea 9 a la 13 verifican si aumentar el *hitCount* del intervalo que corresponde al modo de adquisición de los datos, para la distancia analizada.

6. La línea 14 a la 16 pone a cero el *hitCount* de cada uno de los intervalos que usan el modo de adquisición datos no utilizada para la distancia *d* analizada.

7. En las líneas 17 a la 22 se ingresa la información a la grilla, para ello se verifica que el *hitCount* del i-ésimo intervalo sea distinto de cero. La información que se ingresa es la distancia y el intervalo que en la grilla significa aumentar el estado de todas las celdas que se encuentren a una distancia *d* del centro de la grilla y pertenezcan al intervalo angular definido por el i-ésimo intervalo.

Para realizar la asignación de la mediciones obtenidas en la grilla, se define el conjunto de intervalos que está directamente relacionado con el modo de operación de los sonares. El modo de operación es el 68, por lo cual se generan tres intervalos por la superposición de los conos derecho e izquierdo, este tercer intervalo se presenta en el centro del robot.

Dentro de la clase MultiUSDataObstacleGridProvider se define la subclase Interval que es una representación que contiene los siguientes parámetros:

- *Angle Left* que corresponde al ángulo de apertura del intervalo.
- *Angle Right* es el ángulo de cierre del intervalo.
- *hitCount* corresponde al peso que tiene el intervalo.

Cada modo de operación de los sonares definirá distintos intervalos, por lo cual en el constructor de la clase principal se define el número de intervalos que se utilizarán y se cargan los ángulos que definen su apertura y cierre desde el archivo de configuración **usCalibration.cfg**.

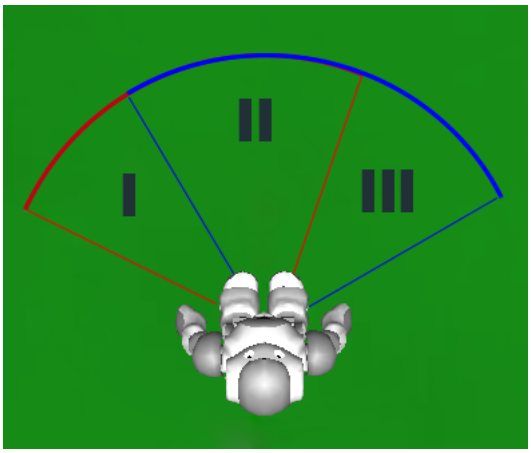


Figura 4.8: Conos formados por el modo de operación de los sonares.

La figura 4.8 muestra la disposición que tendrán los conos de detección del robot. El cono central se forma de la superposición de los conos derecho e izquierdo de los sonares. Por otra parte si un objeto está en el cono I o cono III, este será detectado exclusivamente por el sonar izquierdo o derecho respectivamente. Cada cono tiene dos parámetros que definen el tamaño de éste, para obtener estos ángulos de apertura y cierre se debe calibrar el robot, el cual consiste en detectar donde comienzan los intervalos de detección para el sonar derecho e izquierdo.

El método requiere el uso de un semi-circulo graduado como muestra la figura 4.9 donde el centro es ocupado por el robot, se coloca un objeto a 90^o del robot y comienza a moverse en sentido horario, cuando se detecta el objeto por el sonar izquierdo, se establece el ángulo de apertura del sonar izquierdo, cuando se deja de detectar se obtiene el angulo de cierre del sonar izquierdo. El procedimiento es el mismo para el sonar derecho. En caso que hubieran más intervalos, se debe realizar este mismo procedimiento para cada uno y así obtener los intervalos que se ingresan en el constructor de la clase principal.

Las líneas 9 a 18 del algoritmo 6 modifican el *hitCount* de los tres intervalos utilizados. Dada una medición de un objeto que se encuentra en el cono I(figura 4.8), los pasos a seguir según el algoritmo de *checkUS* son:

1. En el arreglo *measurements* se almacena como el par $(d, leftToLeft)$, dado que se obtuvo con el sonar derecho.
2. Como sólo se obtiene una medición el grupo es de tamaño 1.
3. Cuando se comienza a recorrer los arreglos, se llega a la medición obtenida, se pone en *true* el modo *leftToLeft* en el vector *usedModes*.

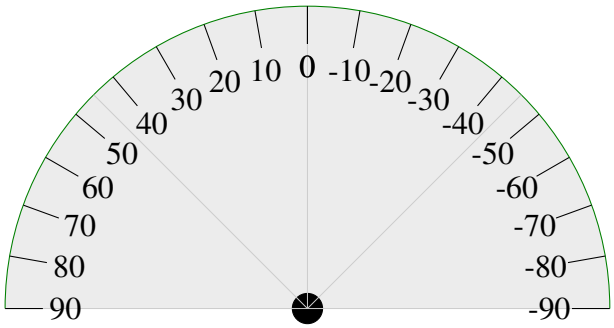


Figura 4.9: Bosquejo del círculo para calibrar el rango de detección de los sonares.

4. Después se incrementa el *hitCount* para los modos que tienen valor *true*. En este caso solo para el modo *leftToLeft*. Los intervalos que aumentan son I y II, ya que están dentro del campo de detección de este modo.

5. Después de asignar, se ponen a cero el *hitCount* de todos los intervalos que pertenezcan a los modos que estén en *false*. Dado que *rightToRight* no fue utilizado, los intervalos II y III pasan a 0, ya que pertenecen a este modo.

6. Finalmente sólo el intervalo I tiene un valor distinto a cero, por lo cual se ingresa a la grilla de obstáculos. Para ello se entregan la distancia *d* y los ángulos de apertura y cierre. Una función se encarga de transformar los datos a la grilla, lo que significa asignar un estado mayor a cero a las celdas que pertenezcan al intervalo entregado y a la distancia *d* medida desde el centro de la grilla.

4.4. Modelo de Obstáculos

Para obtener los obstáculos, se utiliza la representación definida por la clase ObstacleModel. Los parámetros que definen la representación son:

- **type** indica si se obtuvo mediante sonares, visión, choque con los brazos o bumpers de los pies.
- **center** es la posición del centro del obstáculo respecto del robot.
- **closestPoint** es el punto más cercano del obstáculo respecto del robot.
- **leftCorner** y **rightCorner** definen los límites del obstáculo.

Esta representación es modificada por la clase ObstacleCombinator que se encarga de leer la grilla de obstáculos. Para ello utiliza los métodos *generateCellCluster*

y *generateObstacleFromCurrentCluster* que son descritos por los algoritmos 7 y 8 respectivamente. El primer algoritmo tiene la finalidad de buscar dentro de la grilla todas las celdas que tengan un estado mayor a cero, es decir, las que están ocupadas. Cuando se cumple la línea 4, la celda se ingresa a un *cluster* que es un vector de celdas, la característica principal del algoritmo es que analiza las celdas vecinas y de esa forma recorre un segmento dentro de la grilla y genera el vector final con todas las celdas que ocupen un área determinada. La salida que se obtiene es un conjunto de *clusters* que representan los obstáculos almacenados en la grilla.

Algoritmo 7 Generar cluster desde la grilla

```
1: for y := 0 y y < 46 do
2:     for x := 0 y x < 46 do
3:         c = cells[T(x,y)]
4:         if c.state > 0 then
5:             cluster_i ← c
6:         end if
7:     end for
8: end for
```

Después se toman los clusters generados y se determina el centro, la distancia a la que está y los límites que lo definen para ingresarlo dentro del vector de obstáculos como muestra la línea 10 del algoritmo 8. En esa misma línea se realiza una transformación para generar la información necesaria para la representación definida por ObstacleModel a partir de un cluster, se realiza esto con todos los clusters existentes. *Obstacles* es un vector de tipo *ObstacleModel* que contiene todo tipo de obstáculos que se puedan obtener por medio de sonares, visión, brazos o bumpers de los pies.

Algoritmo 8 Obtener un obstáculo a partir de un clúster

```
1: robotPosition ← (23,23)
2: closestPoint ← (46,46)
3: closestPointSqrDist ← 4232
4: for c ∈ clusters.cells! = 0 do
5:     if distance(c, robotPosition) then
6:         closestPoint ← vector(c)
7:         closestPointSqrDist ← distance(c, robotPosition)
8:     end if
9: end for
10: obstacles ← obstacle(c)
```

La figura 4.10 establece todos los módulos y las representaciones necesarias para adaptar la información obtenidas con los sonares y posteriormente entregada para ser utilizada en la etapa de toma de decisiones. A partir del módulo **MultiUSDataObstacleGridProvider** la ejecución se realiza a 30 Hz, dado que se está en el proceso de Cognition, por ende la actualización es más lenta respecto de Motion.

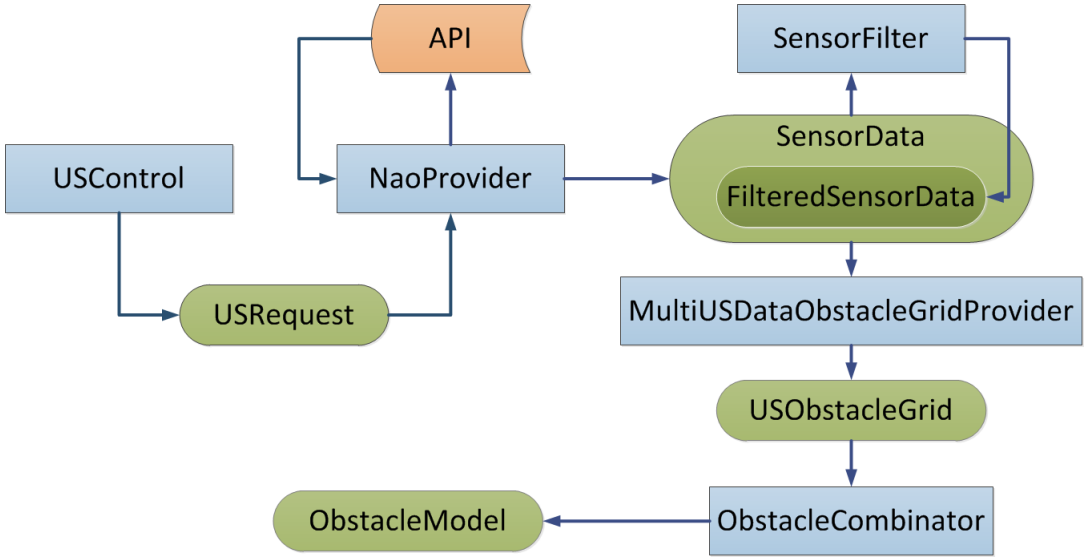


Figura 4.10: Esquema general de los módulos para generar los obstáculos.

4.5. Creación de Símbolos para XABSL

A partir de la representación ObstacleModel, la clase ObstacleSymbols genera los símbolos para XABSL. Estos son:

- obstacle.left.distance, obstacle.left.angle, obstacle.arc_left.
- obstacle.right.distance, obstacle.right.angle, obstacle.arc_right.
- obstacle.center.distance, obstacle.center.angle, obstacle.arc_center.

Se definen tres parámetros que representan la distancia del obstáculo en el cono en que fue detectado, el ángulo del borde más cercano al robot y el largo del arco que ocupa dentro del cono. De esta forma se establece una caracterización del objeto para que el robot pueda encontrar un camino hacia una posición libre.

Algoritmo 9 Lectura del vector de obstáculos y creación de los símbolos para XABSL

```
1:  for obstacle ∈ Obstacles do
2:      if obstacle → type == US then
3:          sqr ← obstacle → center → abs()
4:          angle ← obstacle → center → angle()
5:          if abs(angle) < 5 y sqr < 3000 then
6:              obstacle.arc_center ← 40πsqr / 180
7:              obstacle.center.distance ← sqr
8:              obstacle.center.angle ← 0
9:          end if
10:         if angle < −5 y sqr < 3000 then
11:             obstacle.arc_right ← 40πsqr / 180
12:             obstacle.right.distance ← sqr
13:             obstacle.right.angle ← −20
14:         end if
15:         if angle > 5 y sqr < 3000 then
16:             obstacle.arc_left ← 40πsqr / 180
17:             obstacle.left.distance ← sqr
18:             obstacle.left.angle ← 20
19:         end if
20:     end if
21: end for
```

El algoritmo 9 solo establece los obstáculos para cuando estos son de tipo ultrasonido, línea 2. Posteriormente el método de clasificación se basa en el ángulo respecto al robot y con ello se puede tener más de un obstáculo ingresado a la máquina de estados de evasión.

Capítulo 5

Diseño General de la Máquina de Estados

En este capítulo se desarrolla el comportamiento para que el robot emule un jugador que va hacia la pelota y la patea. La adquisición de la información del entorno provine de los siguientes dispositivos:

- Cámara para obtener la información de la pelota, arco y líneas para la localización.
- Sonares para detectar los robots que se encuentren en el campo.
- Sensores táctiles en los pies, para determinar si existe colisión con un robot que yace en el suelo.

El comportamiento del jugador que se diseñó, se denomina **Striker**, éste se encarga de realizar las acciones para que el robot realice los goles durante el partido. Corresponde a uno de los comportamientos que se utilizan dentro del esquema de juego que posee el equipo UChRoboticsTeam, dentro de esos se encuentran:

Striker: encargado de realizar los goles, su comportamiento corresponde al de un delantero para el caso del fútbol humano.

Supporter: acompaña al jugador que lleva la pelota, es una especie de compañero que colabora en aumentar la probabilidad de realizar goles.

Goalie: realiza las acciones de arquero.

Defender: es el defensa que permanece cerca del área del equipo.

Para desarrollar el comportamiento se deben considerar ciertos aspectos que se dan durante un partido. Una de las situaciones más importantes en el juego, es evitar los obstáculos, que son los robots del equipo contrario. Un robot del mismo equipo no es un obstáculo. Esto está determinado por un comportamiento de mayor jerarquía que controla la asignación de los roles de cada jugador, para el caso del equipo, sólo un robot será **Striker**, lo cual se determina por la condición del robot más cercano a la pelota.

La figura 5.1 muestra una de las tantas situaciones que se presentan en un partido de fútbol robótico. El comportamiento de Striker debe realizar los siguientes pasos para hacer un gol:

1. Caminar hacia la pelota.
2. Alinearse en dirección hacia el arco.
3. Patear la pelota.

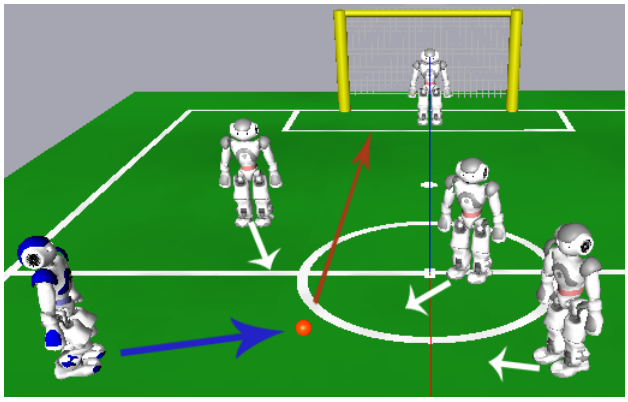


Figura 5.1: Ejemplo de una situación típica durante un partido.

