UNIVERSIDAD REY JUAN CARLOS

Ingeniería Técnica en Informática de Sistemas

Escuela Superior de Ciencias Experimentales y Tecnología

Curso académico 2002-2003

Proyecto Fin de Carrera

Equipo simulado para la RoboCup basado en Jerarquía Dinámica de Esquemas

Autor: JuanJosé Martínez Gil

Tutor 1: Vicente Matellán Olivera

Tutor 2: JoseMaría Cañas Plaza

A mi familia y amigos

Quiero dar las gracias a Vicente Matellán y Jose Mª Cañas por haberme proporcionado sus conocimientos y ayuda para la realización de este proyecto.

Resumen

El presente proyecto se ha desarrollado en el contexto de la RoboCup, una organización que pone en marcha una serie de competiciones a nivel mundial, con el fin de fomentar la investigación en el campo de la robótica móvil. Entre las múltiples categorías de participación que propone, nos centraremos en la liga simulada.

La liga simulada consiste en un entorno virtual que simula un partido de fútbol entre robots. Cada robot jugador es un programa independiente, que se comunica con el servidor del juego para que éste le informe de lo que ve, oye y siente ése jugador en particular. Con esta información, el programa puede decidir qué acciones realizar en el terreno de juego, que luego comunicará al servidor para que éste las ejecute.

Lo que se ha desarrollado son precisamente estos agentes que, en conjunto, son capaces de comportarse como jugadores de fútbol virtuales, en un entorno dinámico. Para su implementación se ha utilizado la arquitectura JDE: Jerarquía Dinámica de Esquemas, mediante la cual se construye un árbol de comportamientos que reaccionan ante determinados estímulos sensoriales, produciendo una salida que se traduce finalmente en las acciones que realiza el agente. Como objetivo principal del proyecto, nos proponemos probar la validez de la arquitectura JDE dentro de este entorno.

El equipo que hemos obtenido es capaz de jugar en la liga simulada, comportándose en general satisfactoriamente frente a la distintas situaciones a las que se le ha enfrentado.

Índice general

Índice de figuras

Índice de cuadros

Capítulo 1

Introducción

Introduciremos en este capítulo la robótica, marco en el que se inscribe este proyecto. Además se explicará qué es la *RoboCup*, describiremos las distintas competiciones en las que se divide e indicaremos en cuál de ellas entraría el presente proyecto.

Finalmente, haremos una primera introducción al entorno de desarrollo y a las características de la implementación que se ha realizado.

1.1. Robots y Robótica

El término *robot* fue inventado por el escritor Karel Capek [1] en su obra *Rossum's Universal Robot*, donde describe unas creaciones humanoides capaces de trabajar incansablemente, diseñados para sustituir a los trabajadores humanos.

El diccionario de la Real Academia define la robótica de esta forma:

> *Técnica que aplica la informática al diseño y empleo de aparatos que, en sustitución de personas, realizan operaciones o trabajos, por lo general en instalaciones industriales.*

Desde que el escritor y científico Isaac Asimov inventara la palabra robótica (*robotics*, en inglés), ésta ha llegado a multitud de entornos, saltando desde la ciencia ficción a las aplicaciones puramente industriales, hasta llegar hoy en día al ámbito doméstico y de ocio. Hoy se pueden encontrar robots que hacen de guías en un museo[1], que representan el papel de una mascota[2], o que incluso juegan al fútbol unos con otros[3].

[1] http://www-2.cs.cmu.edu/ minerva/
[2] http://www.sony.net/Products/aibo/
[3] http://www.robocup.org

La gran diversidad de aplicaciones que tiene la robótica hoy en día hace que su investigación se potencie deste mucho entornos. En los últimos años, la róbotica móvil está teniendo cada vez más aplicaciones prácticas y también de ocio, a la vez que se mejoran las tecnologías implicadas en este campo.

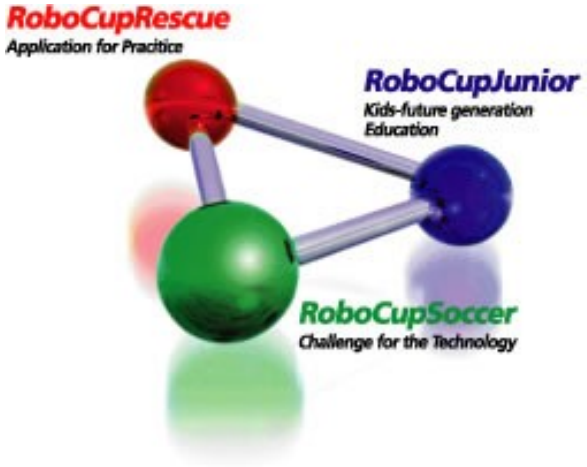
1.2. La RoboCup

La RoboCup es una iniciativa que propone utilizar el fútbol como entorno para la investigación y el desarrollo de la robótica móvil. El sueño y objetivo de la RoboCup es, para mediados del presente siglo, desarrollar un equipo de robots humanoides, capaz de ganar al equipo humano campeón mundial de fútbol, según las normas de la FIFA. Es un objetivo ambicioso cuya consecución habrá impulsado el desarrollo de tecnologías, tanto en el ámbito del hardware, como de las técnicas de inteligencia artificial.

1.2.1. Competiciones de la RoboCup

La RoboCup engloba tres grandes escenarios de participación:

- *RoboCupSoccer*: El principal, y único que había en un principio. El objetivo es desarrollar un equipo de fútbol robótico (ya sea real o simulado) con el que participar en un campeonato.
- *RoboCupRescue*: Añadido posteriormente como un nuevo dominio de investigación, este escenario propone simular situaciones de grandes desastres, como terremotos, en los que coordinar distintos agentes heterogéneos para buscar y rescatar a las víctimas.
- *RoboCupJunior*: Iniciativa orientada a estudiantes de primaria y secundaria, que engloba varias competiciones y eventos robóticos, con fines educativos.



Dentro del escenario que nos compete, la *RoboCupSoccer*, hay varias categorías:

- *Humanoid League*: Introducida recientemente (año 2002), será la que lleve al objetivo último de la RoboCup. Robots bípedos humanoides compiten en pruebas de andar y chutar, tiros de penaltis y "partidos" de uno contra uno.
- *Four-legged robot league*: Patrocinado por Sony, equipos de cuatro robots perritos Aibo, compiten en partidos de 20 minutos en un campo de 3 x 5 metros.
- *Middle-size robot league*: Equipos de cuatro robots de hasta 50 cm. de diámetro compiten en un campo del tamaño de nueve mesas de ping pong en partidos de 20 minutos.
- *Small-size robot league*: Robots de no más de 18 cm. de diámetro compiten en equipos de cinco, durante 20 minutos, en un campo del tamaño de dos mesas de ping pong.
- *Simulation league*: La primera de las ligas que se pusieron en marcha. Aquí, equipos de once agentes software compiten en un entorno virtual simulado, en partidos de 10 minutos de duración.

1.3. Objetivo: Un equipo en la liga simulada

En este proyecto nos proponemos programar un equipo de jugadores software capaz de jugar en la *Simulation league* de la RoboCup. Este equipo se programará usando una arquitectura de control desarrollada en el grupo de robótica de la Universidad Rey Juan Carlos, llamada Jerarquía Dinámica de Esquemas (JDE [3]), siendo la prueba de la validez en este entorno de dicha arquitectura el objetivo fundamental del proyecto.

Participar en la liga simulada supone programar una serie de agentes software que harán las veces de jugadores. Cada uno de estos jugadores será un proceso que se ejecute de forma totalmente independiente a los demás, sin ninguna clase de comunicación entre ellos más allá de la que se permite en el entorno virtual donde actuarán. Esto supone que en ningún momento se tendrá una visión global del escenario. Cada jugador poseerá únicamente la información que recibe de sus sensores virtuales a la cual, además, se le añade un elemento de ruido que simula los problemas sensoriales que tienen los robots reales.

Los agentes software que programan los participantes deciden las acciones que quieren realizar como jugadores, pero el control del juego, la ejecución de estas acciones y la generación del resultado de las mismas corren por cuenta de un programa

proporcionado por la organización de la RoboCup, el *Soccerserver*, que es el que hace las funciones de servidor, simulador y árbitro del partido, como explicaremos en el capítulo 2.

Para el desarrollo de los equipos que participan en esta categoría, es necesario recurrir a técnicas de inteligencia artificial que permitan crear comportamientos adecuados en los agentes. Dentro de estas técnicas nosotros, como ya se ha comentado, utilizaremos JDE. Esta arquitectura basada en comportamientos, tal y como se explica en el capítulo 3, propone la creación de esquemas de comportamientos básicos, cada uno de ellos con un objetivo concreto. Estos esquemas, reunidos incrementalmente en un árbol jerárquico, generan finalmente comportamientos más complejos.