En la figura 5.1, la flecha azul corresponde a caminar hacia la pelota y la flecha roja es la dirección en la que debería patear para realizar un gol. Por otra parte los jugadores contrarios estarán realizando las mismas acciones y suele darse que varios jugadores se dirigen hacia la pelota, lo cual se representa con las flechas blancas para los robots del equipo rojo. Entonces el comportamiento de Striker debe realizar estos tres pasos mencionados, pero para solucionarlos se presentan las siguientes condiciones durante el juego:

I. Mantener siempre a la vista la pelota, en caso que deje de verla por un determinado tiempo deberá buscarla.

II. Dependiendo de la posición respecto al robot, éste debe elegir la manera más rápida para aproximarse.

III. Cuando el robot camina hacia la pelota, los jugadores contrarios pueden obstaculizar el paso, entonces debe evadirlos.

IV. Cuando está próximo a la pelota, debe determinar la orientación para patear hacia el arco.

V. Seleccionar el tipo de patada en base a la distancia respecto del arco y la orientación hacia éste.

Cada condición corresponde a una transición de un estado a otro dentro del comportamiento del *Striker*.

5.1. Construcción del Comportamiento Striker

Se define el *option* STRIKER.XABSL correspondiente a la máquina de estados finita de mayor jerarquía para el comportamiento del jugador.

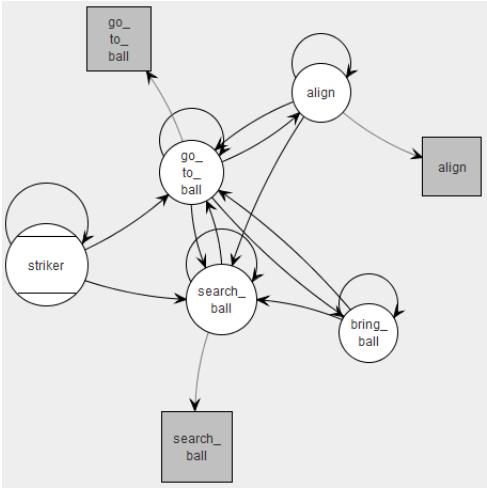


Figura 5.2: Diagrama de estados del Striker.

La figura 5.2 muestra el grafo de los estados que componen a **Striker**. Se definen 4 estados principales:

striker : se define como el estado inicial y se ejecuta cuando se pasa al estado **play** dentro del juego.

go to ball: es el comportamiento para ir hacia la pelota, en este caso se hace la llamada a un *option* que define la máquina de estados que controla la llegada a la pelota.

align : cuando el robot esté a una distancia $d < \delta$ de la pelota, se ejecuta un comportamiento que alineará el robot para patear hacia el arco.

bring_ball: lleva la pelota hacia el lado contrario. No se requiere de definir una máquina de estados para ejecutar la acción.

search ball: busca la pelota cuando la deja de ver un determinado tiempo τ. Define una nueva máquina de estados para considerar los casos en la que búsqueda sea activa o pasiva.

Striker comprende acciones muy complejas para caminar hacia la pelota, buscarla o alinearse para patear. Es decir, cada una comprende estados que ayudan a adecuar una acción en el comportamiento usando los símbolos de salida ya predefinidos en la plataforma de desarrollo del robot. El esquema de la figura 5.2 es cíclico, eso implica que no existe un estado terminal y por lo cual siempre estará en algún estado dependiendo de las condiciones del juego en cada momento.

El estado inicial (***striker***) tiene dos acciones básicas: tener al robot de pie y mirar la pelota. En el capítulo 3, se mencionó que XABSL puede utilizar comportamientos básicos que se construyen dentro de la plataforma del software del robot, por lo cual para la realización de las acciones que se darán a conocer, se utiliza la siguiente sintaxis dentro de un *action*:

```
action
{
  motion.walk_pedantic = true;
  motion.type = walk;
  motion.dribbling = true;
  motion.walk_target.x = ball.seen.x;
  motion.walk_target.y = ball.seen.y;
  motion.walk_target.rot = ball.seen.angle;
  motion.walk_speed = 80;
  head.control_mode = look_at_ball;
}
```

El robot tiene predefinido las acciones para la realización de la caminata hacia la pose (x, y, θ) y el giro de la cabeza hacia un punto en el espacio (x, y, z). Para ello se utilizan los símbolos de salida de **Motion** que es la clase encargada de recibir las solicitudes que se generen dentro de XABSL y ejecutarlas. Cada estado que se construya para el

comportamiento del jugador, utilizará esta sintaxis para desarrollar los movimientos más complejos. Para el símbolo de salida motion.type se define si la acción es estar de pie (stand), patear la pelota (bike), caminar a hacia una pose (walk) o realizar una acción especial (special_action), que son movimientos previamente programados que hará el robot cuando se le solicite. Dependiendo de la salida de este símbolo, se utilizarán los otros símbolos para agregar más parámetros como la pose a la que se desea llegar, la velocidad de la caminata (motion.walk_speed), etc. Para el caso de las acciones con la cabeza, se utiliza el símbolo interno head.control_mode de tipo enum para indicarle los comportamientos diseñados para éste.

Algoritmo 10 Árbol de decisiones para las transiciones desde el estado *striker*.

Require: $t_{ball} > 0, t_{game} > 0, (x_{ball}, y_{ball})$

if $t_{ball} > 4000$ **then**
    go to search_ball
else if $ball \rightarrow was\_seen$ **and** ($t_{game} > 10000$ **or** $kick\_off$ **or** $\|x_{ball}\| > 500$ **or** $\|y_{ball}\| > 500$) **then**
    go to go_to_ball
else
    stay
end if

action ← look at ball and stand

El algoritmo 10 describe las decisiones que se toman para la transición a otro estado, para ello se utiliza la información de la pelota y la información del estado del juego. Los símbolos de entrada para la pelota son:

- ▷ ***ball.was_seen*** : un booleano que indica, si se está viendo la pelota constantemente.
- ▷ ***ball.estimate.x*** , ***ball.estimate.y*** : es la posición de la pelota respecto del robot en el plano XY.
- ▷ ***ball.time_since_last_seen*** : tiempo en milisegundos desde la última vez que se vio la pelota.

Para el estado del juego se utilizan los siguientes símbolos de entrada:

- ▷ ***game.time_since_playing_started*** : tiempo desde que se dio inicio al partido.
- ▷ ***game.kick_off_team*** : indica que equipo patea la pelota en el inicio de cada tiempo.

Definido los símbolos a utilizar, se pueden explicar las decisiones que se toman cuando se entra al estado inicial del comportamiento del jugador.

(a) La línea 1 establece como tiempo máximo 4 segundos para volver a buscar la pelota, t_{ball} representa el símbolo de entrada que indica la última vez que se vio la pelota, el cual vuelve a cero cuando la encuentra nuevamente.

(b) En la línea 3 se verifica si el robot debe ir hacia la pelota, en caso de que el equipo contrario no ha realizado la patada inicial en los diez primeros segundos o está a menos de 500 milímetros de la pelota o bien tiene el derecho de realizar la patada inicial. Por lo cual irá al estado *go_to_ball*.

(c) En caso que ninguna de las decisiones anteriores sea verdadera, se mantiene en el mismo estado esperando a que se genere un cambio en la información.

(d) Las acciones que se realizan en este estado son mirar la pelota mediante el símbolo *look_at_ball* y estar detenido.

Bring_Ball

Este estado tiene la única función de caminar pateando la pelota. Esta acción consiste en ir golpeando la pelota cada vez que avanza de modo de llevar la pelota al lado contrario, ya que patear desde una distancia muy grande se pueden dar las siguientes situaciones:

- ▷ Dependiendo de la fuerza de la patada, es posible que la pelota no alcance a llegar al arco contrario.
- ▷ La cancha tiene un largo de 9 metros, considerando que la velocidad del robot es de alrededor de 50 centímetros por minuto, suponiendo que la pelota no alcanza a llegar al arco oponente, deja al robot una distancia que le tomará mucho tiempo en recorrer para hacer un gol. No es óptimo este esquema de juego, debido a que permite a cualquier jugador contrario patear mientras el robot camina hacia la pelota.

La figura 5.3 muestra al jugador en azul que se dispone a patear la pelota que está cerca de su área, suponiendo que las condiciones impidan que la pelota pueda viajar una distancia muy grande, la pelota quedará en el medio campo del equipo contrario. Por lo tanto estos jugadores se disponen a avanzar en dirección a la pelota, lo cual implicaría que el jugador en azul debe avanzar una distancia d que es muy grande respecto a la distancia a la cual se encuentran los otros robots.

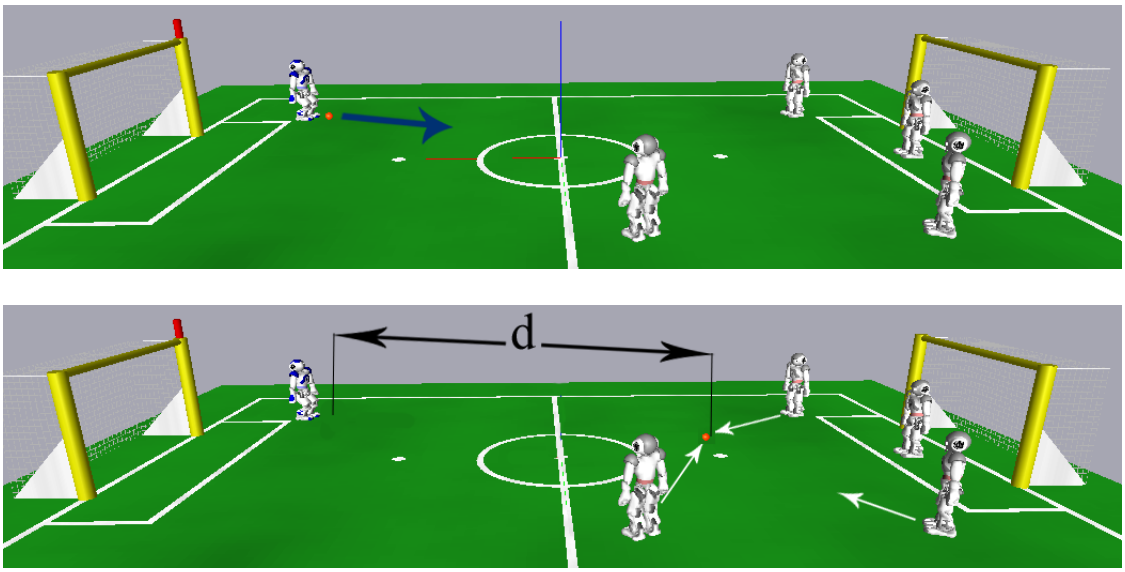


Figura 5.3: Situación en la que la distancia es grande entre el robot y el arco oponente.

Algoritmo 11 Decisiones para realizar a transición de estados desde ***bring_ball***.

```
1:  procedure DECISION
2:      if t_ball > 4000 then
3:          go to search_ball
4:      else if d_ball > 600 or x_ball_global > 1100 or ||y_ball_global|| > 1100 or d_obstacle_closest <
    400 then
5:          go to go_to_ball
6:      else
7:          stay
8:      end if
9:  end procedure
10:
11: procedure ACTION
12:     motion.walk ← (x_ball, y_ball, θ_ball)
13:     motion.walk_speed ← 80
14:     motion.walk_pendantic ← true
15:     head.control_mode ← look_at_ball
16: end procedure
```

En el algoritmo 11, la línea 2 establece la condición de que no puede dejar de ver la pelota por más de 4 segundos. En la línea 4 se indica que si el robot está a más de 600 milímetros de la pelota o si la posición de la pelota en el campo está sobre los 1100 milímetros, lo que equivale a decir que esté dentro del medio campo contrario o si hay

un obstáculo presente, entra a *go_to_ball*. La línea 12 le indica a ***Motion*** la pose a la cual se debe mover, esta corresponde a la posición en la que se encuentra la pelota respecto al robot. Posterior en la línea 13 se pasan la velocidad de la caminata y en la línea 14 se indica que hay movimiento pendular, por lo demás siempre está mirando la pelota. Mientras no se cumpla la línea 4, el robot estará avanzando hacia adelante con la pelota siendo empujada por el movimiento de la caminata. En ese caso también se ejecuta una acción básica para realizar el dribleo de la pelota, fuera de XABSL. Para el caso de $(x_{ball\_global}, y_{ball\_global})$ corresponde a la posición de la pelota respecto al centro del campo.

5.2. Comportamiento GO_TO_BALL

Cuando el robot ve la pelota, debe caminar hacia esta. La forma más rápida de hacerlo es siempre caminar en línea recta, dado que el robot puede ir a mayor velocidad. Entonces la rutina a realizar es la siguiente:

1. Ve la pelota, gira para alinearse con la pelota.
2. Camina hacia directo hacia la pelota.

En esta etapa es cuando se presentan los obstáculos, representados por los robots contrarios. Por ello es necesario que el robot mientras camina pueda esquivarlos y posteriormente continuar en dirección hacia la pelota. Se define el comportamiento que se ejecutará dentro de ambos estados que componen a **go_to_ball**.

En la figura 5.4 se aprecia el diagrama de estados para este comportamiento. Los estados que se utilizan son: **turn** para que el ángulo entre el robot y la pelota sea nulo, **walk** que es el estado inicial y hace que el robot camine en linea recta hasta la pelota. Cuando se ejecute *Go_To_Ball*, se ingresa al estado inicial correspondiente a ***walk*** y desde allí se alterna entre los estados *walk* y *turn* dependiendo de si crece el ángulo entre el robot y la pelota (véase la figura 5.5a). Si el ángulo θ decrece a 0, el robot camina en línea recta hacia la pelota(figura 5.5b).

El estado *turn* tiene la finalidad de hacer girar el robot en la dirección angular en la que se encuentra la pelota respecto de este. Para ello la orden es simple, en la línea 10 la pose objetivo corresponde a la misma posición en la que se encuentra, a excepción de la orientación angular que es la de la pelota, por lo cual el robot girará hacia la izquierda si $\theta > 0$ y a la derecha si $\theta < 0$. Mientras el robot gira, se actualiza el símbolo de entrada para el ángulo de la pelota. Por un asunto de fabricación el robot posee un retardo en la ejecución de las acciones de movimiento, por ende cuando se le indique de que se detenga, el ángulo que se espera tendrá un pequeño error de a lo más 3 grados. Para que sea cercano

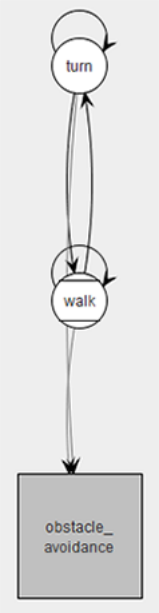
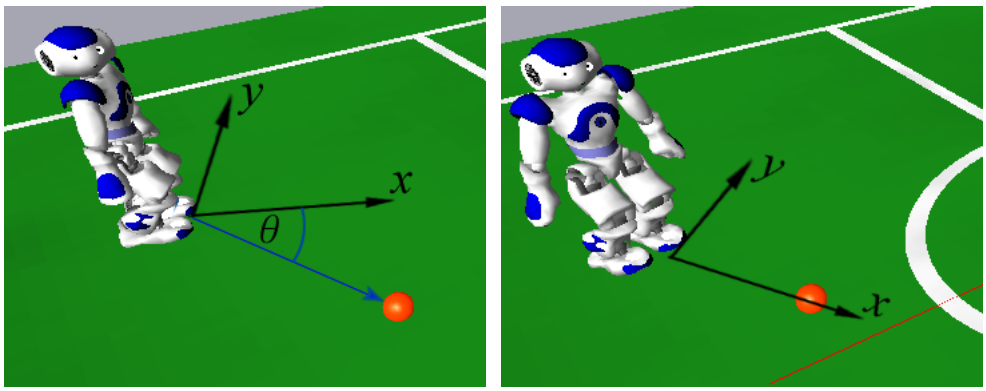


Figura 5.4: Diagrama de la máquina de estado para ejecutar la caminata hacia la pelota.