Como se detalla en el capítulo 4, se ha utilizado una implementación en lenguaje C de la arquitectura JDE para desarrollar los jugadores del equipo. Esta implementación define, en ficheros separados, los distintos esquemas de comportamiento que componen el comportamiento complejo que queremos conseguir. Estos esquemas producen una salida de actuación que, con la ayuda de una biblioteca de comunicaciones, se traducen en órdenes que se envían al servidor *Soccerserver* donde éste las ejecuta.

Se han realizado numerosas pruebas del comportamiento conseguido, tanto a nivel de un sólo proceso jugador como de equipos formados por once de estos procesos. Estas pruebas se detallan en el capítulo 5.

Por último, concluímos en el capítulo 6 con un repaso a los objetivos que se han cumplido con la realización del presente proyecto, así como unas indicaciones sobre las posibles futuras líneas de trabajo.

Capítulo 2

El simulador

En este capítulo se explicarán las características del simulador sobre el que actuarán los procesos del equipo que hemos desarrollado. Se detallará la información sensorial que poseerán los jugadores, así como el formato en que ésta se les presenta. De igual forma, indicaremos las distintas acciones que puede un jugador realizar durante el partido.

2.1. *Soccerserver*

El *Soccerserver* es el programa que, en la liga simulada de la RoboCup, proporciona el entorno donde actuarán los jugadores de los equipos virtuales de los participantes. Actúa como un servidor que habilita un puerto de comunicaciones al que se conecta cada uno de los procesos jugadores, a través del cual envía a éstos periódicamente la información sensorial visual y auditiva (descrita en la sección 2.3.1) que ha de recibir cada uno, de acuerdo con su situación en el campo. Es también a través de este canal como cada jugador comunica las acciones que quiere realizar, de acuerdo con sus intenciones de juego (fig. 2.1).

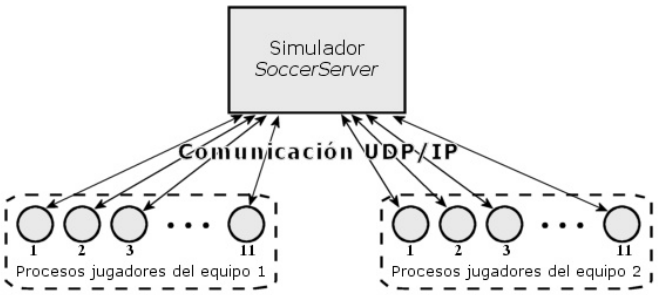


Figura 2.1: Representación de los procesos de una simulación

Para la visualización del partido se utiliza un cliente especial, llamado *monitor*, que

se conecta al *Soccerserver* y presenta de forma visual el terreno de juego. La pelota y cada uno de los jugadores es representado en él de forma que podemos ver la evolución del juego de igual forma a como veríamos un partido real en la televisión, con una vista aérea.

Existen versiones para GNU/Linux y para Windows, tanto del servidor *Soccerserver* como de los programas monitores, y en las competiciones oficiales compiten conjuntamente equipos programados para ambos sistemas operativos. Esto es posible ya que la comunicación entre los programas clientes y el servidor se produce mediante *sockets* UDP/IP, lo que hace irrelevante qué herramientas se hayan utilizado para desarrollar los programas, así como el sistema en el que se estén ejecutando.

2.2. Control del juego

El control del juego lo lleva a cabo el módulo *referee*, o árbitro, del *Soccerserver*. Éste se encarga de que se cumplan las reglas del fútbol dentro del partido virtual.

Además del control automatizado del servidor, se contempla la posibilidad de que un árbitro humano pueda, por ejemplo, parar el partido en cualquier momento y conceder tiros libres a uno de los equipos, si juzga que se ha producido un comportamiento inadecuado que no sea capaz de detectar el simulador.

Cuando se da una situación que produce un cambio en el modo de juego, como por ejemplo que la pelota rebase las líneas del campo, se comunica esta situación a los jugadores mediante un mensaje de tipo auditivo enviado por el árbitro. El simulador impide también que los jugadores hagan algo que se supone que no deben hacer, como por ejemplo acercarse demasiado a la pelota si le toca sacar al equipo contrario. Esto permite que un comportamiento inadecuado ante estos casos (debido a una programación incorrecta o incompleta de los agentes) no suponga una molestia al equipo contrario, pudiéndose seguir jugando el partido con normalidad.

El cuadro 2.1 muestra los posibles modos de juego que pueden darse durante el partido

2.3. Simulación

La simulación se ejecuta en el *Soccerserver* de forma discreta, en ciclos de 100 milisegundos, en los cuales el simulador aplica los cambios de aceleración, giros, etc que cada jugador desee realizar y calcula el movimiento de cada jugador y la pelota de acuerdo a estas acciones, produciendo al final de cada ciclo una nueva posición y velocidad para cada objeto en el campo. Independientemente a los ciclos de la

before_kick_off	antes de comenzar el partido
play_on	modo de juego normal
time_over	se acabó el tiempo
kick_off_*Side*	fuera de banda del equipo *Side*
kick_in_*Side*	saque de banda para el equipo *Side*
free_kick_*Side*	tiro libre para el equipo *Side*
corner_kick_*Side*	tiro de corner para el equipo *Side*
goal_kick_*Side*	tiro de penalti para el equipo *Side*
drop_ball	modo especial usado por el árbitro humano
offside_*Side*	fuera de juego del equipo *Side*

Cuadro 2.1: Modos de juego durante el partido

simulación, cada 150 milisegundos se envía a cada jugador la información visual que deba recibir.

El ciclo de actuación de un programa cliente es, generalmente, recibir la información sensorial del servidor, decidir qué acción realizar, y enviar las órdenes pertinentes.

2.3.1. Información sensorial

La información sensorial se envía a los jugadores mediante paquetes UDP que luego estos han de traducir y procesar. Hay tres tipos de información sensorial: la auditiva, la visual y la introspectiva, es decir, la relativa al estado interno del jugador.

Información auditiva

Se envía a los jugadores en cuanto se produce, sin esperar al final del ciclo de simulación. Hay una limitación en cuanto a la cantidad de información auditiva que un jugador puede recibir en una determinada cantidad de tiempo, pudiendo no llegar a enviársele un mensaje determinado si se produce a la vez que otros.

Los mensajes tienen el siguiente formato:

`(hear` *`Time Sender "Mensaje"`*`)`

Time indica el ciclo de simulación en el que se ha producido el mensaje.

Sender hace referencia al emisor del mensaje. Puede valer *referee* si el emisor es el árbitro, *self* si el emisor es el propio jugador o la dirección relativa si el emisor es otro jugador.

Mensaje es el mensaje que se envía.

Un ejemplo de mensaje con información auditiva sería:

`(hear 1230 referee "goal_l")`

Lo que indicaría que en el ciclo 1.230 el jugador ha oído al árbitro anunciar un gol para el equipo que está jugando en el lado izquierdo del campo.

Información visual

El jugador verá aquellos objetos que se encuentren dentro de su cono de visión, incluyendo a la pelota, los jugadores, y los objetos del terreno de juego representados en la figura 2.2 como son las porterías, las líneas del campo y las balizas o "banderas" que pueden usarse para averiguar la posición absoluta del jugador.