(a) Etapa de ejecución del estado ***turn***.

(b) Caminar hacia la pelota, después de alinearse. Estado ***walk***.

Figura 5.5: Ejecución de *Go_To_Ball* para sus dos estados.

a 0, se decide que el robot cuando ya esté cerca de los 5° por la derecha o izquierda, se indique que se detenga y con ello se alcanza una orientación de casi 0 grados.

Para el estado *walk* se tiene un procedimiento similar, se verifica que el ángulo de la pelota con respecto al robot no sea mayor de 10°, de este modo se puede retornar al estado *turn* para corregir la orientación angular, dado que por menor que sea el ángulo la distancia

Algoritmo 12 Descripción del estado ***turn***.

```
 1: procedure DECISION
 2:     if ||θ_ball|| < 5 then
 3:         go to walk
 4:     else
 5:         stay
 6:     end if
 7: end procedure
 8:
 9: procedure ACTION
10:     motion ← (0, 0, θ_ball)
11:     head.control_mode ← look_at_ball
12:     do obstacle_avoidance(angle ← θ_ball)
13: end procedure
```

hará que crezca la distancia en el eje y e implique que el robot llega en una posición que dificulta la alineación con la pelota y se reduce a más tiempo para generar la patada. Por lo tanto cuando se le indica que siga la pose $(x_{ball}, 0.5 \cdot y_{ball}, 3.9 \cdot \theta_{ball})$ es una manera de forzarlo a que trate de mantener una trayectoria recta hacia la pelota, este problema es parte de la caminata que posee el robot.

Algoritmo 13 Lógica de funcionamiento del estado ***walk***.

```
 1: procedure DECISION
 2:     if ||θ_ball|| > 10 then
 3:         go to turn
 4:     else
 5:         stay
 6:     end if
 7: end procedure
 8:
 9: procedure ACTION
10:     motion ← (x_ball, 0.5 · y_ball, 3.9 · θ_ball)
11:     head.control_mode ← look_at_ball and look_up_and_down
12:     do obstacle_avoidance(angle ← θ_ball)
13: end procedure
```

En ambos estados descritos se menciona en la línea 12 la ejecución de otra máquina de estados finita, la cual recibe como parámetro el ángulo que tiene respecto a la pelota. Este se encarga de realizar el comportamiento de evasión utilizando los sonares, ya que de esa forma se cubre las posibles situaciones en que el robot se vea obstaculizado por un jugador contrario. Este se verá en la siguiente sección.

Algoritmo 14 Esquema del estado *go_to_ball* dentro de **striker**.

```
 1: procedure DECISION
 2:     if t_ball > 4000 then
 3:         go to search_ball
 4:     else if ||θ_goal|| < 50 and x_ball_global < 1000 and ||y_ball_global|| < 1000 and ||θ_ball|| <
    5 and !obstacle then
 5:         go to bring_ball
 6:     else if d_ball < 500 then
 7:         go to align
 8:     else
 9:         stay
10:     end if
11: end procedure
12:
13: procedure ACTION
14:     do go_to_ball()
15: end procedure
```

El algoritmo 14 describe las transiciones de `go_to_ball` dentro del comportamiento **striker**. En la línea 13 se hace la llamada al comportamiento descrito anteriormente, por ende *go_to_ball* es un comportamiento de bajo nivel. Las líneas 1 a la 13 describen el árbol de decisiones que se ejecuta para salir de *go_to_ball*, en la cual se destaca la línea 4 que establece las siguientes condiciones:

- El ángulo respecto al arco debe ser menor a 30°, ello implica que si mantiene una trayectoria recta, entonces el robot minimiza el tiempo para alinearse y patear.
- Cuando $x_{ball_global} < 1000$ y $\left\|y_{ball_global}\right\| < 1000$ se cumplen, la pelota está lejos del lado contrario.
- Si se cumple $\|\theta_{ball}\| < 5$, implica que el robot camina en línea recta hacia la pelota.
- !*obstacle* indica si no hay obstáculos entre el robot y la pelota, de modo que pueda evadir cuando sea necesario.

Con ello si cada una de esas condiciones es verdadera, entonces se pasa al estado *bring_ball*, de lo contrario puede cumplirse que el robot esté a menos de 500 milímetros de la pelota y con ello se pasa al estado *align*.

5.3. Comportamiento ALIGN

Este comportamiento dispone de un estado en ***striker*** que llama a la máquina de estados finita del mismo nombre **align**.

Algoritmo 15 Estado *align* definido en ***striker***.

```
 1: procedure DECISION
 2:     if t_ball > 4000 then
 3:         go to search_ball
 4:     else if d_ball > 500 then
 5:         go to go_to_ball
 6:     else
 7:         stay
 8:     end if
 9: end procedure
10:
11: procedure ACTION
12:     do align()
13: end procedure
```

El algoritmo 15 define dos condiciones para salir de *align*; si el robot no ha visto la pelota por más de 4 segundos; está a más de 500 milímetros de la pelota, entonces se regresa al estado *go_to_ball*, de lo contrario se mantendrá en el estado actual. Posteriormente en la línea 12 se ejecuta la nueva máquina de estados (figura 5.6).

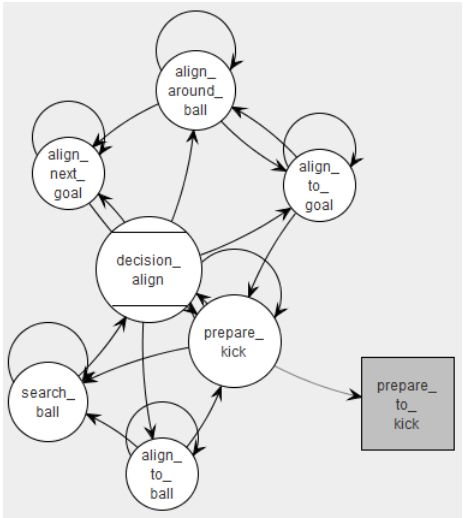
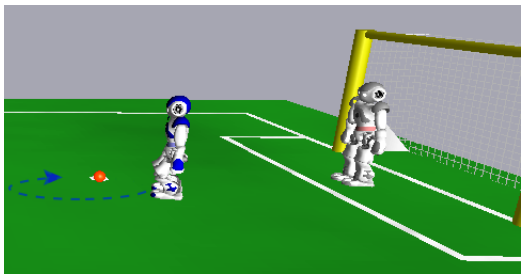


Figura 5.6: Diagrama de estados para la alineación del robot con el arco rival.

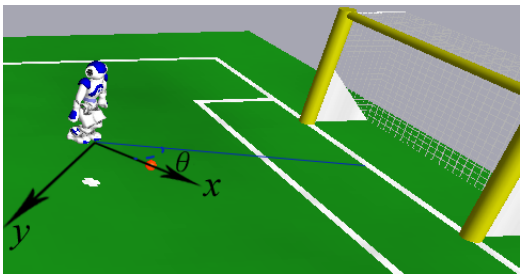
El estado inicial *decision_align* (figura 5.6) se encarga de seleccionar la alineación con el arco. Se definen las siguientes condiciones para tomar una decisión:

- La condición $\left\|\theta_{goal}\right\| > 135$, indica que el robot tiene el arco atrás (figura 5.7a).
- Si se cumple $\left\|\theta_{goal}\right\| < 50$. Por lo tanto se puede alinear el robot con el centro del arco y patear.(Ver figura 5.7b y 5.7d).
- Un jugador contrario impide que se pueda patear hacia el arco, entonces se debe patear hacia un lado libre, por ejemplo en la dirección que apunta la flecha en la figura 5.7c. En este caso el robot se alinea con la pelota para patearla hacia el lado.
- Es importante considerar que la orientación del arco respecto del robot puede cumplir $50 < \left\|\theta_{goal}\right\| < 135$, en cuyo caso se alinea de modo que la orientación esté cercana a los 90 grados y efectuar la patada hacia el lado.

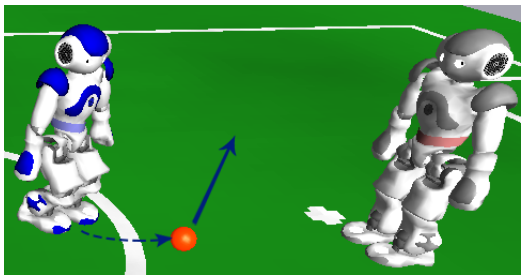
En este comportamiento se requiere del uso de la localización para determinar el ángulo del arco rival respecto del robot y por lo tanto el comportamiento de la cabeza debe permitir mirar el arco y las líneas para disminuir la incertidumbre de la localización.



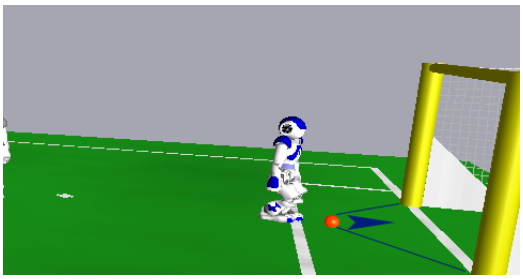
(a) El arco está detrás del robot, $\left\|\theta_{goal}\right\| > 135$



(b) El robot está de frente al arco y $\left\|\theta_{goal}\right\| < 50$.



(c) Obstáculo enfrente de la pelota.



(d) El robot se encuentra dentro del área rival.

Figura 5.7: Situaciones para la alineación con la pelota.

5.3.1. Estado *align_around_ball*

Este estado se encarga de que el robot gire alrededor de la pelota, para que la posición angular del arco contrario respecto del robot sea cercana a cero.

Algoritmo 16 Estado que controla el movimiento de rodear la pelota.

```
1:  procedure DECISION
2:      if ||θ_goal|| < 135 and ||θ_goal|| > 50 then
3:          go to align_next_goal
4:      else
5:          stay
6:      end if
7:  end procedure
8:
9:  procedure ACTION
10:     motion ← (x_ball − 300, y_ball, ||θ_goal|| > 90? θ_goal/3 : 0
11:     head.control_mode ← look_at_ball
12:     kick type ← inwalk_sidewards
13: end procedure
```

Para el algoritmo 16 se establecen dos posibles opciones en la línea 2. La cual es:

- El robot puede acercarse a la pelota con un ángulo respecto al arco $\|\theta_{ball}\| > 135$, por lo cual cuando el robot rodea a la pelota, se llegará a la condición $\|\theta_{ball}\| < 135$, entonces está cercano a la situación de la figura 5.7b y por ende se pasa al estado *align_next_goal*.
- Mientras no se cumpla la condición anterior, continúa rodeando la pelota.

En la línea 12 se indica $x \leftarrow x_{ball} - 300$, es decir, mantiene al robot a una distancia de 300 milímetros de la pelota. Por otro lado, el ángulo que se ingresa corresponde a la proyección con respecto al arco para que gire hacia la derecha o izquierda dependiendo del signo de θ_{ball}.

5.3.2. Estado *align_next_goal*

Alinea el robot en una posición de 90° respecto del arco rival, lo cual permite realizar un golpe lateral. El tiempo de ejecución es menor a patear directamente hacia el arco. Cuando el orientación respecto al arco es mayor a 50°, el tiempo para patear hacia el arco crece y posibilita que otro robot impida ejecutar la patada.

Algoritmo 17 Estado *align_next_goal*.

procedure DECISION
    if $85 < \left\|\theta_{goal}\right\| < 95$ **then**
        go to prepare_kick
    else
        stay
    end if
end procedure

procedure ACTION
    motion $\leftarrow (x_{ball} - 300, y_{ball}, \frac{-\theta_{goal}}{\left\|\theta_{goal}\right\|})$
    head.control_mode $\leftarrow$ look_at_ball
    kick type $\leftarrow$ inwalk_sidewards
end procedure

En este algoritmo se verifica que la orientación no sea mayor a 95° o menor a 85°, es el intervalo para salir del estado y ejecutar la patada lateral. El valor θ_{goal} induce al robot a generar el movimiento alrededor de la pelota.

5.3.3. Estado *align_to_ball*

En este caso se resuelve la situación presentada en la figura 5.7c, en la que se debe patear la pelota hacia el lado. La dirección que se deba escoger depende exclusivamente de la proyección del arco con respecto al robot.

Algoritmo 18 Estado *align_to_ball*

procedure DECISION
    if $d_{ball} < 250$ **and** $ball.was\_seen$ **then**
        go to prepare_kick
    else if $!ball.was\_seen$ **then**
    else
        stay
    end if
end procedure

procedure ACTION
    motion $\leftarrow (x_{ball}, y_{ball}, 3.9 * \theta_{ball})$
    kick type $\leftarrow$ inwalk_sidewards
end procedure

El procedimiento de este estado es simple, cuando el robot está a menos de 250 milímetros de la pelota y la ve constantemente, pasa directamente a *prepare_kick*. En caso que pierda la pelota, se ejecuta el estado *search_ball* que a su vez llama a la máquina de estados que controla el movimiento de la cabeza para buscar la pelota. En la línea 11 se indica que la pelota esté en el eje X del robot, es decir, frente a éste.

5.3.4. Estado *align_to_goal*

Este estado resuelve la situación presentada en la figura 5.7b, en la cual el robot debe alinearse de modo que el ángulo proyectado del centro del arco con respecto al este sea casi nulo. Entonces se define la siguiente rutina:

Algoritmo 19 Estado *align_to_goal*

```
 1: procedure DECISION
 2:     if ||θ_angle|| < 10 and ||y_ball|| < 50 then
 3:         go to prepare_kick
 4:     else
 5:         stay
 6:     end if
 7: end procedure
 8:
 9: procedure ACTION
10:     motion ← (x_ball − 300, y_ball, θ_goal)
11:     head.control_mode ← look_at_ball and look_at_gool_opp
12:     walk type ← d_goal > 3000 ? foward : inwalk_forward
13: end procedure
```

En el algoritmo 19 se verifica que el angulo proyectado del arco respecto al robot sea menor a 10° y la posición de la pelota respecto al eje Y no supere los 50 milímetros. Si la distancia del arco contrario es mayor a 3 metros (línea 12), se opta por la patada *forward*, la cual tiene mayor alcance, de lo contrario se utiliza la patada *inWalk_forward*.

5.4. Comportamiento PREPARE_TO_KICK

Esta máquina de estados, se encarga de generar la alineación del pie del robot respecto a la pelota para patearla. La figura 5.8 muestra el esquema de todos los estados que se encargan de preparar los tipos de patadas existentes en el código del jugador. Cuando la alineación está lista, se pasa a un estado de más bajo nivel que se encarga de ejecutar la patada seleccionada en *prepare_to_kick*.

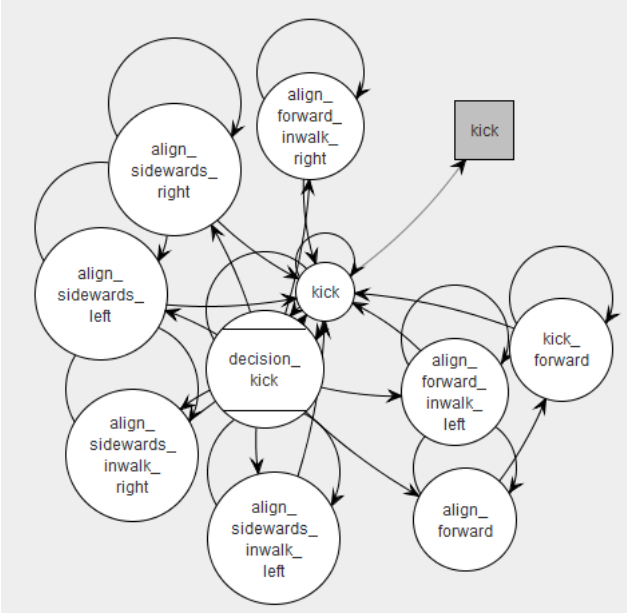


Figura 5.8: Diagrama de estados para *prepare_to_kick*.

Los tipos de patadas que puede realizar el robot son las siguientes:

1. **Kick in walk**: es un comportamiento básico que realiza una patada mientras está caminando. Hay dos patadas: adelante y lateral.
2. **Kick forward**: es un movimiento parametrizado de mayor duración para ejecutarlo, que permite patear desde los 2 metros de distancia del arco.
3. **Kick sidewards**: se ejecuta de la misma forma que la patada anterior. Permite patear hacia el lado con mayor alcance de distancia.

Los parámetros que se asignan en *align* son:

- *kick.type* : es el tipo de patada a ejecutar, la cual puede ser **forward**, **sidewards**, **inwalk_forward** o **inwalk_sidewards**.
- *kick.side* : indica la pierna a utilizar, derecha o izquierda.

El algoritmo 20 describe la alineación para patear la pelota. El parámetro λ desplaza el centro del robot en el eje Y, esto implica que si $\lambda < 0$, entonces la pierna izquierda queda frente a la pelota. Por el contrario, cuando $\lambda > 0$, la pierna derecha quedará frente a la pelota. El parámetro δ controla la distancia entre la pelota y el pie del robot en el eje X.

Algoritmo 20 Estructura base para ejecutar la alineación del robot según la patada escogida

```
1: procedure ACTION
2:     motion ← (x_ball + δ, y_ball + λ, 0)
3:     head.control_mode ← look_at_ball
4: end procedure
```

El algoritmo 21 es el estado inicial, en el cual se selecciona el tipo de alineación del robot con la pelota, para ejecutar la patada especificada en la variable *kick.type*.

Algoritmo 21 Estado inicial que selecciona la alineación para la patada en ***prepare_kick***

```
1: procedure DECISION
2:     if kick.type == sidewards and kick.side == left then
3:         go to align_sidewards_left
4:     else if kick.type == sidewards and kick.side == right then
5:         go to align_sidewards_right
6:     else if kick.type == forward then
7:         go to align_forward
8:     else if kick.type == inwalk_forward and kick.side == right then
9:         go to align_forward_inwalk_right
10:    else if kick.type == inwalk_forward and kick.side == left then
11:        go to align_forward_inwalk_left
12:    else if kick.type == inwalk_sidewards and kick.side == left then
13:        go to align_sidewards_inwalk_left
14:    else if kick.type == inwalk_sidewards and kick.side == right then
15:        go to align_sidewards_inwalk_right
16:    end if
17: end procedure
18:
19: procedure
20:    head.control_mode ← look_at_ball
21: end procedure
```

A continuación se presentan los parámetros (λ, δ), para realizar las diferentes patadas:

- **Kick forward** $\rightarrow \lambda = 50$ y $\delta = -160$
- **Kick inwalk_forward right** $\rightarrow \lambda = -40$ y $\delta = 160$
- **Kick inwalk_forward left** $\rightarrow \lambda = -40$ y $\delta = -160$
- **Kick inwalk_sidewards right** $\rightarrow \lambda = -10$ y $\delta = 150$

- **Kick inwalk_sidewards left** $\rightarrow \lambda = -10$ y $\delta = -155$
- **Kick sidewards right** $\rightarrow \lambda = -10$ y $\delta = 130$
- **Kick sidewards left** $\rightarrow \lambda = -10$ y $\delta = -130$

5.5. Comportamiento *Kick*

A diferencia del resto de los comportamientos mencionados, en esta máquina de estados se utilizó un concepto propio del lenguaje XABSL que es la estructura *common decision*. La utilidad para este caso se debe al hecho de que cada estado actualiza los símbolos de salida de `bike_symbols.xabsl`, para luego el software del robot ejecute la petición.

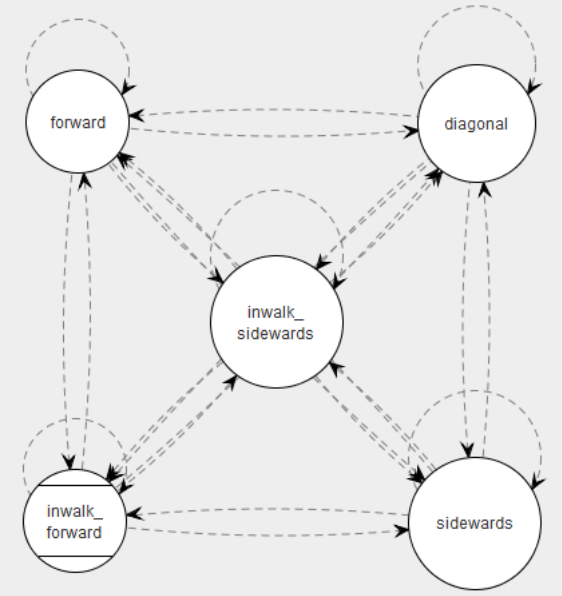


Figura 5.9: Diagrama de estados para ***kick***.

Los estados *inwalk_forward* (algoritmo 22)y *inwalk_sidewards* (algoritmo 23) utilizan los mismos símbolos de salida para el movimiento básico de realizar la patada durante la caminata. Estos símbolos definen la dirección que tendrá la patada, el pie que se usará y la posición de la pelota. Los símbolos de salida utilizados son:

walkKick.type: este símbolo de tipo **enum** recibe el tipo de patada a ejecutar, si es hacia adelante(*forward*), si se patea con la pierna derecha(*sidewards_right*) o con la pierna izquierda(*sidewards_left*).

Algoritmo 22 Estado *inwalk_forward*

```
1: procedure ACTION
2:     motion ← (0,0,0)
3:     head.control_mode ← look_at_ball
4:     walkKick.type ← kick.side == right ? right : left
5:     kickBallPosition.x ← ball.seen.x
6:     kickBallPosition.y ← ball.seen.y
7:     kickTarget.x ← 100
8:     kickTarget.y ← 0
9: end procedure
```

Algoritmo 23 Estado *inwalk_sidewars*

```
1: procedure ACTION
2:     motion ← (0,0,0)
3:     head.control_mode ← look_at_ball
4:     walkKick.type ← kick.side == right ? sidewards_right: sidewards_left
5:     kickBallPosition.x ← ball.seen.x
6:     kickBallPosition.y ← ball.seen.y
7:     kickTarget.x ← 0
8:     kickTarget.y ← kick.side == right ? 100 : -100
9: end procedure
```

kickBallPosition.x: se ingresa la posición en x respecto del robot.

kickBallPosition.y: se ingresa la posición en y respecto del robot.

kickTarget.x: se escribe 1 si se patea hacia adelante o 0 en caso contrario.

kickTarget.y: se puede colocar un número mayor a cero para indicar que la patada es con la derecha o menor a cero para la izquierda. Pero si se usa cero es para la patear hacia adelante.

Para el estado *sidewards* (algoritmo 24) se ejecuta un *special_action* que corresponde a un archivo que contiene la ejecución de un movimiento predefinido. Para ello se utiliza el símbolo de salida **motion.special_action** que define el archivo a leer para ejecutar la patada hacia el lado. En la línea 3 del algoritmo, se selecciona **kick_sidewards_left_nao** o **kick_sidewards_nao**, si **kick.side** es *left* o *right* respectivamente.

Finalmente el estado *forward* realiza la patada más fuerte hacia adelante, sólo se indica en la línea 2 del algoritmo 25 que el tipo de movimiento será patear(*bike*). Con ello se ejecuta un movimiento predefino en el software del robot.

Algoritmo 24 Estado *sidewars*

```
1: procedure ACTION
2:     motion ← special_action;
3:     motion.special_action ← kick.side == left ? kick_sidewards_left_nao :
   kick_sidewards_nao;
4:     head.control_mode ← look_at_ball;
5: end procedure
```

Algoritmo 25 Estado *forward*

```
1: procedure ACTION
2:     motion ← bike;
3:     head.control_mode ← look_at_ball;
4: end procedure
```

5.6. Comportamiento *Search_Ball*

Controla el giro de la cabeza, para enfocar la cámara en algún punto (x, y, z) del campo de juego. En la figura 5.10 se destacan tres estados: **Static_Search**, **Turn** y **Active_Search**.

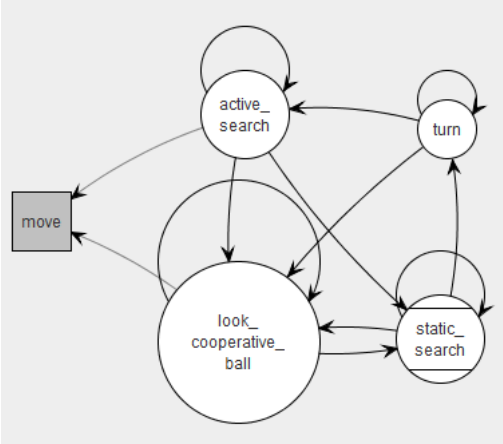
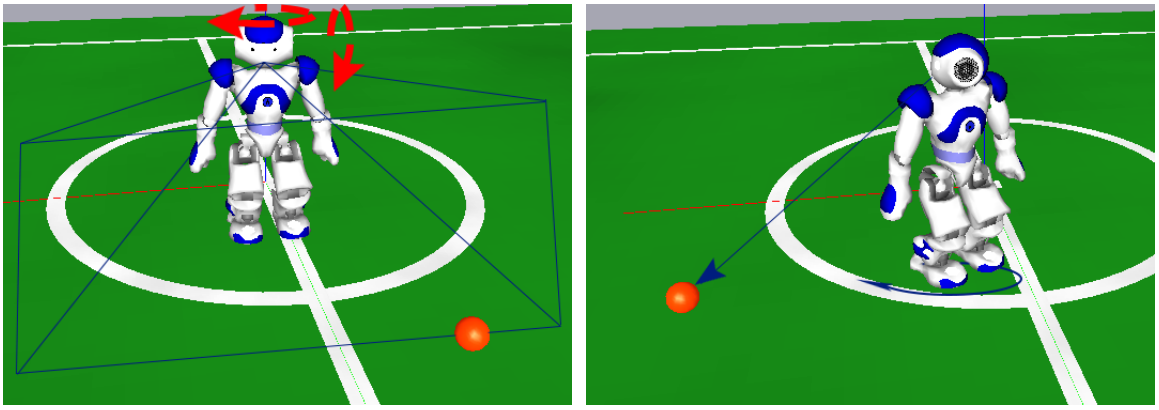


Figura 5.10: Diagrama de estado de la búsqueda de la pelota.

Si el robot no ha visto la pelota por más de 4 segundos, se realiza la búsqueda pasiva (*static_search*), en la cual no se mueve de su posición y usa la cabeza para orientar la cámara hacia distintos puntos en el plano XY del campo. La figura 5.11a muestra el esquema de los movimientos de la cabeza en los ejes cartesianos X-Y-Z. Se programa un movimiento de tipo elipsoidal, para cubrir una mayor área del campo de juego con la cámara.



(a) El robot gira sobre su propia posición.

(b) El robot mueve la cabeza para cubrir un mayor campo visual.

Figura 5.11: Ejemplo de los movimiento de la cabeza según el estado en que se encuentra ejecutando.

Algoritmo 26 Funcionamiento del estado *static_search*

```
 1: procedure DECISION
 2:     if state_time > 3000 then
 3:         go to turn
 4:     else if ball seen by other robot and pose validity > 0.7 then
 5:         go to look_cooperative_ball
 6:     else
 7:         stay
 8:     end if
 9: end procedure
10:
11: procedure ACTION
12:     motion ← (0,0,0)
13:     head.control_mode ← look_ellipse
14: end procedure
```

El algoritmo 26 verifica en la línea 2 si algún compañero de equipo está viendo la pelota, y si está bien localizado, pasa al estado *look_cooperative_ball*, el cual no forma parte de la solución propuesta. Si no se cumple esa condición y no ve la pelota por más de 3 segundos, pasa al estado `turn`. El movimiento elipsoidal se ejecuta al asignar al símbolo interno *head.control_mode* el valor *look_ellipse*.

Para agregar un nuevo movimiento de la cabeza, se debe crear la máquina de estados correspondiente, crear un estado en *head_control.xabsl* que llama al comportamiento, y agregar dentro del símbolo **head.control_mode** el identificador de este nuevo estado. Se crea la máquina de estados *look_ellipse* que dispone del siguiente algoritmo(27).

Algoritmo 27 Esquema del control del movimiento elipsoidal.

```
1:  procedure DECISION
2:      stay
3:  end procedure
4:
5:  procedure ACTION
6:      head.mode ← pan_tilt;
7:      head.pan ← panAngle ∗ cos(360·t / T)
8:      head.tilt ← tiltAngle ∗ sin(360·t / T)
9:      head.speed ← 100
10: end procedure
```

El estado *initial_Position* (figura 5.12) sitúa la cabeza en $(0,0)$ en el plano XY, después se ejecuta el movimiento repetidamente. Para la construcción de este movimiento, se definieron tres constantes que definen el tamaño de la elipse y el período de ejecución:

tiltAngle: es el ángulo de apertura vertical de la cabeza. Se estableció en $20°$.

panAngle: corresponde al ángulo de apertura horizontal de la cabeza. Su valor es $60°$.

period_ellipse: es el período para realizar el movimiento de la elipse. El tiempo es de 4 segundos. Variando este parámetro se regula la velocidad de ejecución de la elipse.

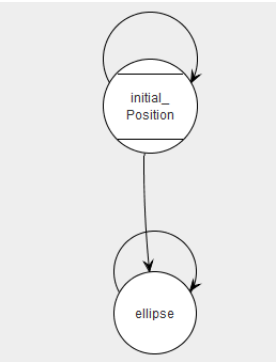


Figura 5.12: Diagrama de estados para *look_ellipse*.

El estado *turn* (algoritmo 28) se encarga de girar el robot durante un tiempo de 9 segundos, posterior a ello se pasa a buscar la pelota de forma activa. Los otros estados corresponden a comportamientos que existían previo a la realización de este trabajo, pero se utilizan para el desarrollo del comportamiento a un nivel más complejo de la búsqueda de la pelota.