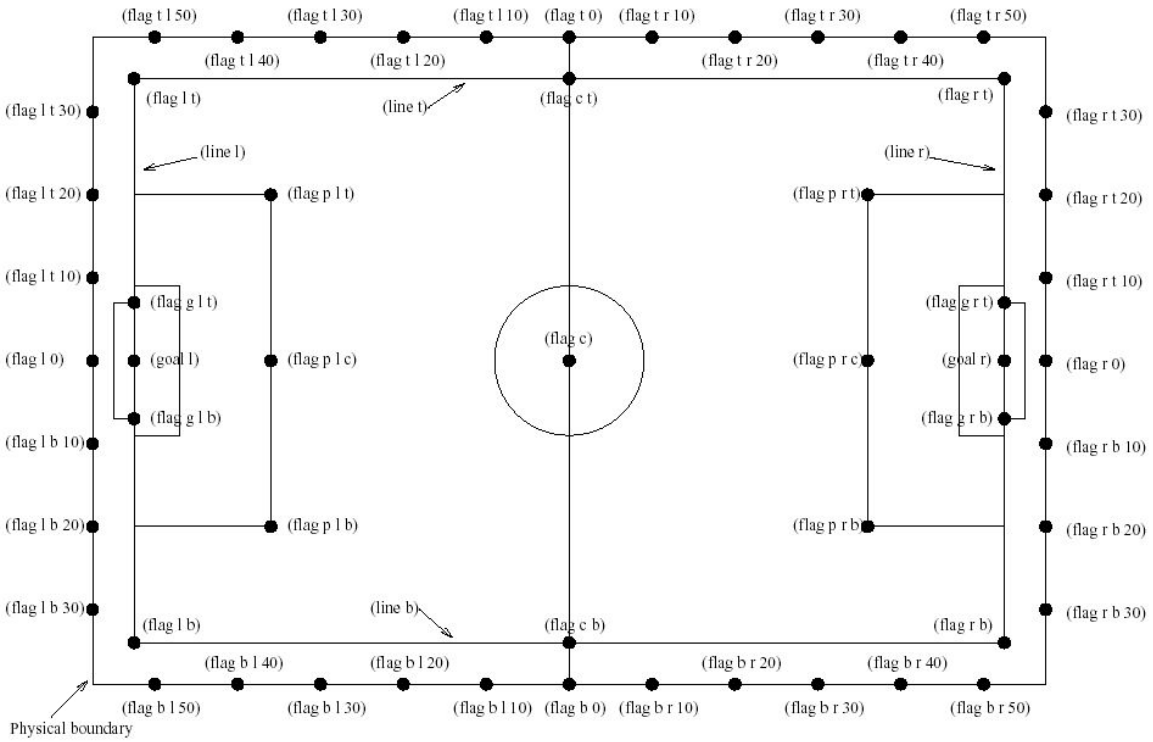


Figura 2.2: Posición de los objetos fijos del terreno de juego

Esta información se envía a los jugadores periódicamente, en ciclos de 150 milisegundos. Un jugador puede en cualquier momento elegir recibir información visual menos detallada, pero más a menudo. En ese caso, se le enviaría en la mitad de tiempo.

El formato de los mensajes será el siguiente:

`(see` *`Time ObjInfo`*`)`

Time indica el ciclo de simulación actual.

ObjInfo dá la información del objeto que se está viendo. Consta de un nombre de

objeto, seguido de más o menos información dependiendo del modo visual que tengamos seleccionado y de lo lejos que esté el objeto de nosotros.

Un ejemplo sería:

```
(see 647 ((p ‘‘ESCET’’ 5) 20.3 -9.4))
```

Que indicaría que en el ciclo 647 estamos viendo al jugador número 5 del equipo ESCET a 20.3 metros de distancia y a 9.4 grados a la izquierda del centro de nuestro cono de visión.

Información introspectiva

Se envía al jugador a petición de éste, y dá información sobre sí mismo, como puede ser su velocidad, hacia dónde está mirando su "cámara" con respecto a su cuerpo, su cansancio, etc.

El formato de estos mensajes es:

```
(sense body Time
(view mode ViewQuality ViewWidth)
(stamina Stamina Effort)
(speed AmountOfSpeed DirectionOfSpeed)
(head angle HeadDirection)
(kick KickCount)
(dash DashCount)
(turn TurnCount)
(say SayCount)
(turn neck TurnNeckCount)
(catch CatchCount)
(move MoveCount)
(change view ChangeViewCount))
```

Los parámetros son los siguientes:

Time es el ciclo de simulación actual.

ViewQuality y ViewWidth hacen referencia al nivel de detalle de información visual que está recibiendo el jugador, y a la apertura del cono de visión, respectivamente.

Stamina y Effort se refieren al nivel de cansancio del jugador. Cada jugador comienza con una cantidad de `Stamina`, que indica lo "fresco" que está. Con cada movimiento, el jugador la vá perdiendo, y en cada ciclo recupera una pequeña cantidad. Si el valor de `Stamina` llega a bajar demasiado, el jugador se moverá más despacio y chutará con menos fuerza.

AmountOfSpeed y DirectionOfSpeed informan de la velocidad del jugador y de la dirección de ésta, respectivamente.

HeadDirection es el ángulo que forma la "cámara" del jugador, con respecto a la dirección del cuerpo.

KickCount, DashCount, etc. Esta información son contadores para cada una de esas acciones, que el agente puede utilizar con el fin de controlar cuándo se han ejecutado las órdenes que envía.

2.3.2. Órdenes motrices

Cuando el agente quiere realizar alguna acción, enviará al servidor un paquete UDP con un mensaje en el que indica la acción que quiere realizar. Si este movimiento está permitido, el simulador hará que el jugador que representa al agente lo ejecute.

Los posibles mensajes de este tipo que pueden enviarse al servidor son:

Turn Gira el cuerpo del jugador.

`(turn` *`Moment`*`)`

Moment son los grados que queremos girar (positivo hacia la derecha y negativo hacia la izquierda).

Dash Acelera el jugador en la dirección de su cuerpo.

`(dash` *`Power`*`)`

La magnitud de la aceleración será la indicada por *Power*. Si es un valor negativo, será hacia atrás. El uso de este comando produce cansancio en el jugador, decrementando su valor de *Stamina*.

Kick Trata de chutar la pelota.

`(kick` *`Power Direction`*`)`

Se enviará la pelota en la dirección *Direction* (que será relativa al cuerpo del jugador), con una fuerza indicada por *Power*.

Catch Intenta coger la pelota.

`(catch` *`Direction`*`)`

Uno de los jugadores de cada equipo puede tomar el rol de portero. Si lo hace, podrá utilizar este comando para intentar coger la pelota en la dirección relativa a su cuerpo indicada por *Direction*. Si la pelota se encuentra en el rectángulo de un

metro de ancho por dos de largo que indica la figura 2.3, el portero podrá coger la pelota. Si la coge, ésta permanecerá en sus manos hasta que haga uso del comando *kick*.

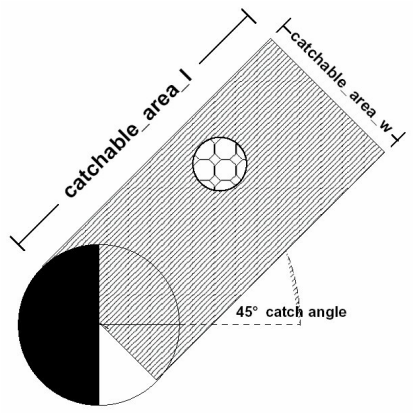


Figura 2.3: Funcionamiento del comando `(catch 45.0)`

Move Mueve instantáneamente al jugador a un punto del campo.

`(move` *X Y*`)`

El punto será el indicado por X e Y. Se utiliza principalmente para colocar a los jugadores en sus posiciones al comenzar el partido.

Este comando sólo puede utilizarse en el modo de juego *before_kick_off* y justo después de un gol.

El simulador sólo aceptará una de estas órdenes de movimiento cada ciclo de 100 milisegundos. Si se le envían más, ejecutará sólo una de ellas, generalmetne la recibida en primer lugar.

2.3.3. Otras órdenes

Turn Neck Hace girar el cono de visión del jugador.

`(turn_neck` *Angle*`)`

Se girará *Angle* grados con respecto al ángulo anterior. Como máximo, este ángulo alganzará 90 grados con respecto a la dirección del cuerpo, en uno u otro sentido.

Say Se usa para que los jugadores "hablen".

`(say` *Mensaje*`)`

Esto hace que el jugador "diga" el *mensaje* para que pueda oirlo cualquier jugador que esté lo suficientemente cerca, que lo recibirá como información auditiva.

Sense Body Solicita al servidor que envíe la información relativa al estado interno del jugador.

`(sense_body)`

Change View Cambia el modo de visión.

`(change_view` *`Width Quality`*`)`

Width será la anchura del cono, y *Quality* el nivel de detalle de la información recibida.

Capítulo 3

JDE

El presente proyecto ha de resolver un problema complejo, dinámico, con oponente y multiproceso. A la hora de abordar un problema de estas características es necesario optar por alguna técnica de inteligencia artificial. Estas técnicas permiten definir una base sobre la que apoyarnos para poder programar un agente que efectúe la tarea que queramos resolver.

Antes de explicar en qué consiste la arquitectura JDE, introduciremos tres principales técnicas de inteligencia artificial que se utilizan actualmente para implementar comportamientos.