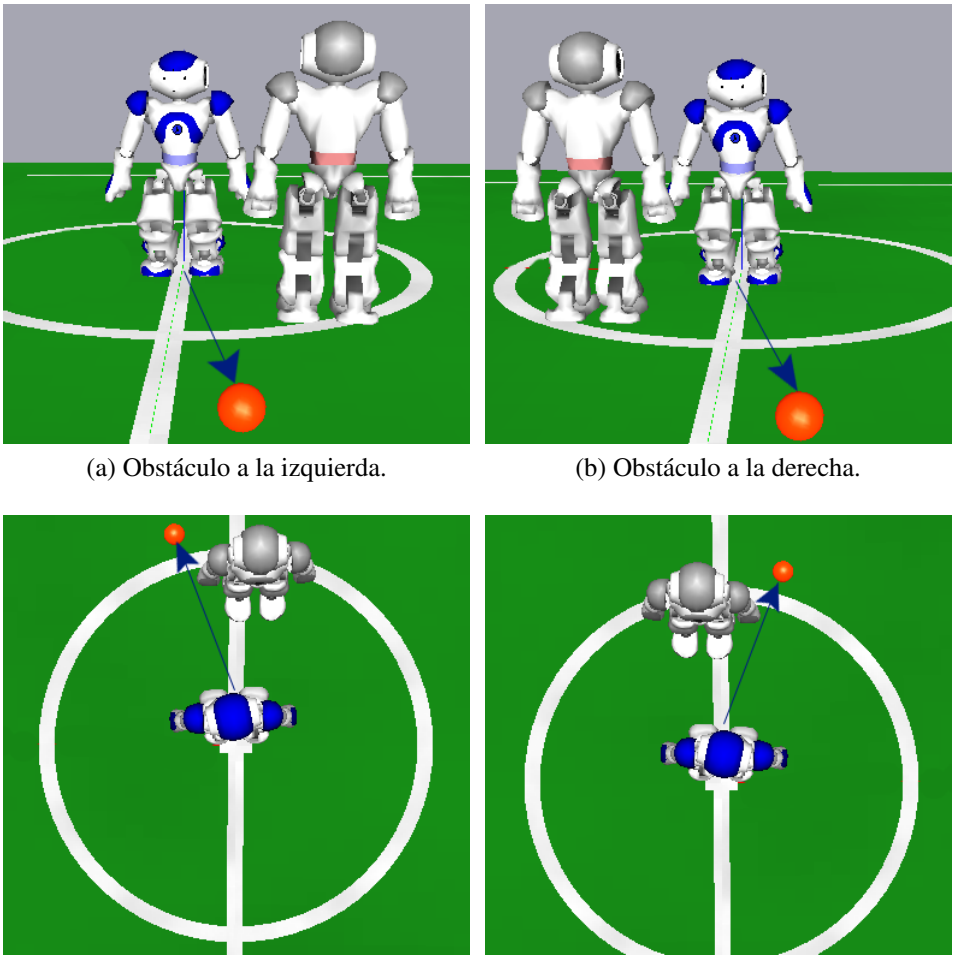
Algoritmo 28 Estado *turn*

```
1:  procedure DECISION
2:      if state_time > 9000 then
3:          go to active_search
4:      else if ballseenbyotherrobot and posevalidity > 0.7 then
5:          go to look_cooperative_ball
6:      else
7:          stay
8:      end if
9:  end procedure
10:
11: procedure ACTION
12:     motion ← (0, 0, θ_ball)
13:     head.control_mode ← look_up_and_down
14: end procedure
```

5.7. Comportamiento para Evadir Obstáculos

En este apartado se describe el comportamiento para evadir a otros robots basado en las mediciones realizadas con el sonar. Para ello se utilizan los símbolos de la sección 4.5 para la toma decisiones de la máquina de estados. Este comportamiento considera los siguientes casos:

- Cuando camina hacia la pelota, un jugador contrario puede estar aproximándose por la izquierda del robot, entonces deberá evadir hacia la derecha. La figura 5.13a muestra la dirección que lleva el robot hacia la pelota.
- El ejemplo de la figura 5.13b representa el caso en que el robot se encuentra con un obstáculo por la derecha, la flecha indica la dirección que lleva el robot hacia la pelota, por lo cual debe evadir hacia la izquierda para no colisionar.
- Cuando un robot se interpone en el centro del camino hacia la pelota (figura 5.13c y 5.13d), se evade según la posición angular de la pelota respecto del robot.



(a) Obstáculo a la izquierda.

(b) Obstáculo a la derecha.

(c) Obstáculo al centro y la pelota a la izquierda.

(d) Obstáculo al centro y la pelota a la derecha.

Figura 5.13: Situaciones de obstáculo por parte de los jugadores contrarios.

El estado principal es *sonar* (figura 5.14), en el cual se realiza la toma de decisiones para elegir el movimiento de evasión. Posteriormente los estados restantes se encargan de realizar los movimientos y retornar a *sonar*, para reaccionar ante otro obstáculo. El algoritmo 29 se puede dividir en tres partes principales:

1. Las líneas 2 a la 7 definen la decisión para evadir hacia la derecha o izquierda en función de la distancia del obstáculo en el cono central. Existe un obstáculo, si la distancia es menor o igual a 600 milímetros, el parámetro *@angle* corresponde a la dirección angular de la pelota respecto del robot. En base a eso, en la línea 3 se selecciona la dirección para evadir.

2. Las líneas 8 a la 12 verifican si el obstáculo está a la izquierda o derecha del robot y con ello seleccionan la dirección contraria para evadir.

3. Por último las líneas 12 a la 19 verifican si existen obstáculos que sean detectados con la colisión de los brazos o los bumpers en los pie del robot. Para este trabajo, esos símbolos ya habían sido creados.

Algoritmo 29 Control de las acciones para realizar la evasión

```
1:  procedure DECISION
2:      if obstacle.center.distance < 600 then
3:          if @angle < 0 then
4:              go to walk_right
5:          else
6:              go to walk_left
7:          end if
8:      else if obstacle.left.distance < 600 then
9:          go to walk_right
10:     else if obstacle.right.distance < 600 then
11:         go to walk_left
12:     else if obstacle.arm_left then
13:         go to walk_right_arm
14:     else if obstacle.arm_right then
15:         go to walk_left_arm
16:     else if obstacle.footbumper_left then
17:         go to walk_right_bumper
18:     else if obstacle.footbumper_right then
19:         go to walk_left_bumper
20:     else
21:         stay
22:     end if
23: end procedure
```

Las detecciones que se realicen con los sensores táctiles de los pies o lo brazos permiten mejorar la evasión con los sonares, dado que estos dejan de ser efectivos cuando la distancia respecto del obstáculo sea menor a los 25 centímetros. Y asegura un movimiento que impida que el robot sea penalizado por *pushing*.

Los movimientos que se realizan para evadir con los sonares, consisten en que el robot camine en diagonal en la dirección donde esté libre de obstáculos, se establece un control por velocidad para los ejes X-Y, en el cual la velocidad en el eje Y será positiva o negativa según sea hacia la izquierda o derecha respectivamente. Cuando existe una colisión con los

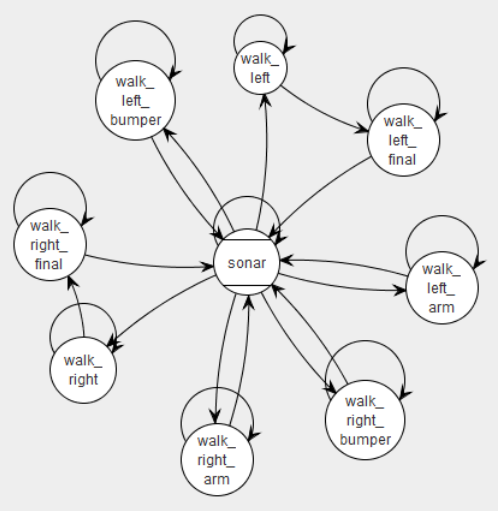


Figura 5.14: Esquema de la máquina de estados para la evasión.

brazos, dependiendo de con cual brazo se detecte la colisión, se evadirá según la siguiente regla:

- Si se detecta un golpe en el brazo izquierdo, el movimiento será hacia la derecha para dejar de hacer contacto con el brazo.
- Si el contacto es por el lado del brazo derecho, entonces el movimiento es a la izquierda.

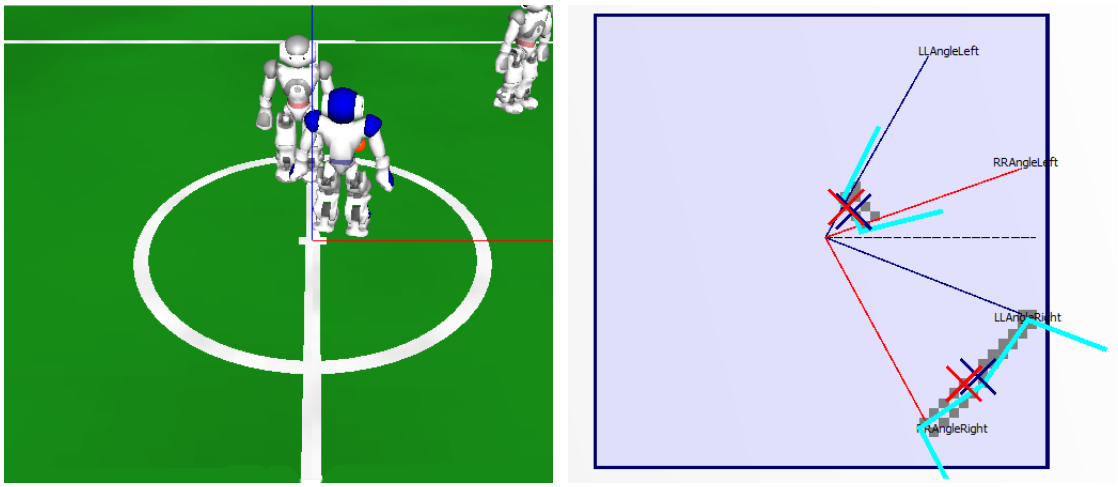
Por otro lado con los sensores táctiles se recurre a la opción de moverse hacia atrás, dado que la colisión indica que estará enfrente del robot y por la proximidad impide que pueda evadirlo sin que pierda el equilibrio.

Capítulo 6

Discusión de resultados, simulaciones y pruebas en el robot

6.1. Prueba del Control de los Sonares

Las pruebas del funcionamiento fueron realizadas en el simulador (SimRobot) conectado al robot para realizar la depuración del código de *USControl* y *ObstacleSymbols*. La figura 6.1 muestra un ejemplo de la detección con sonares en el simulador.



(a) Situación de dos robots enfrente del jugador. (b) Resultados de la detección en la grilla.

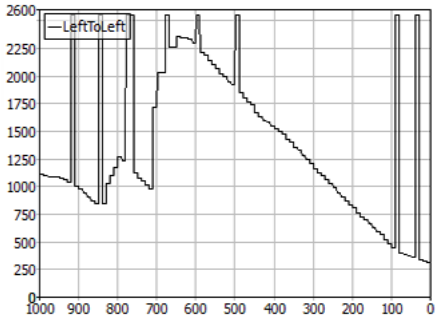
Figura 6.1: Resultado de la operación del sonar en el modo 68.

En la figura 6.1b se muestra el concepto de la grilla de obstáculos, según lo visto en la sección 4.3.2, el tamaño de los conos se establece en el archivo `usCalibration.cfg`. Por lo cual después de realizada la calibración del sonar para este modo, se obtuvieron los siguientes intervalos de detección:

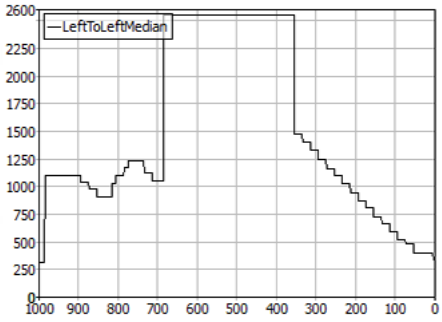
Cono Izquierdo	Cono Central	Cono Derecho
(60,20)	(20,-20)	(-20,-60)

Tabla 6.1: Resultado de la calibración para la detección en el modo 68.

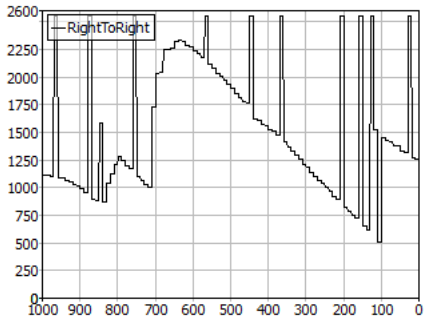
La tabla 6.1 muestra el intervalo de apertura del cono al principio y luego el intervalo de cierre. El hecho de que el primer valor sea mayor que el segundo se relaciona con el círculo de calibración de la figura 4.9. Estos datos son ingresados en el archivo de configuración correspondiente y se genera la grilla de la figura 6.1b.



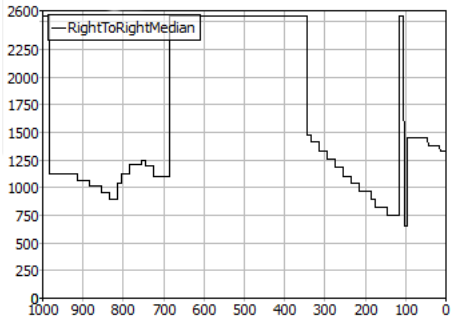
(a) Lectura del sonar para el lado izquierdo.



(b) Lecturas filtradas con mediana.



(c) Lectura del sonar para el lado derecho.



(d) Lectura filtrada con mediana.

Figura 6.2: Lecturas de los sonares para el ejemplo de la figura 6.1.

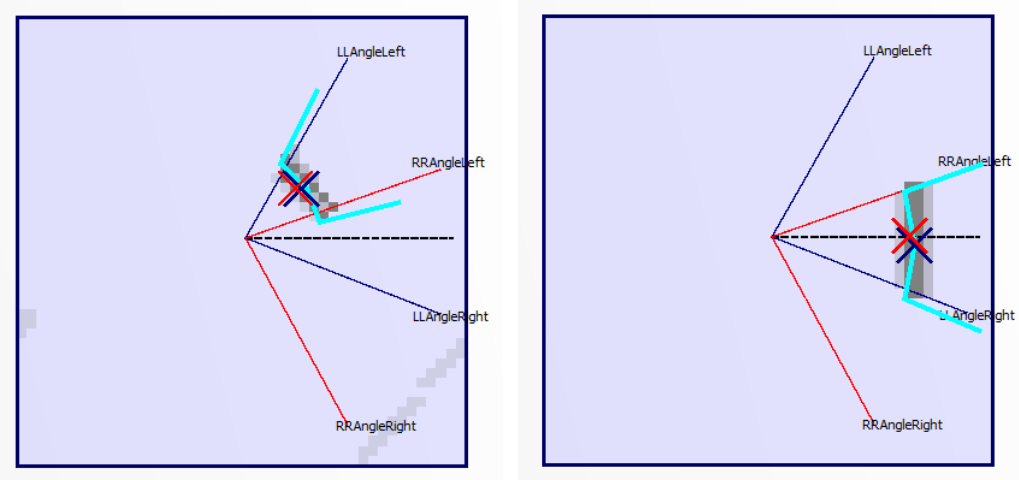
Los resultados de la figura 6.2 muestran que el filtro elimina todo el ruido presente en las lecturas realizadas con el sonar. Para ello se utilizó una ventana de tamaño 60. Este valor

se obtuvo después de aumentar gradualmente el tamaño de la ventana para ir obteniendo una curva de mediciones que presentara el mínimo de fluctuaciones. Por otro parte el parámetro **maxScanRange** se estableció en 1500 milímetros, lo que muestra que para mediciones superiores a este valor, pasan a ser 2550 milímetros después de que pasan por el filtro de mediana. En la figura 6.2d, en el intervalo $[400, 500]$ milisegundos, se aprecia el efecto de limitar el rango de escaneo. Esta es la forma en que se puede limitar por software la detección mediante ultrasonido.

El rango máximo que posee el robot no es suficiente, dado que un jugador puede ser un obstáculo si éste se encuentra a una distancia menor a 1 metro. Otra razón para limitar de esta forma, es que impide que un objeto detectado sobre esta distancia sea ingresado a la grilla. Lo importante dentro de la detección de los objetos es la reactividad que posee para detectar nuevos objetos o para eliminar los que ya dejaron de ser detectados por el sonar. En la sección 4.3.2 se explicó la función de los parámetros que se encuentran en el archivo `usObstacleGrid.cfg`, de los cuales destacan dos valores que le dan mayor velocidad de actualización del estado de cada elemento de la grilla. Para el caso de **cellFreeInterval** los valores que puede tomar son sobre los 33 milisegundos, debido a que un valor inferior a este impide que se almacene información en la grilla. Esto se debe al período de trabajo de **Cognition** que es de aproximadamente 33 milisegundos, por ende ingresar un tiempo menor a este impide que se procese la información, porque siempre se estará eliminando la información de la celda. Para efectos de un buen tiempo de vida de las celdas se opta por tomar un tiempo cercano a los 50 milisengundos, lo que en la práctica le da un tiempo de vida cercano a los 20 milisegundos a cada celda, con ello se consigue una actualización de los datos adecuada para detectar eventos como por ejemplo el movimiento de un jugador cerca del robot. Por otra parte, el factor **agingFactorOnFreeDrawing** es un ponderador que disminuye el estado de una celda, mientras más grande sea, es más rápido la disminución del estado de la celda con cada actualización que se haga.

En la figura 6.3 se muestra la transición de un objeto que va desde el costado izquierdo hacia la derecha. Es importante considerar que en el partido, los obstáculos que estarán en movimiento, por ende es muy importante que la información en la grilla se actualice rápido. Por lo tanto, para efectos de este trabajo, los tiempos definidos se consideraron en base a las pruebas realizadas en el robot, en el cual existe un retardo natural entre la ejecución de un comando y las lecturas de los sensores.

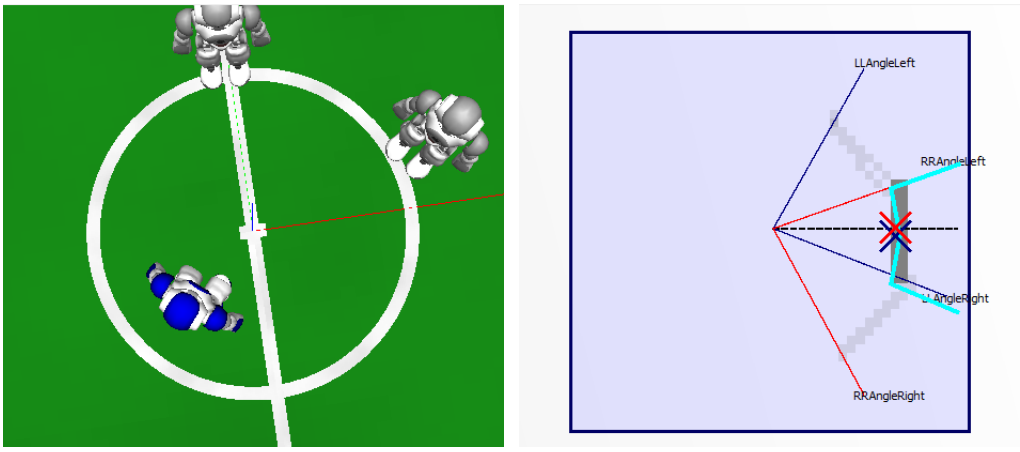
Una de las limitantes que presenta el sistema de detección de obstáculos desarrollado, es cuando se presentan dos obstáculos, uno a la derecha y el otro a la izquierda y están a la misma distancia o difieran en menos de 100 milímetros. Lo cual será identificado como un obstáculo en el centro. En la figura 6.4a se ejemplifica esta situación, en este caso cada robot contrario es detectado por el cono izquierdo y derecho exclusivamente. La distancia es menor al factor de agrupamiento, por ende se tiende a clasificar a este como un solo obstáculo, dado que no se tiene un sonar en el centro que permita descartar esta situación.



(a) Objeto detectado por la izquierda. (b) El objeto vuelve a aparecer en el centro.

Figura 6.3: Ejemplo de la actualización de la grilla.

Una posible solución es recurrir a los demás ecos para que se determine si existe algo en el cono central, verificando si existen mediciones iguales entre los ecos. Esta situación no se presenta regularmente en el partido, queda como futura mejora de la solución desarrollada.



(a) Dos obstáculos a la misma distancia en la derecha e izquierda. (b) Detección en la grilla.

Figura 6.4: Error en la detección del obstáculo.

6.2. Evaluación del Comportamiento Striker

La evaluación del desempeño del comportamiento es más complicada de expresar en datos. Cabe mencionar que la evasión de obstáculos depende en parte de la velocidad del robot, considerando la alta reactividad de la grilla, se puede realizar una medición de la distancia a la que se frena el robot ante un obstáculo.

Prueba	Distancia
1	551.54
2	442.94
3	510.88
4	381.84
5	551.54
6	426.38

Tabla 6.2: Mediciones de la distancia a la que frena el robot respecto del obstáculo.

La tabla 6.2 muestra 6 mediciones tomadas para la detección de un obstáculo puesto a la izquierda y el robot caminando a un 80 % de su velocidad. Para el comportamiento se utiliza una distancia de 600 milímetros para indicarle al robot que está frente a un obstáculo, de acuerdo a la tabla presentada el promedio es de 477.52 milímetros para la distancia en la cual se detiene el robot. Se debe considerar que la detección de los sonares depende mucho de ciertos factores como el área en la cual incide la onda sonora y el tipo de material. Además por la geometría del robot, la detección no es buena cuando el objeto está en los límites de detección de los conos, eso provoca variabilidad en la posición del obstáculo, hace que el comportamiento de evasión empeore.

El comportamiento de ir hacia la pelota, no tiene fallas en el sentido de que la forma más óptima para moverse es siempre ir en línea recta, lo que importa es el tiempo de reacción que tiene para moverse. Entonces los movimientos que se definen para el jugador tienen como clave que sean rápidos, esto está influenciado por:

a) El la velocidad de procesamiento de las imágenes captadas por la cámara.

b) La complejidad de las decisiones que realiza el robot.

c) La velocidad de respuesta de los actuadores ante los cambios en las solicitudes que se les hace.

En el caso del comportamiento de *Align*, éste puede fallar en situaciones inesperadas en que el robot se queda en un estado dado, no cambian las condiciones definidas en este, para

pasar a otro estado. Esto son eventos que no siempre se pueden prever, sino hasta realizar las primeras simulaciones. La primera versión desarrollada permite cubrir los casos más básicos dentro de un juego como son:

- Alinearse respecto al arco para patear.
- Darse vuelta para patear cuando se está orientado en la dirección del propio arco.
- Despejar la pelota en caso que otro robot impida patear hacia el arco.

Este último es el más simple en realizar, pero a la vez el más complejo para tomar la decisión. Actualmente como primera versión se dejó que patee la pelota conforme el ángulo que tiene respecto al arco, pero esto no es suficiente dado que se debe considerar si el robot está próximo a las líneas que delimitan el área de juego, conocer la posición del resto de sus compañeros y realizar un pase en aquella dirección. Esta labor queda para un desarrollo más complejo del comportamiento del jugador.

Capítulo 7

Conclusiones

La detección de obstáculos en el juego, permite al robot reducir las colisiones y por ende las penalizaciones por golpes al oponente. Las mejoras hechas al jugador tienen dos aspectos para la detección de obstáculos en el juego: *control de los sonares* y *comportamiento de evasión de obstáculos*.

El funcionamiento de los sonares no es completamente eficiente en el juego. La onda de ultrasonido emitida por otro robot genera interferencia, esta se expresa en un error que puede inducir a considerar al objeto más cerca de lo que puede estar en realidad. Durante las pruebas, se detectó la interferencia, cuando dos robots estaban frente a frente, en cuyo caso es máxima. La línea de tiempo del funcionamiento de los sonares (capítulo 4), reduce el tiempo en que se presenta la interferencia. Este problema surge debido a que todos los robots utilizan la misma frecuencia de ultrasonido, de modo que la solución es que a cada robot pudiese configurarse la frecuencia de la onda emitida, lo que en la actualidad está limitado por el hardware disponible en el robot. El comportamiento diseñado está limitado por la información de la que dispone. Se mencionó en la sección 4.5, se pueden obtener como máximo dos obstáculos. Al crearse el cono central por la intersección de los conos izquierdo y derecho, causa errores en la generación del mapa de obstáculos, debido a que la intersección se realiza comparando la distancia del obstáculo detectado por el cada uno de los conos, cuando la diferencia es menor a 10 centímetros se ingresa como un objeto que está en el centro. Esta lógica de programación falla cuando se presentan dos obstáculos a la misma distancia. A lo cual la evasión será ineficiente, pero considerando la robustez de ésta, se llegará a evadir en un tiempo mayor.

Se puede mejorar el comportamiento de la evasión. Eliminando los errores en la detección de obstáculos en el centro del robot. Utilizar los ecos restantes de cada sonar, elimina la ambigüedad en la detección de un obstáculo en el cono central. Para ello, se deben

modificar las clases que controlan la grilla y `NaoProvider.cpp`. En la sección 4.1.1 se mencionó el modo 70, que permite captar con el receptor izquierdo la onda que se emite con el receptor derecho, esto crea un nuevo campo de detección más acotado, pero combinándolo con el modo de operación 68, se puede crear hasta 8 conos de detección. Al crear más conos, estos reducen el ángulo de apertura, pero en los bordes de cada cono la detección no es constante. Aumentar los conos de detección no es eficiente, puesto que el ángulo de detección varía cuando el objeto está cerca del borde del cono.

Para efectos del funcionamiento en un partido, el sistema es reactivo, que es lo fundamental. Si bien no puede entregar con presión la posición angular del obstáculo, es más que suficiente para el comportamiento de la evasión. Los parámetros de los archivos de configuración de la grilla y el modelo de obstáculos son de gran importancia, ya que con ello se puede tener una grilla que reacciona más lento a los cambios de movimiento o bien lo haga más rápido. Variar estos parámetros definirán el desempeño del comportamiento de la evasión.

El desarrollo de los comportamientos para buscar la pelota, caminar hacia esta, evadir y alinearse entorno a un punto son más simples de lo que se realizaba antes de iniciar este trabajo. XABSL dispone de una sintaxis simple y efectiva de controlar un robot. La capacidad de crear máquinas de estado que pudieran recibir parámetros, facilitan la creación de comportamientos adaptables, es decir, el robot podrá ejecutar acciones que varían por un parámetro de entrada en el código, lo cual implica hacer más eficiente el trabajo.

Un aspecto importante de XABSL es la capacidad de depurar el funcionamiento del código por medio de un simulador o programa que acceda al robot, pero ello implica desarrollar las herramientas para integrarlo. El grafo de los estados y el editor de este entorno facilitan mucho el trabajo de diseño. Por otro lado, sólo los símbolos de salida y entrada son accesibles para su depuración durante el funcionamiento del código.

XABSL permite tener dos máquinas de estados ejecutándose en paralelo, por ejemplo los movimientos de locomoción y de la cabeza. Además el permitir que los estados no siempre tengan definido un árbol de decisiones, se pueden crear estados que ejecutan repetitivamente un movimiento. Puede ser un inconveniente que se deben crear símbolos de salida y entrada para la nueva información que se quiera vincular o bien con el nuevo comportamiento básico que se defina en *Motion*. La utilización de XABSL permite construir comportamientos más complejos que los presentados, por ejemplo crear jugadas como pases entre los robots, despejar la pelota en la dirección donde se encuentre el jugador más cercano dentro del mismo equipo, etc.

A partir del comportamiento diseñado, se puede continuar el desarrollo del jugador. Considerar nuevos casos para *Align*, como por ejemplo que el robot pueda utilizar la pose del jugador contrario para despejar la pelota. En general las acciones de buscar, ir y patear la pelota pueden realizarse de diferentes formas, en este caso mediante este lenguaje se puede desarrollar tipos de jugador que obedezcan a ciertas condiciones del juego como por ejemplo:

i. El equipo contrario dispone de pocos jugadores, se puede utilizar un **Striker** que realice patadas desde lejos, esta decisión sería de acuerdo el número de jugadores contrarios.

ii. Cuando el partido sea parejo en cantidad de jugadores, se puede definir un delantero que espere cerca del arco enemigo.

iii. Construir un comportamiento de contra-ataque para integrarlo dentro del Striker, dado que éste no tiene la capacidad de reaccionar cuando la pelota es movida por el equipo contrario.

7.1. Trabajos Futuros

El control de los sonares utiliza el primer eco, por lo cual queda como tarea ingresar los otros ecos como parte de la información de la grilla, para mejorar la asignación del obstáculo en alguno de los conos disponibles. Manteniendo el modo de operación 68, el trabajo va en habilitar los otros ecos y filtrar las mediciones y luego establecer una técnica para superponer los conos para encontrar los obstáculos en el cono central.

La simplicidad para escribir máquinas de estados finitas en XABSL, permite desarrollar comportamientos más complejos. Los robots se pueden comunicar, por lo cual se puede mejorar el Striker considerando otras variables de decisión como lo pueden ser los obstáculos de otro robot, para mejorar el mapa global, desarrollar los pases que es un elemento decisivo en un juego de cooperación como el fútbol.

Bibliografía

[1] SMACH http://www.ros.org/wiki/smach [Consulta: 15 de Marzo de 2013]

[2] XABSL http://www.xabsl.de/ [Consulta: 22 de Abril de 2013]

[3] Michael A. Ainslie. Springer, Edición 2010. Principles of Sonar Performance Modeling.

[4] Roland Siegwart, Illah R. Nourbakhsh, Davide Scaramuzza. Introduction to Autonomous Mobile Robots. Second edition, Intelligent Robotics and Autonomous Agents series The MIT Press, Massachusetts Institute of Technology Cambridge, Massachusetts 02142 ISBN 0-262-01535-8, 2004.

[5] M. Risler. Behavior Control for Single and Multiple Autonomous Agents Based on Hierarchical Finite State Machines. Fortschritt-Berichte VDI Reihe 10: Informatik/Kommunikation, No. 801, Darmstadt, VDI-Verlag, 2009.

[6] M. Risler and O. von Stryk. Formal Behavior Specification of Multi-Robot Systems Using Hierarchical State Machines in XABSL. In AAMAS08-Workshop on Formal Models and Methods for Multi-Robot Systems, Estoril, Portugal, 2008.

[7] M. Lötzsch, M. Risler, and M. Jüngel. XABSL - A Pragmatic Approach to Behavior Engineering. In Proceedings of IEEE/RSJ International Conference of Intelligent Robots and Systems (IROS), pages 5124-5129, Beijing, China, 2006.

[8] M. Lötzsch. XABSL - A Behavior Engineering System for Autonomous Agents. Diploma thesis. Humboldt-Universität zu Berlin, 2004.

[9] M. Lötzsch, J. Bach, H.-D. Burkhard, and M. Jüngel. Designing Agent Behavior with the Extensible Agent Behavior Specification Language XABSL. In D. Polani, B. Browning, and A. Bonarini, editors, RoboCup 2003: Robot Soccer World Cup VII, volume 3020 of Lecture Notes in Artificial Intelligence, pages 114-124, Padova, Italy, 2004. Springer.