Sistemas deliberativos

En sus comienzos, se abordó la inteligencia artificial desde un punto de vista deliberativo, es decir, el agente tenía que hacerse un plan a seguir para resolver el problema que se le planteaba. Para ello, primero necesitaba poseer un modelo detallado del mundo, aunque fuera de forma simbólica. Se razonaba sobre éste modelo, haciendo generalmente predicciones, y trazando así una línea de ejecución. Típicamente, estos comportamientos tienen tres fases: observación, planificación y ejecución (fig. 3.1).

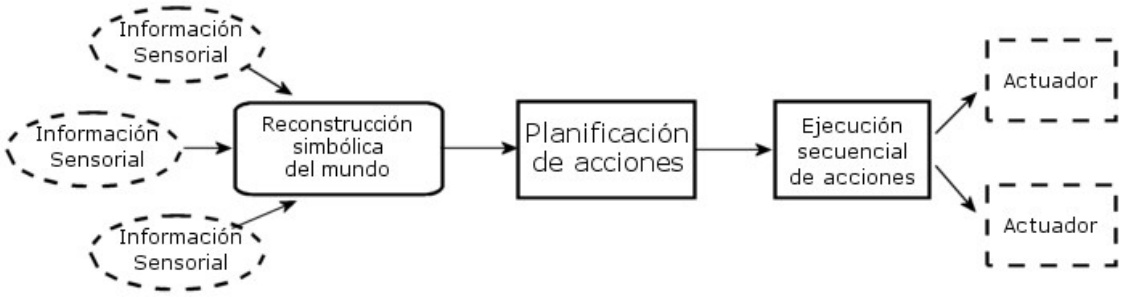


Figura 3.1: Sistemas deliberativos clásicos

Sistemas reactivos

En oposición a los sistemas deliberativos, los reactivos no usan un modelo del mundo, si no que razonan directamente sobre la información perceptiva, generando una salida inmediata dependiendo de las entradas sensoriales que tengan en un momento dado (fig. 3.2). Para ello, se divide el *espacio perceptivo* en conjuntos disjuntos, y se asigna a cada uno de ellos una respuesta adecuada. El camino entre observación y ejecución es directo. Tienen la ventaja de una gran rapidez de respuesta y una realimentación constante.

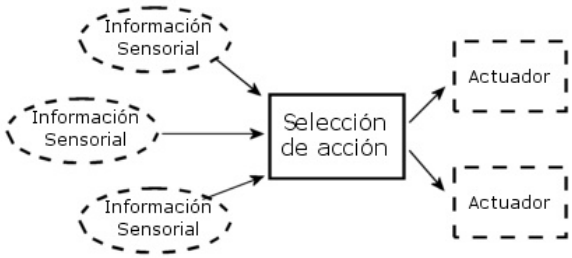


Figura 3.2: Sistemas reactivos

Sistemas basados en comportamientos

En los sistemas basadas en comportamientos, se divide el control en varios esquemas de comportamiento independientes. Cada uno de estos esquemas genera una salida motriz dependiendo de una o mas entradas sensoriales, para conseguir un objetivo concreto. Hace falta luego un mecanismo para elegir qué salida motriz será la que se utilice realmente, de entre las propuestas por los esquemas.

Figura 3.3: Sistemas basadas en comportamientos

3.1. Jerarquía Dinámica de Esquemas (JDE)

JDE [3] es una *arquitectura de control* basada en comportamientos, que se ha basado fundamentalmente en la ideas y trabajos de Michael Arbib [4] y de Ronald Arkin [5].

Se contempla el comportamiento como una combinación de percepción y control, aunque desarrollados separadamente. Ambos problemas se dividen en pequeñas partes con identidad propia, a las que se denomina *esquemas*, como se explicará a continuación. Además, la arquitectura propone que dichos esquemas se organicen jerárquicamente (de ahí su nombre) en un árbol que puede ir cambiando de forma, en tanto por los esquemas que lo componen como por las características específicas de los mismos, dependiendo de las condiciones de actuación necesarias en cada momento.

3.1.1. Esquemas

La idea del esquema como parte explicativa del comportamiento surge en el campo de la fisiología y la neurofisiología, y muchos investigadores apoyaron su integración en el ámbito de la robótica, muy especialmente Arbib [4] y Arkin [5].

En JDE, un esquema es un flujo de ejecución iterativo que tiene un objetivo concreto. Es modulable y puede ser activado y desactivado en cualquier momento. Hay esquemas perceptivos, que generan información a partir de los datos sensoriales o de la salida de otros esquemas, y esquemas motrices o de actuación que, basándose en esta información, producen una salida de movimiento con vistas a conseguir un objetivo concreto.

Los esquemas tienen asociado un estado. Los perceptivos pueden estar en estado `dormido`, que indica que no se encuentra en ejecución, o `activo`, que indica que está funcionando y actualizando la información perceptiva que genera. Los esquemas motrices son más complejos. Tienen cuatro estados: además de `dormido` y `activo`, que tienen un significado equivalente al de los esquemas perceptivos, tienen los estados `alerta` y `preparado`, que son necesarios durante la decisión de cuál de ellos toma el control en un momento dado, mediante los mecanismos explicados en el apartado 3.1.3.

La modulación de los esquemas se consigue mediante una serie de parámetros que modifican el comportamiento del esquema, para adaptarlo a la necesidad actual. El ajuste de estos parámetros lo realizará otro esquema (típicamente, aquel que le activó) o incluso puede en un momento dado hacerlo él mismo para adaptarse a una situación cambiante. Uno de los parámetros principales de todo comportamiento es su velocidad de iteración, que permite definir con cuánta frecuencia queremos que se ejecute.

La figura 3.4 muestra un esquema cuyo objetivo sería encarar el robot con la pelota.

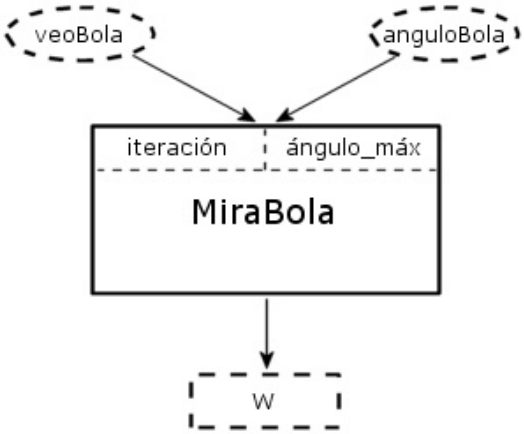


Figura 3.4: Ejemplo de esquema: MiraBola

Es un esquema motriz con dos parámetros: `iteración` y `angulo_máximo`. Su salida sería la velocidad angular que propondría dar a los motores para conseguir ponerse de frente a la pelota. Necesitaría acceder a ciertos datos generados por algún esquema perceptivo, como serían `veoBola`, que indicaría si estamos viendo la pelota, y `anguloBola` que indicaría el ángulo relativo a la pelota, en el caso de estar viéndola. El funcionamiento del esquema sería tan sencillo como enviar una velocidad angular alta a los motores si no vemos la pelota, para así girar rápidamente; si sí que estamos viendo la pelota y se encuentra a un ángulo mayor que `angulo_máximo`, entonces enviaríamos una velocidad angular más baja, en la dirección de la misma, para así mirar hacia ella.

3.1.2. Árbol jerárquico

Cada uno de estos esquemas, tal y como se explica en el apartado anterior, tiene un objetivo concreto pero sencillo, como por ejemplo acercarse a la pelota, o avanzar hacia un determinado sitio; tareas más complejas requieren combinar varios. Para conseguir esto con JDE, los esquemas se organizan a modo de árbol jerárquico. Los esquemas de nivel superior del árbol activan a otros esquemas (que pasarán a ser sus esquemas hijos) cuyos objetivos particulares ayudan a conseguir su objetivo propio. Estos esquemas hijos, a su vez, pueden necesitar de la actuación de otros esquemas de menor nivel, configurándose así una jerarquía de esquemas que implementan el comportamiento concreto que queremos conseguir.

La salida de un esquema motriz de los niveles más bajos será generalmente órdenes que se envían a los actuadores, mientras que la salida de los esquemas de niveles superiores será la activación de una serie de esquemas hijos. Estos esquemas hijos pueden ser tanto motrices como perceptivos.