Apéndice A

Código para controlar sonares

USControl.cpp

```
1  MODIFY("parameters:USControl", parameters);
2    if(SystemCall::getTimeSince(lastSendTime) > parameters.
       sendInterval)
3    {
4            usRequest.sendMode = -1;
5            usRequest.receiveMode = -1;
6            lastSendTime = SystemCall::getCurrentSystemTime();
7    }
8    else if(SystemCall::getTimeSince(lastSendTime) > parameters.
       ignoreAfterSwitchMode)
9    {
10           usRequest.sendMode = parameters.modes[0];
11           usRequest.receiveMode = parameters.modes[0];
12   }
13
14 }
```

NaoProvider.cpp Extracto del código para leer los datos de los sonares.

```
1  SensorData::UsActuatorMode mode = SensorData::
     leftRightToLeftRightP;
2
3    if(theUSRequest.receiveMode != -1)
4    {
5      for(int i = SensorData::usL; i < SensorData::usREnd; ++i)
6      {
7        float data = sensors[i - SensorData::usL + lUsSensor];
8        sensorData.data[i] = data ? data * 1000.f : 2550.f;
```

```
9       }
10      sensorData.usTimeStamp = theJointData.timeStamp;
11
12      //the second bit in receiveMode describes which actuator is
           sending
13      switch(theUSRequest.receiveMode)
14      {
15          case 68:
16                  mode = SensorData::leftRightToLeftRightP;
17                  break;
18          case 70:
19                  mode = SensorData::leftRightToRightLeftP;
20                  break;
21      default:
22        ASSERT(false);
23      }
24    }
25    else
26            mode = SensorData::none;
27
28    sensorData.usActuatorMode = mode;
29
30    PLOT("module:NaoProvider:usLeft", sensorData.data[SensorData::
         usL]);
31    PLOT("module:NaoProvider:usRight", sensorData.data[SensorData::
         usR]);
32    PLOT("module:NaoProvider:usLeft1", sensorData.data[SensorData::
         usL1]);
33    PLOT("module:NaoProvider:usRight1", sensorData.data[SensorData::
         usR1]);
34    PLOT("module:NaoProvider:usLeft2", sensorData.data[SensorData::
         usL2]);
35    PLOT("module:NaoProvider:usRight2", sensorData.data[SensorData::
         usR2]);
36    PLOT("module:NaoProvider:usReceiveMode", (float)theUSRequest.
         receiveMode);
37    PLOT("module:NaoProvider:usSendMode", (float)theUSRequest.
         sendMode);
38    PLOT("module:NaoProvider:usActuatorMode", (float)sensorData.
         usActuatorMode);
```

SensorFilter.cpp

```
1 if(theSensorData.usActuatorMode == SensorData::
     leftRightToLeftRightP)
```

```
2   {
3                   float data = 0;
4                   float dataEco = 0;
5                   for(int i = 0; i < 3; i++)
6                   {
7                           data = filteredSensorData.data[SensorData
                               ::usL + i];
8                           llBuffer[i].add(data);
9                   }
10                  data = getMedianEco(llBuffer[0]);
11                  dataEco = getMedianEco(llBuffer[1]);
12
13                  if (isUsSensorDataValid(data))
14                  {
15                          if(abs(filteredSensorData.leftToLeft -
                               data) > errorMeasure)
16                                  filteredSensorData.leftToLeft =
                                       data;
17                  }
18                  else
19                          filteredSensorData.leftToLeft = 2550;
20
21                  if (isUsSensorDataValid(dataEco))
22                          filteredSensorData.leftToLeft1 = dataEco;
23                  else
24                          filteredSensorData.leftToLeft1 = 2550;
25
26                  // Detecting with right sonar
27                  for(int i = 0; i < 3; i++)
28                  {
29                          data = filteredSensorData.data[SensorData
                               ::usR + i];
30                          rrBuffer[i].add(data);
31                  }
32                  data = getMedianEco(rrBuffer[0]);
33                  dataEco = getMedianEco(rrBuffer[1]);
34
35                  if (isUsSensorDataValid(data))
36                  {
37                          if(abs(filteredSensorData.rightToRight -
                               data) > errorMeasure)
38                                  filteredSensorData.rightToRight = data
                                       ;
39                  }
```

```
40                  else
41                          filteredSensorData.rightToRight = 2550;
42
43                  if (isUsSensorDataValid(dataEco))
44                          filteredSensorData.rightToRight1 = dataEco
                               ;
45                  else
46                          filteredSensorData.rightToRight1 = 2550;
47
48
49          PLOT("module:SensorFilter:RightToRight", theSensorData.
                data[SensorData::usR]);
50          PLOT("module:SensorFilter:LeftToLeft", theSensorData.data[
                SensorData::usL]);
51          PLOT("module:SensorFilter:RightToRight1", theSensorData.
                data[SensorData::usR1]);
52          PLOT("module:SensorFilter:LeftToLeft1", theSensorData.data
                [SensorData::usL1]);
53          PLOT("module:SensorFilter:LeftToLeftMedian",
                filteredSensorData.leftToLeft);
54          PLOT("module:SensorFilter:RightToRightMedian",
                filteredSensorData.rightToRight);
55          PLOT("module:SensorFilter:LeftToLeft1Median",
                filteredSensorData.leftToLeft1);
56          PLOT("module:SensorFilter:RightToRight1Median",
                filteredSensorData.rightToRight1);
57  }
```

ObstacleSymbols.cpp

```
1 engine.registerDecimalInputSymbol("obstacle.left.distance", this,
     &BH2011ObstacleSymbols::getDistanceLeftObstacle);
2   engine.registerDecimalInputSymbol("obstacle.right.distance",
       this, &BH2011ObstacleSymbols::getDistanceRightObstacle);
3   engine.registerDecimalInputSymbol("obstacle.center.distance",
       this, &BH2011ObstacleSymbols::getDistanceCenterObstacle);
4   engine.registerDecimalInputSymbol("obstacle.left.angle", this, &
       BH2011ObstacleSymbols::getAngleLeftObstacle);
5   engine.registerDecimalInputSymbol("obstacle.right.angle", this,
       &BH2011ObstacleSymbols::getAngleRightObstacle);
6   engine.registerDecimalInputSymbol("obstacle.center.angle", this,
        &BH2011ObstacleSymbols::getAngleCenterObstacle);
7   engine.registerDecimalInputSymbol("obstacle.arc_center", this, &
       BH2011ObstacleSymbols::getArcCenter);
```

```
8   engine.registerDecimalInputSymbol("obstacle.arc_right", this, &
       BH2011ObstacleSymbols::getArcRight);
9   engine.registerDecimalInputSymbol("obstacle.arc_left", this, &
       BH2011ObstacleSymbols::getArcLeft);
10  engine.registerBooleanInputSymbol("obstacle.exist", &
       obstacleExist);
11
12 if(obstacle.type == ObstacleModel::Obstacle::US)
13         {
14                 float sqr = obstacle.center.abs();
15                 float angle = toDegrees(obstacle.center.angle());
16                 if( abs(angle) < 5 && sqr < distanceClosestCenter)
                       // Obstacle is in front of robot
17                 {
18                         centerArcLength = sqr*40*pi/180;
19                         distanceClosestCenter = sqr;
20                         angleClosestCenter = 0;
21                 }
22                 if(angle < -5 && sqr < distanceClosestRight) //
                      Obstacle is in the right side
23                 {
24                         rightArcLength = sqr*40*pi/180;
25                         distanceClosestRight = sqr;
26                         angleClosestRight = -20;
27                 }
28                 else if(angle > 5 && sqr < distanceClosestLeft) //
                       Obstacle is in the left side
29                 {
30                         leftArcLength = sqr*40*pi/180;
31                         distanceClosestLeft = sqr;
32                         angleClosestLeft = 20;
33                 }
34         }
```

Apéndice B

Algoritmos para el comportamiento Striker

Striker

```
option striker
{
  initial state striker
  {
    decision
    {
      if(ball.time_since_last_seen > 4000)
        goto search_ball;
      else if(ball.was_seen && (game.kickoff_team == game.
         team_color
                              || game.time_since_playing_started >
                                 10000
                              || abs(value = ball.estimate.x) > 500
                              || abs(value = ball.estimate.y) > 500
                                 )
)
        goto go_to_ball;
      else
        stay;
    }
    action
    {
      motion.type = stand;
      head.control_mode = look_at_ball;
    }
  }
```

```
24
25
26    state search_ball
27    {
28      decision
29      {
30        if(ball.was_seen)
31          goto go_to_ball;
32        else
33          stay;
34      }
35      action
36      {
37        search_ball();
38      }
39    }
40
41    state go_to_ball
42    {
43      decision
44      {
45        if(ball.time_since_last_seen > 4000)
46          goto search_ball;
47        else if(abs( value = opp_goal.local_projected.angle ) < 30
48                && ball.global.x < 1000
49                && abs(value = ball.estimate.angle) < 5
50                && between(value = ball.estimate.x, min = 0, max =
                     500)
51                && between(value = ball.estimate.y, min = -200, max
                     = 200)
52                && ball.estimate.distance < obstacle.
                     distance_to_closest)
53          goto bring_ball;
54        else if(ball.estimate.distance < 500)
55          goto align;
56        else
57          stay;
58      }
59      action
60      {
61        go_to_ball();
62      }
63    }
64
```

```
65  state align
66  {
67    decision
68    {
69      if(ball.time_since_last_seen > 3000)
70        goto search_ball;
71      else if(ball.estimate.distance > 550)
72        goto go_to_ball;
73      else
74        stay;
75    }
76    action
77    {
78      align();
79    }
80  }
81
82  state bring_ball
83  {
84    decision
85    {
86      if(ball.time_since_last_seen > 1000)
87        goto search_ball;
88      else if(ball.estimate.distance > 600
89              || ball.global.x > 1100
90              || abs(value = ball.global.y) > 1000
91              || obstacle.distance_to_closest < 400)
92        goto go_to_ball;
93      else
94        stay;
95
96    }
97    action
98    {
99      motion.walk_pedantic = true;
100     motion.type = walk;
101     motion.dribbling = true;
102     motion.walk_target.x = ball.seen.x;
103     motion.walk_target.y = ball.seen.y;
104     motion.walk_target.rot = ball.seen.angle;
105     motion.walk_speed = 80;
106     head.control_mode = look_at_ball;
107   }
108 }
```

```
}
```

Search Ball

```
option search_ball
{
  initial state static_search
  {
    decision
    {
      if(ball.cooperative.time_since_disappeared < 1000 && locator
         .pose.validity > 0.7)
        goto look_cooperative_ball;
      else if(state_time > 3000)
        goto turn;
      else
        stay;
    }
    action
    {
      motion.type = walk;
      motion.walk_target.x = 0;
      motion.walk_target.y = 0;
      motion.walk_target.rot = 0;
      motion.walk_speed = 30;
      head.control_mode = look_ellipse;
    }
  }

  state turn
  {
    decision
    {
      if(ball.cooperative.time_since_disappeared < 1000 && locator
         .pose.validity > 0.7)
        goto look_cooperative_ball;
      else if(state_time > 9000)
        goto active_search;
      else
        stay;
    }
    action
    {
      motion.type = walk;
      motion.walk_target.x = 0;
```

```
40        motion.walk_target.y = 0;
41        motion.walk_target.rot = ball.estimate.y < 0 ? -180: 180;
42        motion.walk_speed = 50;
43        head.control_mode = state_time % 2500 < 1000 ? look_down :
              look_up;
44      }
45    }
46    state look_cooperative_ball
47    {
48      decision
49      {
50        if( state_time > 5000 )
51          goto static_search;
52        else
53          stay;
54      }
55      action
56      {
57        move(x = ball.cooperative.x, y = ball.cooperative.y, angle =
              0, dist = 0 );
58        head.control_mode = look_at_cooperative_ball;
59      }
60    }
61    state active_search
62    {
63      decision
64      {
65        if(ball.cooperative.time_since_disappeared < 1000 && locator
              .pose.validity > 0.7)
66          goto look_cooperative_ball;
67        else if( distance_to_target_pose < 40 )
68          goto static_search;
69        else
70          stay;
71      }
72      action
73      {
74        move( x = locator.ready_positionning.x,
75              y = locator.ready_positionning.y,
76              angle = locator.ready_positionning.angle,
77              dist = 500);
78        head.control_mode = distance_to_target_pose < 500?
              look_at_goal_opp : look_at_goal_own;
79      }
```

```
  }
}
```

Go To Ball

```
option go_to_ball
{
  state turn
  {
    decision
    {
      if( abs( value = ball.estimate.angle ) < 5 )
        goto walk;
      else
        stay;
    }
    action
    {
      motion.type = walk;
      motion.walk_target.x = 0;
      motion.walk_target.y = 0;
      motion.walk_target.rot = ball.estimate.angle;
      motion.walk_speed = 45;
      obstacle_avoidance(angle = ball.estimate.angle);
      head.control_mode = look_at_ball;
    }
  }

  initial state walk
  {
    decision
    {
      if(abs(value = ball.seen.angle) > 10)
        goto turn;
      else
        stay;
    }
    action
    {
      motion.type = walk;
      motion.walk_target.x = ball.estimate.x;
      motion.walk_target.y = 0.5 * ball.estimate.y;
      motion.walk_target.rot = 3.9 * ball.estimate.angle;
      motion.walk_speed = 100;
      obstacle_avoidance(angle = ball.estimate.angle);
```

```
41        head.control_mode = state_time % 4000 < 2000 ?
              look_up_and_down : look_at_ball;
42      }
43    }
44  }
```