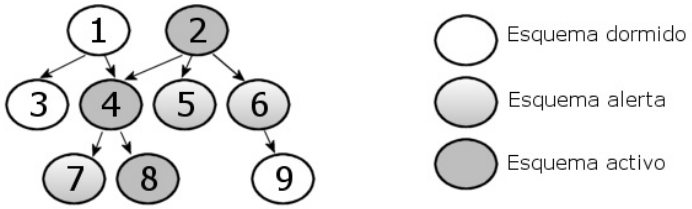


Figura 3.5: Ejemplo de árbol de esquemas JDE

En la figura 3.5 se muestra un ejemplo de árbol JDE. En ella, los esquemas 1 y 2 son los de mayor nivel, de los cuales el número 2 está `activo`. Este esquema necesita para su funcionamiento a los esquemas 4, 5 y 6, por lo que los pone en estado `alerta`, con lo que entrarán en una competición por el control en la que sólo uno de ellos pasará al estado `activo` (el 4 en la figura), dependiendo de las condiciones del entorno. El esquema 4 necesita a su vez al 7 y al 8, por lo que, siguiendo el mismo proceso anterior, los pone en estado `alerta`. Entre ellos, resulta ganador el número 8 y éste se activa.

Las configuraciones de los árboles de esquemas no son estáticas, si no que pueden cambiar según las necesidades de cada instante.

Por un lado, de entre los esquemas en estado `alerta` de un mismo nivel, puede ser necesario que se active uno u otro, con la consiguiente activación de sus hijos, y la desactivación de los hijos del esquema que estuviera activado anteriormente. De esta forma, el árbol de esquemas activos cambia a cada instante según las condiciones del entorno dicten la activación de uno u otro esquema motriz. En el ejemplo anterior, supongamos que, en un momento dado, el entorno dicta la activación del esquema 6; en este caso sería este esquema el que ganaría la competición por el control y se activaría, quedando el 4 en estado `alerta` (figura 3.6).

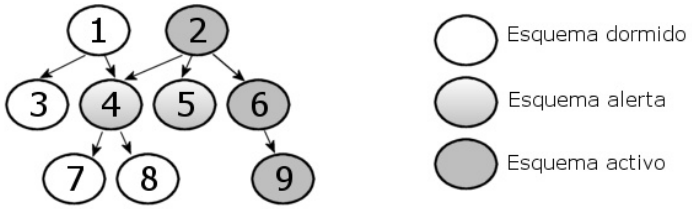


Figura 3.6: Ejemplo de reconfiguración de esquemas activos

Por otro lado, además, un mismo esquema padre puede decidir poner en `alerta` a hijos diferentes cada vez para alcanzar su objetivo, con lo que no sólo cambiaría el árbol de esquemas activos, si no que la forma de éste sería totalmente distinta. Siguiendo con el mismo ejemplo, el esquema 6, en el momento de activarse, podría decidir que el

esquema 8 es útil para sus propósitos en ese momento, por lo que pone en estado de alerta tanto al esquema 8 como al 9, ganando uno de ellos el control (el 8 en la figura 3.7).

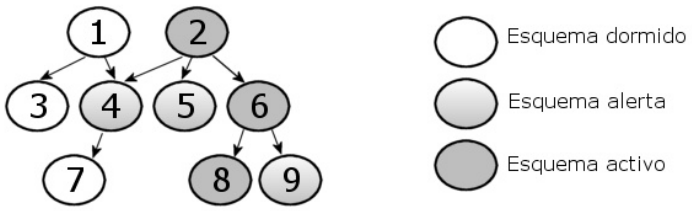


Figura 3.7: Ejemplo de reconfiguración morfológica del árbol

3.1.3. Selección de acciones

Cuando hay más de un esquema motriz despierto, los sistemas basados en comportamiento han de tener un mecanismo por el cual decidir qué acción en particular hay que llevar a cabo en cada momento, de entre las que podrían proponer cada uno de estos esquemas. Una solución podría ser que los esquemas "voten" por la acción que desean, y después se elija aquella que tenga más votos. Otra solución podría ser que la salida de cada esquema se sume a la salida final, que resultará ser una combinación de las proposiciones de cada esquema.

En el caso de JDE, se opta por permitir que sólo uno de los esquemas despiertos genere una salida en un momento dado. El problema, por tanto, pasa a ser elegir cuál de entre los esquemas despiertos (en estado `alerta`) pasará a estado `activo`, para que sean sus salidas las que se tengan en cuenta, ya sean éstas la activación de esquemas hijos u órdenes directas a los actuadores. Se utilizan dos técnicas para resolver esto: las precondiciones y el arbitraje.

Precondiciones

Como hemos comentado anteriormente, cuando un esquema motriz entra en ejecución, despierta a sus hijos y los pone en estado `alerta`. Cada uno de ellos, antes de intentar activarse realmente, tiene que comprobar que se dan las condiciones adecuadas para que actúe. A esto se le llama comprobar sus precondiciones.

Estas precondiciones, lo que hacen es dividir el espacio sensorial en distintos subconjuntos, y asociar la activación de cada esquema a cada uno de ellos. Así, cada esquema motriz tiene una función, denominada *precondition*, que comprobará ciertos valores de entre el espacio sensorial y devolverá `verdadero` o `falso`, indicando si se dan o no las condiciones de activación de ese esquema en particular.

Mientras un esquema esté despierto, comprobará su función *precondition* en cada ciclo de iteración y, si el resultado es `verdadero`, pasará al estado `preparado`, indicando así que está listo para generar una salida motriz. Si nadie más que él está en dicho estado en esa iteración, el esquema pasará al estado `activo` y empezará a ejecutar.

Arbitraje

Lo ideal es dividir el espacio perceptivo en subconjuntos disjuntos y totales, para que de ese modo, en un momento dado, haya uno y sólo uno de entre los esquemas motrices de cada nivel cuyas precondiciones se cumplan. Sin embargo, no siempre es esto posible, por lo que pueden producirse casos en los que las precondiciones no puedan resolver el problema de la selección de acción, pudiéndose producir dos situaciones problemáticas: el solapamiento de control y la ausencia de control (fig. 3.8).

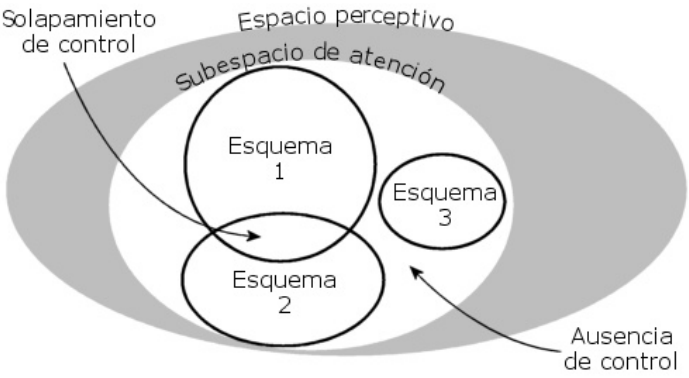


Figura 3.8: Espacio perceptivo y regiones de activación

Se produce un solapamiento de control cuando, en un momento dado, hay dos o más esquemas cuyas precondiciones se cumplen, lo que haría que más de un esquema quisiera actuar. Como JDE resuelve la selección de acción haciendo que sólo haya una proposición de acción posible, sólo podemos dejar actuar a uno de los esquemas. El caso de la ausencia de control se produce cuando la situación del entorno no está cubierta por ninguna precondición, por lo que ninguno de los esquemas motrices desea generar una salida.

Para tratar estos casos, JDE utiliza un sistema de arbitraje basado en el esquema padre. Cuando un esquema comprueba su función *preconditions*, también mira las de sus hermanos; si ninguna de ellas se cumple, pero la suya sí, toma el control y empieza a ejecutar. Si detecta que hay más de una función *preconditions* que es verdadera, estaría en el caso de un solapamiento de control, por lo que llamaría a una función especial, ubicada en el esquema padre, que decidirá cuál de los esquemas `preparados` debe activarse. El esquema también llamará a la función en el caso de que ningún esquema

quiera activarse en ese instante, al no cumplirse ninguna de las precondiciónes (ausencia de control). A esta función se la denomina función de arbitraje.

La función de arbitraje, en el caso del solapamiento de control, comprueba cuáles de los esquemas hijos quieren activarse, y con esta información (y quizá también información perceptiva) decidirá cuál de ellos lo hará, forzando su activación (estado `activo`) y devolviendo al resto al estado `alerta`.

En el caso de la ausencia de control, igualmente forzará la activación de alguno de los esquemas (aunque sea un esquema nulo, que no realice ninguna acción), ya que es necesario para el funcionamiento de la arquitectura que siempre haya algún esquema `activo`.

En este proyecto, utilizaremos JDE para diseñar cada jugador como un proceso formado por esquemas, que estarán organizados en forma de árbol siguiendo esta arquitectura. Como resultado, obtendremos once procesos, cada uno ejecutando su árbol de comportamientos, pudiendo introducir diferencias en los esquemas que los componen para lograr un equipo de jugadores.

Capítulo 4

Descripción Informática

Una vez explicado del funcionamiento del servidor del juego, y también las características de la arquitectura JDE, pasaremos a detallar la aplicación que hemos desarrollado. Haremos una primera explicación del análisis, el diseño (donde veremos la construcción del árbol de esquemas) y pasaremos después a la implementación.

4.1. Análisis y Metodología

En esta sección presentaremos un sencillo análisis de los requerimientos del programa y comentaremos la metodología de desarrollo que se ha seguido.

4.1.1. Análisis

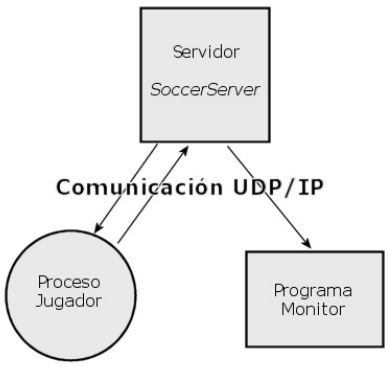


Figura 4.1: Esquema general del entorno de la aplicación

Como muestra la figura 4.1, tenemos un programa servidor proporcionado por la organización de la RoboCup, al cual se conecta un programa monitor que permite visualizar los partidos. Cada uno de nuestros procesos jugadores

se conectará directamente con el servidor, no poseyendo ellos ninguna clase de comunicación entre sí, si no es a través del mismo.

Requisitos

- El entorno de desarrollo será el sistema GNU/Linux utilizado en el laboratorio de robótica.
- El programa tendrá que poder conectarse con el servidor *Soccerserver*, y este utiliza paquetes UDP en sus comunicaciones con los programas cliente, por lo que se tendrán que utilizar sockets UDP en la parte de comunicaciones.
- Se seguirá la arquitectura JDE en el desarrollo del comportamiento del agente, ya que la prueba de dicha arquitectura, en el entorno de la liga simulada de la RoboCup, es el objetivo principal del proyecto.
- Se utilizará el lenguaje C ya que, a parte de ser el lenguaje de programación más extendido en los sistemas GNU/Linux, la implementación de JDE disponible está programada en dicho lenguaje.
- Dentro de lo posible, se intentará que el comportamiento de jugador de fútbol que se consiga sea eficiente desde un punto de vista competitivo, con vistas a formar un equipo de procesos capaz de jugar un partido contra equipos oficiales de forma satisfactoria.

4.1.2. Metodología

Como metodología de desarrollo se ha utilizado el modelo incremental. La implementación del comportamiento complejo de jugador de fútbol, que es uno de los requisitos, se subdivide en diversos comportamientos simples, como son chutar a puerta, pasar la pelota, avanzar hacia la portería, etc. Estos comportamientos se han añadido al sistema incrementalmente, siguiendo esta metodología, hasta llegar a configurar el comportamiento completo del agente, como se muestra en el esquema de la figura 4.2.

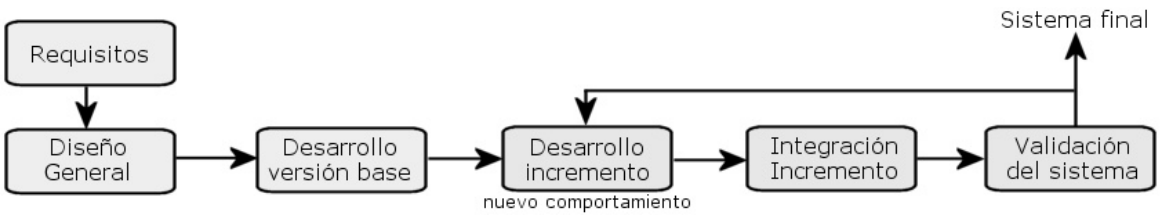


Figura 4.2: Metodología de desarrollo: modelo incremental

4.2. Diseño

Detallaremos ahora las características del diseño que se ha seguido para implementar la aplicación.

4.2.1. Diseño de un jugador

Teniendo en cuenta los requisitos, se desarrollará el programa siguiendo el esquema de la figura 4.3, en el cual se identifican las distintas partes que compondrá cada proceso.

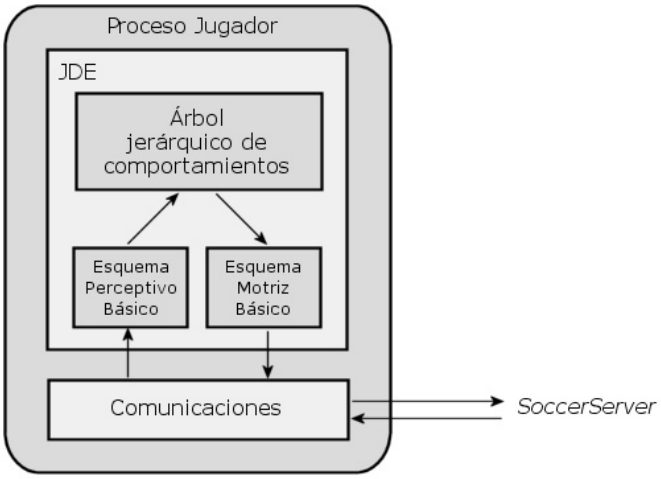


Figura 4.3: Diseño general de un proceso jugador

Tenemos dos grandes divisiones: la parte de generación de comportamiento, y la de comunicación con el servidor. Con respecto a la generación del comportamiento con JDE, se diferencian en la figura el árbol de esquemas jerárquico por una parte, y los esquemas sensorial y motriz básicos por otro. Estos últimos son en realidad esquemas de servicio y se encontrarán siempre en ejecución, sin tener que estar integrados en el árbol, ya que brindarán un entorno mínimo sobre el que programar cualquier comportamiento con JDE para actuar sobre el *Soccerserver*.

Las comunicaciones se dividen a su vez en dos partes. Por un lado hay que formar mensajes con las órdenes, en el formato del simulador, así como procesar los mensajes que envía éste. Por otra parte, hay que hacer llegar estos mensajes al servidor y recibir los que nos lleguen de él. Al no ser ésta una parte fundamental del proyecto, nos apoyaremos en una biblioteca de comunicaciones de las existentes para comunicarse con el *Soccerserver*.

4.2.2. Diseño del equipo

Para que los distintos jugadores que conforman un equipo jueguen en una zona del campo determinada, se optó por la definición de distintas zonas de referencia para cada rol que pueda tomar un proceso jugador determinado.

Cuando se lanza un proceso jugador, además de indicarle por línea de comandos el servidor al que debe conectarse y el nombre del equipo al que debe unirse, se le dirá también qué rol queremos que tome.

Dependiendo del rol que se le asigne, el proceso fijará la zona rectangular del terreno de juego en la que ha de quedarse el jugador, así como el punto dentro de dicho rectángulo donde ha de posicionarse al empezar y hacia donde ha de dirigirse cuando se salga de la zona. La figura 4.4 muestra las zonas de referencia de los distintos roles.