Align

```
1  option align
2  {
3    initial state decision_align
4    {
5      decision
6      {
7        if(abs( value = opp_goal.local_projected.angle ) > 135)
8          goto align_around_ball;
9        else if(abs(value = opp_goal.local_projected.angle) < 135 &&
               abs(value = opp_goal.local_projected.angle) > 50)
10         goto align_next_goal;
11       else if(abs( value = opp_goal.local_projected.angle ) < 50)
12         goto align_to_goal;
13       else if(ball.estimate.distance < obstacle.left.distance ||
              ball.estimate.distance < obstacle.right.distance)
14         goto align_to_ball;
15     }
16     action
17     {
18     }
19   }
20
21   state align_next_goal
22   {
23     decision
24     {
25       if(abs(value = opp_goal.local_projected.angle) < 95 && abs(
              value = opp_goal.local_projected.angle) > 85)
26         goto prepare_kick;
27       else
28         stay;
29     }
30     action
31     {
32       motion.type = walk;
33       motion.walk_target.x = ball.estimate.x - 300;
34       motion.walk_target.y = ball.estimate.y;
```

```
35        motion.walk_target.rot = opp_goal.local_projected.angle;
36        motion.walk_speed = 80;
37        head.control_mode = state_time % 3000 < 1000 ?
             look_at_goal_opp : look_at_ball;
38        walkKick.type = none;
39        kick.type = inwalk_sidewards;
40        kick.side = opp_goal.local_projected.angle > 0 ? right :
             left;
41      }
42    }
43
44    state align_to_goal
45    {
46      decision
47      {
48        if(abs( value = opp_goal.local_projected.angle ) < 10 && abs
             ( value = ball.estimate.y ) < 100)
49          goto prepare_kick;
50        else if(abs( value = opp_goal.local_projected.angle ) > 90)
51          goto align_around_ball;
52        else
53          stay;
54      }
55      action
56      {
57        motion.type = walk;
58        motion.walk_target.x = ball.estimate.x - 300;
59        motion.walk_target.y = ball.estimate.y;
60        motion.walk_target.rot = opp_goal.local_projected.angle;
61        motion.walk_speed = 80;
62        head.control_mode = state_time % 3000 < 1000 ?
             look_at_goal_opp : look_at_ball;
63        walkKick.type = none;
64        kick.type = opp_goal.local_projected.distance > 3000 ?
             forward : inwalk_forward;
65        kick.side = ball.estimate.y > 0 ? left : right;
66      }
67    }
68
69    state align_around_ball
70    {
71      decision
72      {
73        if(abs( value = opp_goal.local_projected.angle) < 50)
```

```
74          goto align_to_goal;
75        else if(abs( value = opp_goal.local_projected.angle) < 135
             && ball.global.x > 1500 && abs(value = ball.global.y) <
             1500)
76          goto align_next_goal;
77        else
78          stay;
79      }
80      action
81      {
82        motion.type = walk;
83        motion.walk_target.x = ball.estimate.x - 300;
84        motion.walk_target.y = ball.estimate.y;
85        motion.walk_target.rot = abs( value = opp_goal.
             local_projected.angle ) > 90  ? opp_goal.local_projected.
             angle / 3 : 0;
86        motion.walk_speed = 80;
87        head.control_mode = look_at_ball;
88        kick.type = inwalk_sidewards;
89        kick.side =  opp_goal.local_projected.angle > 0 ? right :
             left;
90      }
91    }
92
93    state prepare_kick
94    {
95      decision
96      {
97        if(!ball.was_seen)
98          goto search_ball;
99        else if(state_time > 4000)
100         goto decision_align;
101       else
102         stay;
103     }
104     action
105     {
106       prepare_to_kick();
107     }
108   }
109
110   state search_ball
111   {
112     decision
```

```
113       {
114         if(ball.was_seen)
115           goto decision_align;
116         else
117           stay;
118       }
119       action
120       {
121         motion.type = walk;
122         motion.walk_target.x = 0;
123         motion.walk_target.y = 0;
124         motion.walk_target.rot = 0;
125         motion.walk_speed = 30;
126         head.control_mode = look_ellipse;
127       }
128     }
129
130     state align_to_ball
131     {
132       decision
133       {
134         if(ball.estimate.distance < 250 && ball.was_seen)
135           goto prepare_kick;
136         else if(!ball.was_seen)
137           goto search_ball;
138         else
139           stay;
140       }
141       action
142       {
143         motion.type = walk;
144         motion.walk_target.x = ball.seen.x;
145         motion.walk_target.y = ball.seen.y;
146         motion.walk_target.rot = ball.seen.angle*3.9;
147         motion.walk_speed = 40;
148         head.control_mode = look_at_ball;
149         walkKick.type = none;
150         kick.type = abs(value = opp_goal.local_projected.angle) > 60
                ? inwalk_sidewards:inwalk_forward;
151         kick.side = opp_goal.local_projected.angle < 0 ? right:left;
152       }
153     }
154   }
```

Prepare Kick

```
option prepare_to_kick
{
  initial state decision_kick
  {
    decision
    {
            if(kick.type == sidewards && kick.side == left)
              goto align_sidewards_left;
      else if(kick.type == sidewards && kick.side == right)
        goto align_sidewards_right;
            else
 if(kick.type == forward)
              goto align_forward;
      else if(kick.type == inwalk_forward && kick.side == right)
        goto align_forward_inwalk_right;
      else if(kick.type == inwalk_forward && kick.side == left)
        goto align_forward_inwalk_left;
      else if(kick.type == inwalk_sidewards && kick.side == left)
        goto align_sidewards_inwalk_left;
      else if(kick.type == inwalk_sidewards && kick.side == right)
        goto align_sidewards_inwalk_right;
      else
        stay;
    }
    action
    {
      head.control_mode = look_at_ball;
    }
  }

  state align_forward
  {
    decision
    {
            //if(between(value = obstacle.distance_to_closest, min
                = 25, max = 350))
        //goto align_sidewards_left;
      //if(between(value = obstacle.distance_to_closest, min = 25,
          max = 350))
        //goto align_sidewards_right;
      if(between( value = ball.seen.y, min = -60, max = -40)&&
         between( value = ball.seen.x, min = 150, max = 170))
        goto kick_forward;
```

```
41        else
42          stay;
43      }
44      action
45      {
46        walkKick.type = none;
47        motion.type = walk;
48        motion.walk_target.x = ball.seen.x - 160;
49        motion.walk_target.y = ball.seen.y + 50;
50        motion.walk_target.rot = 0;
51        motion.walk_speed = 50;
52        head.control_mode = look_at_ball;
53      }
54    }
55
56    state align_sidewards_left
57    {
58      decision
59      {
60        if(between( value = ball.seen.y, min = -10, max = 10 )&&
             between( value = ball.seen.x, min = 130, max = 150 ))
61          goto kick;
62        else
63          stay;
64      }
65      action
66      {
67        walkKick.type = none;
68        motion.type = walk;
69        motion.walk_target.x = ball.seen.x - 130;
70        motion.walk_target.y = ball.seen.y - 10;
71        motion.walk_target.rot = 0;
72        motion.walk_speed = 50;
73        head.control_mode = look_at_ball;
74        kick.type = sidewards;
75        kick.side =left;
76      }
77    }
78    state align_sidewards_right
79    {
80      decision
81      {
82        if(between( value = ball.seen.y, min = -10, max = 10 )&&
             between( value = ball.seen.x, min = 150, max = 170 ))
```

```
83          goto kick;
84        else
85          stay;
86      }
87      action
88      {
89        walkKick.type = none;
90        motion.type = walk;
91        motion.walk_target.x = ball.seen.x - 130;
92        motion.walk_target.y = ball.seen.y + 10;
93        motion.walk_target.rot = 0;
94        motion.walk_speed = 50;
95        head.control_mode = look_at_ball;
96        kick.type = sidewards;
97        kick.side =right;
98      }
99    }
100   state align_forward_inwalk_right
101   {
102     decision
103     {
104       //if(between(value = obstacle.distance_to_closest, min = 25,
              max = 350))
105         //goto align_sidewards_left;
106       //if(between(value = obstacle.distance_to_closest, min = 25,
              max = 350))
107         //goto align_sidewards_right;
108       if(between( value = ball.seen.y, min = -50, max = -30)&&
             between( value = ball.seen.x, min = 150, max = 170))
109         goto kick;
110       else
111         stay;
112     }
113     action
114     {
115       walkKick.type = none;
116       motion.type = walk;
117       motion.walk_target.x = ball.seen.x - 160;
118       motion.walk_target.y = ball.seen.y + 40;
119       motion.walk_target.rot = 0;
120       motion.walk_speed = 50;
121       head.control_mode = look_at_ball;
122     }
123   }
```

```
124
125   state align_forward_inwalk_left
126   {
127     decision
128     {
129       //if(between(value = obstacle.distance_to_closest, min = 25,
              max = 350))
130         //goto align_sidewards_left;
131       //if(between(value = obstacle.distance_to_closest, min = 25,
              max = 350))
132         //goto align_sidewards_right;
133       if(between( value = ball.seen.y, min = 30, max = 50)&&
             between( value = ball.seen.x, min = 150, max = 170))
134         goto kick;
135       else
136         stay;
137     }
138     action
139     {
140       walkKick.type = none;
141       motion.type = walk;
142       motion.walk_target.x = ball.seen.x - 160;
143       motion.walk_target.y = ball.seen.y - 40;
144       motion.walk_target.rot = 0;
145       motion.walk_speed = 50;
146       head.control_mode = look_at_ball;
147     }
148   }
149
150   state align_sidewards_inwalk_left
151   {
152     decision
153     {
154       if(between( value = ball.seen.y, min = 5, max = 15 )&&
             between( value = ball.seen.x, min = 130, max = 160 ))
155         goto kick;
156       else
157         stay;
158     }
159     action
160     {
161       walkKick.type = none;
162       motion.type = walk;
163       motion.walk_target.x = ball.seen.x - 155;
```

```
164       motion.walk_target.y = ball.seen.y - 10;
165       motion.walk_target.rot = 0;
166       motion.walk_speed = 50;
167       head.control_mode = look_at_ball;
168       kick.type = inwalk_sidewards;
169       kick.side =left;
170     }
171
172   }
173
174   state align_sidewards_inwalk_right
175   {
176     decision
177     {
178       if( between( value = ball.seen.y, min = -15, max = 5 )&&
             between( value = ball.seen.x, min = 130, max = 160 ) )
179         goto kick;
180       else
181         stay;
182     }
183     action
184     {
185       walkKick.type = none;
186       motion.type = walk;
187       motion.walk_target.x = ball.seen.x - 150;
188       motion.walk_target.y = ball.seen.y + 10;
189       motion.walk_target.rot = 0;
190       motion.walk_speed = 50;
191       head.control_mode = look_at_ball;
192       kick.type = inwalk_sidewards;
193       kick.side = right;
194     }
195
196   }
197
198   state kick_forward
199   {
200     decision
201     {
202       if(motion.kick_forward( x = ball.seen.x, y = ball.seen.y,
             mirror = false, updates = false))
203         goto kick;
204       else
205         stay;
```

```
206       }
207       action
208       {
209         motion.type = stand;
210         head.control_mode = look_at_ball;
211       }
212     }
213
214     state kick
215     {
216       decision
217       {
218         if(state_time > 2500)
219           goto decision_kick;
220         else
221           stay;
222       }
223       action
224       {
225          kick();
226       }
227     }
228   }
```

Kick

```
1   option kick
2   {
3     common decision
4     {
5       if(kick.type == forward)
6         goto forward;
7       else if(kick.type == sidewards)
8         goto sidewards;
9       else if(kick.type == diagonal)
10        goto diagonal;
11      else if(kick.type == inwalk_forward)
12        goto inwalk_forward;
13      else if(kick.type == inwalk_sidewards)
14        goto inwalk_sidewards;
15    }
16
17    initial state inwalk_forward
18    {
19      action
```

```
20      {
21        motion.type = walk;
22        motion.walk_target.x = 0;
23        motion.walk_target.y = 0;
24        motion.walk_target.rot = 0;
25        motion.walk_speed = 50;
26        head.control_mode = look_at_ball;
27        walkKick.type = kick.side == right ? right : left;
28        kickBallPosition.x = ball.seen.x;
29        kickBallPosition.y = ball.seen.y;
30        kickTarget.x = 100;
31        kickTarget.y = 0;
32      }
33    }
34
35    state inwalk_sidewards
36    {
37      action
38      {
39        motion.type = walk;
40        motion.walk_target.x = 0;
41        motion.walk_target.y = 0;
42        motion.walk_target.rot = 0;
43        motion.walk_speed = 50;
44        head.control_mode = look_at_ball;
45        walkKick.type = kick.side == right ? sidewards_right:
             sidewards_left;
46        kickBallPosition.x = ball.seen.x;
47        kickBallPosition.y = ball.seen.y;
48        kickTarget.x = 0;
49        kickTarget.y = kick.side == right ? 100 : -100;
50      }
51    }
52
53    state forward
54    {
55      action
56      {
57        motion.type = bike;
58        head.control_mode = look_at_ball;
59      }
60    }
61
62    state sidewards
```

```
63    {
64      action
65      {
66        motion.type = special_action;
67        motion.special_action = kick.side == left ?
              kick_sidewards_left_nao : kick_sidewards_nao;
68        head.control_mode = look_at_ball;
69      }
70    }
71
72    state diagonal
73    {
74      action
75      {
76        motion.type = special_action;
77        motion.special_action = kick_diagonal_nao;
78        head.control_mode = look_at_ball;
79      }
80    }
81  }
```

Avoid Obstacle

```
1  option obstacle_avoidance
2  {
3    float @angle; //en el sistema de referencia local del robot.
4
5    initial state sonar
6    {
7      decision
8      {
9        if (obstacle.center.distance < 600)
10         if (@angle < 0)
11           goto walk_right;
12         else
13           goto walk_left;
14       else if (obstacle.left.distance < 600)
15         goto walk_right;
16       else if (obstacle.right.distance < 600)
17         goto walk_left;
18       else if (obstacle.arm_left)
19         goto walk_right_arm;
20       else if (obstacle.arm_right)
21         goto walk_left_arm;
22       else if (obstacle.footbumper_left)
```

```
23          goto walk_right_bumper;
24        else if (obstacle.footbumper_right)
25          goto walk_left_bumper;
26        else
27          stay;
28      }
29      action
30      {
31      }
32    }
33
34    state walk_left
35    {
36      decision
37      {
38        if (obstacle.right.distance > 600)
39          goto walk_left_final;
40        else
41          stay;
42      }
43      action
44      {
45        motion.type = walk;
46        motion.walk_speed.x = ((obstacle.right.distance > 600 ? 600
             : obstacle.right.distance) / 600) * 70;
47        motion.walk_speed.y = (1 - (obstacle.right.distance > 600 ?
             600 : obstacle.right.distance) / 600) * 150;
48        motion.walk_speed.rot = @angle * 0.4;
49      }
50    }
51
52    state walk_left_final
53    {
54      decision
55      {
56        if (state_time > 2000)
57          goto sonar;
58        else
59          stay;
60      }
61      action
62      {
63        motion.type = walk;
64        motion.walk_speed.x = 100;
```

```
65        motion.walk_speed.y = abs(value = @angle) * 2.0;
66        motion.walk_speed.rot = @angle * 0.3;
67      }
68    }
69
70    state walk_right
71    {
72      decision
73      {
74        if (obstacle.left.distance > 600)
75          goto walk_right_final;
76        else
77          stay;
78      }
79      action
80      {
81        motion.type = walk;
82        motion.walk_speed.x = ((obstacle.left.distance > 600 ? 600 :
              obstacle.left.distance ) / 600) * 70;
83        motion.walk_speed.y = (1 - (obstacle.left.distance > 600 ?
             600 : obstacle.left.distance ) / 600) * -150;
84        motion.walk_speed.rot = @angle * 0.4;
85      }
86    }
87
88    state walk_right_final
89    {
90      decision
91      {
92        if (state_time > 2000)
93          goto sonar;
94        else
95          stay;
96      }
97      action
98      {
99        motion.type = walk;
100       motion.walk_speed.x = 100;
101       motion.walk_speed.y = abs(value = @angle) * -2.0;
102       motion.walk_speed.rot = @angle * 0.3;
103     }
104   }
105
106   state walk_left_arm
```

```
107   {
108     decision
109     {
110       if (state_time > 2000)
111         goto sonar;
112       else
113         stay;
114     }
115     action
116     {
117       motion.type = walk;
118       motion.walk_speed.x = -10;
119       motion.walk_speed.y = 50;
120       motion.walk_speed.rot = @angle * 0.1;
121     }
122   }
123
124   state walk_right_arm
125   {
126     decision
127     {
128       if (state_time > 2000)
129         goto sonar;
130       else
131         stay;
132     }
133     action
134     {
135       motion.type = walk;
136       motion.walk_speed.x = -50;
137       motion.walk_speed.y = -50;
138       motion.walk_speed.rot = @angle * 0.1;
139     }
140   }
141
142   state walk_left_bumper
143   {
144     decision
145     {
146       if (state_time > 2000)
147         goto sonar;
148       else
149         stay;
150     }
```

```
151       action
152       {
153         motion.type = walk;
154         motion.walk_speed.x = -100;
155         motion.walk_speed.y = 50;
156         motion.walk_speed.rot = @angle * 0.1;
157       }
158     }
159
160     state walk_right_bumper
161     {
162       decision
163       {
164         if (state_time > 2000)
165           goto sonar;
166         else
167           stay;
168       }
169       action
170       {
171         motion.type = walk;
172         motion.walk_speed.x = -100;
173         motion.walk_speed.y = -50;
174         motion.walk_speed.rot = @angle * 0.1;
175       }
176     }
177   }
```