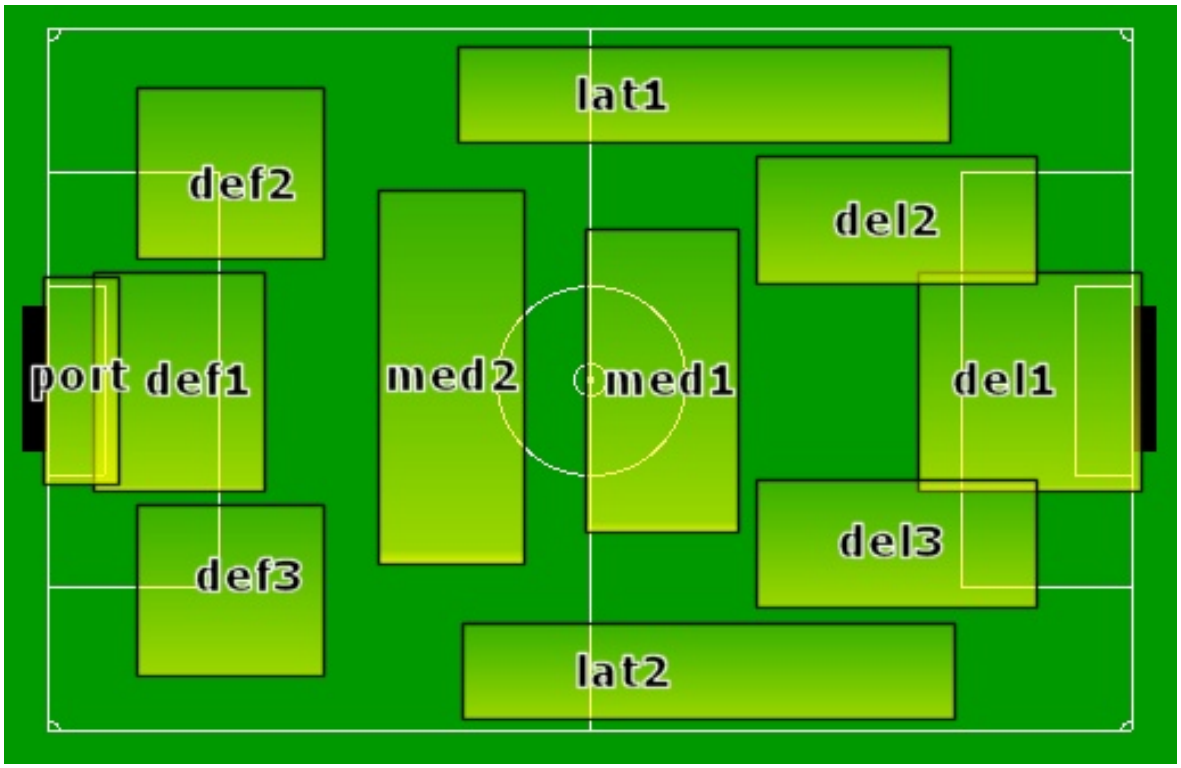


Figura 4.4: Zonas de referencia de los distintos roles de jugador

Los jugadores se mantendrán dentro de dichas zonas, a no ser que la pelota se acerque a menos de 10 metros de su posición, llegando con este margen a las zonas del terreno no cubiertas en la figura.

Cada uno de los jugadores será un proceso separado (pudiendo incluso ejecutarse en una máquinas distintas) y todos ejecutarán el mismo árbol de comportamientos, diferenciándose unos de otros únicamente por la zona de referencia correspondiente al rol que se les haya asignado a cada uno.

4.2.3. Diseño del árbol de comportamientos

Una vez visto el diseño general, pasaremos a definir la primera de las partes que componen un proceso jugador: el árbol de esquemas que configura el comportamiento de jugador de fútbol. Empezaremos por la parte superior del árbol e iremos bajando hasta llegar a los niveles inferiores. Por cada esquema enunciaremos su objetivo, detallaremos los posibles parámetros que tenga, describiremos el subconjunto del espacio sensorial ante el cual ha de reaccionar e indicaremos cuales son sus salidas.

Nivel raíz de la jerarquía

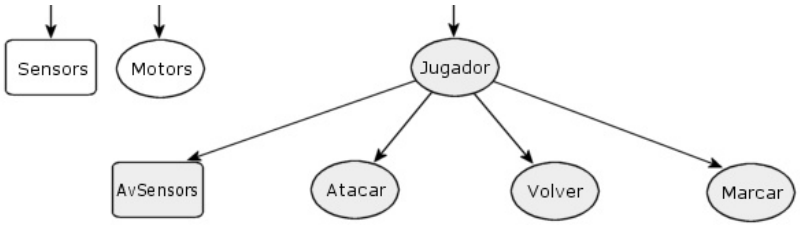
Este es el nivel bajo el cual se despliega todo el árbol JDE de comportamientos. También explicaremos aquí los esquemas de servicio de actuadores y sensores, ya que no están ligados al árbol. Estos esquemas estarán constantemente activos, sin tener que competir por el control.

Sensors : Este es un esquema de servicio y no está ligado al resto de la jerarquía. Es un esquema perceptivo, y sus salidas configuran el espacio sensorial del que dispondrá el agente. Pone todos los datos sensoriales a disposición de los esquemas del árbol de comportamientos. Se explicará con detalle en la sección de implementación (4.2.4).

Motors : Otro esquema de servicio. Es un esquema motriz y sus salidas son órdenes que se envían al servidor en forma de mensajes UDP, siendo éste el único esquema que se comunica directamente con él. Pone a disposición de los demás esquemas una serie de variables donde estos escribirán para solicitar una acción de movimiento. También se explicará en profundidad en la sección 4.2.4.

Jugador : El esquema que engloba el comportamiento de jugador de fútbol que hemos implementado. Al activarse despierta a sus hijos para que compitan por el control, los cuales componen el siguiente nivel.

Nivel jugador



El siguiente nivel del árbol es el formado por los esquemas hijos de *Jugador*. Se compone de un esquema sensorial y tres motrices.

AvSensors : Esquema perceptivo que extrae información de las salidas de *Sensors* para uso de los demás esquemas de este nivel y siguientes. Se explicará en detalle en la sección 4.3.

Atacar : Este esquema reúne los comportamientos necesarios para conseguir la pelota y tratar de meter gol con ella.

Se activa cuando el jugador está dentro de su zona de referencia, o bien está cerca de la pelota (menos de diez metros), a no ser que, según el modo de juego (cuadro 2.1), le toque sacar al equipo contrario. Cabe recordar que cada jugador ejecuta su propia instancia del árbol de comportamientos, por lo que es perfectamente normal que un jugador tenga activo este esquema (es decir, esté atacando) y otro no. Como esquema padre que es, despertará a sus hijos, y los pondrá en estado `alerta` para que compitan por el control.

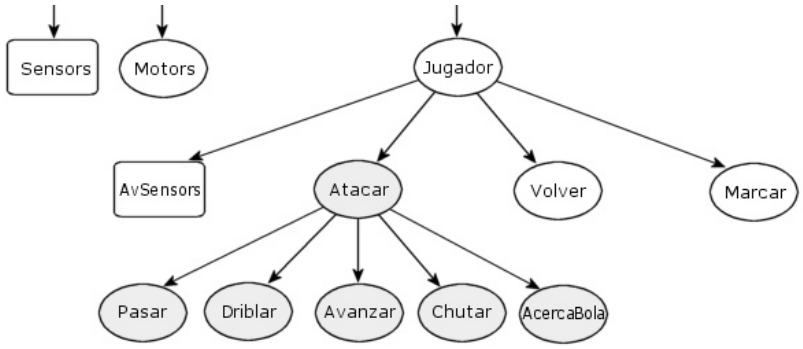
Volver : Hace que el jugador vuelva a su posición de referencia (ver sec. 4.2.2).

Se activa cuando el jugador se ha alejado demasiado de su posición y está lejos de la pelota (o no la ve), haciendo que gire y se dirija a su punto de origen. La dirección en la que se encuentra dicho punto se calcula utilizando una estimación de la posición actual proporcionada por el esquema *Sensors*.

Marcar : Hace que el jugador se acerque al contrario más cercano para marcarlo.

Se activa cuando el modo de juego indica que va a sacar el equipo contrario (saques de banda, faltas, etc.). Si no se ve a ningún contrario cerca, el esquema lo buscará haciendo que el jugador gire sobre sí mismo.

Nivel atacar



Se explicará aquí la parte del árbol formada por los esquemas hijos de *Atacar*. Son ya esquemas de comportamiento básicos, que en la mayoría de los casos producen una salida a los actuadores por sí mismos. Este nivel representa el cuerpo principal del comportamiento del agente. La información perceptiva necesaria para los esquemas de este nivel e inferiores se extraen de las salidas de los esquema *Sensors* y *AvSensors*.

Pasar : Pasa la pelota a un compañero que se encuentre por delante, de camino a la portería contraria.

Su activación se produce cuando, teniendo la pelota a sus pies (distancia menor de 1 metro), el jugador detecta que hay un compañero por delante suya. Se entiende que está por delante suya si está a una distancia de entre 5 y 30 metros de sí mismo, y la diferencia de ángulo con respecto a la portería contraria es menor de 45 grados.

Al ejecutarse, este esquema realiza el pase dando orden de chutar en dirección al compañero, con una fuerza proporcional a la distancia a la que se encuentre éste.

Driblar : Trata de esquivar a los jugadores contrarios, mientras se intenta avanzar con la pelota, realizando un autopase.

- *rc_driblar_kick* : Fuerza con la que queremos golpear para hacer el autopase.
- *rc_driblar_angulo*: Máxima diferencia entre ángulo con el que se ve al contrario y con el que se vé la portería, para que el esquema se active.
- *rc_driblar_distContrario*: Distancia máxima a la que tiene que estar el jugador contrario para que se active el esquema.

Entrará en acción cuando, estando en posesión de la pelota, se detecta un jugador del equipo contrario por delante nuestra. Se entiende que está por delante si se encuentra a menos de *rc_driblar_distContrario* metros, y la diferencia de ángulo con respecto a la portería contraria es menor de *rc_driblar_angulo* grados.

Este es un esquema compuesto. Tiene dos hijos: *Driblar1* y *Driblar2*, pero no los activa simultáneamente como hace *AcercaBola*, si no que los usa para implementar estados, ya que la consecución completa del comportamiento implica la ejecución secuencial de varias acciones:

- En el estado 0, activa a su hijo *Driblar1*, que se encarga de chutar en una dirección a 45 grados del contrario y con fuerza *rc_driblar_kick* (para lo cual lo parametriza con este mismo valor), pasando después al estado 1.
- En el estado 1, comprueba que la pelota se haya separado de nosotros, lo que le indica que las órdenes de su hijo ya se han ejecutado. Cuando esto se produzca, duerme a *Driblar1*, activa a su otro hijo *Driblar2*, y pasa a estado 2. *Driblar2* comanda a los motores una velocidad muy alta, y va corrigiendo la trayectoria para llegar hasta la pelota.
- En el estado 2, comprueba que hayamos vuelto a recuperar la pelota (distancia a la misma menor de 1 metro) o bien que la hayamos perdido de vista. Cuando esto sucede, se supone terminado el comportamiento, por lo que se duerme a *Driblar2* y se vuelve a pasar a estado 0.

Se han explicado aquí los esquemas *Driblar1* y *Driblar2* aunque componen un nivel distinto en la jerarquía ya que, dada la naturaleza de los mismos, no tiene mucho sentido desligarlos de su esquema padre *Driblar*.

Avanzar : Avanzar con la pelota hacia la portería contraria.

- *rc_avanzar_kick* : Fuerza con la que queremos golpear la pelota a cada paso.
- *rc_avanzar_limite*: distancia a la portería hasta la que queremos acercarnos.
- *rc_avanzar_distContrario*: distancia mínima al contrario más cercano para que siga activado este esquema.

Se activa cuando vemos la bola a menos de un metro de nosotros, la portería contraria está a más de *rc_avanzar_limite* metros (o bien no la vemos), no hay de camino a la portería ningún compañero por delante nuestra (en cuyo caso sería preferible pasarle la pelota) ni tampoco ningún contrario a menos de *rc_avanzar_distContrario* metros en nuestro camino (en cuyo caso sería mejor esquivarle).

Su salida es parecida a la de *chutaPuerta*, solicitando el envío de un comando `Kick` con fuerza *rc_avanzar_kick* (que será poca fuerza) y ángulo igual a la dirección a la portería. Como no es requisito que estemos viendo la portería para la activación de este esquema, en el caso de no verla se calcula su dirección, usando para ello

la estimación de tenemos de nuestra posición que nos proporciona el esquema *Sensors*.

El efecto de avance se produce al alternarse la activación de este esquema con *AcercaBola*. Al alejarse la pelota del jugador, se activa el esquema *AcercaBola* para acercarse a ella. Una vez cerca de ella, se activa *Avanzar*, haciendo que la pelota se aleje de nuevo, y así sucesivamente.

Chutar : Su objetivo es tratar de meter gol.

- *rc_chutaPuerta_kick* : Fuerza con la que queremos chutar.
- *rc_chutaPuerta_limite*: distancia a la portería a la que queremos estar como máximo para chutar.

Se activará cuando la pelota esté a distancia de tiro (a menos de 1 metro del jugador) y además la portería contraria esté a una distancia menor de *rc_chutaPuerta_limite*.

Produce el envío de un comando `Kick`, con fuerza igual a *rc_chutaPuerta_kick* y ángulo igual a la dirección del centro de la portería.

AcercaBola : Localiza la pelota y se acerca hasta ella.

Se activará cuando no se esté viendo a la pelota o bien ésta se encuentre lejos de nosotros.

Su salida es activar simultáneamente a sus dos esquemas hijos, *miraBola* y *sigueBola*, para que compitan por el control. El subconjunto del espacio perceptivo ante el cual reacciona este esquema se subdivide de forma total entre sus dos hijos, tal y como muestra la figura 4.5.

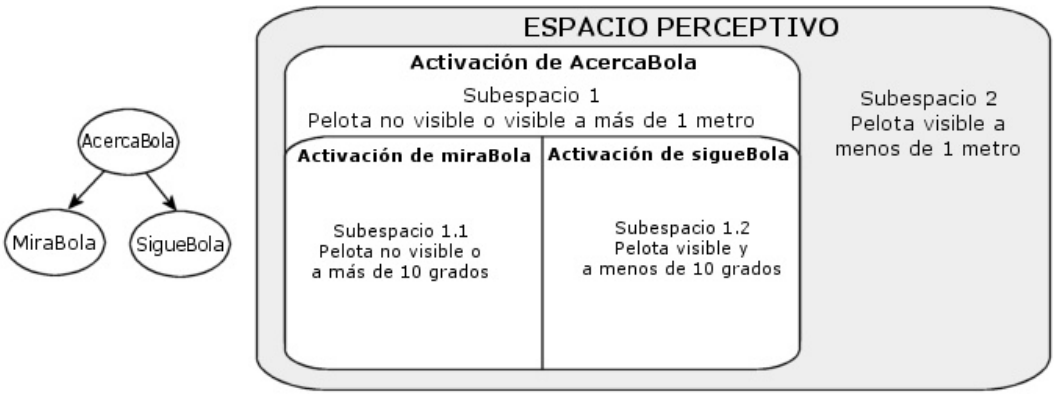
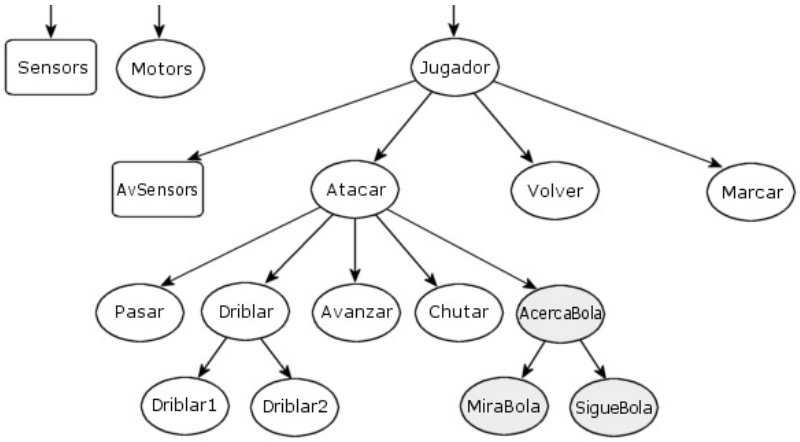


Figura 4.5: Ejemplo de subdivisión del espacio perceptivo entre esquemas hijos

Este es un comportamiento apetititivo, es decir, en lugar de satisfacer una necesidad (pasar el balón a un compañero, marcar gol), favorece la aparición de un estímulo que hará que se disparen otros comportamientos. Este estímulo es que la pelota se encuentre "a los pies" del jugador.

Nivel AcercaBola



Este es el nivel compuesto por los esquemas hijos de AcercaBola, los cuales entran en `alerta` cuando se activa su padre, consiguiendo con su comportamiento individual la consecución del objetivo del padre: acercarse a la pelota allá donde esté.

miraBola : Busca la pelota y gira hasta quedarse mirando hacia ella.

Se activará cuando no tengamos la pelota a la vista, o bien se la vea con más de 10 grados de ángulo con respecto a nosotros.

En el caso de no ver la pelota, comanda girar 80 grados para tratar de encontrarla. Si la pelota está localizada, gira tantos grados como se la vea desviada.

sigueBola : Sigue a la pelota hasta acercarse lo suficientemente a ella para chutar.

- *rc_sigueBola_v* : Velocidad a la que se desea seguir a la pelota.

Para poder actuar, necesita que la pelota esté localizada, y se la vea con una desviación de menos de 10 grados del centro del cono de visión.

Su salida es comandar una velocidad igual a *rc_siguebola_v*, así como hacer las correcciones de rumbo necesarias para seguir a la pelota.

4.2.4. Percepción y Actuación básica

Los esquemas que forman parte del arbol jerárquico de comportamientos, anteriormente descrito, necesitan una base perceptiva en la que apoyarse, así como

la posibilidad de que sus salidas de movimiento se materialicen en forma de órdenes enviadas a los actuadores del agente (en este caso, el servidor/simulador). Se describen en este apartado los esquemas de servicio que se encargan de esto: *Sensors* y *Motors*, así como el esquema perceptivo *AvSensors*.

Sensors

Éste es el esquema perceptivo básico de cada agente. Pone a su disposición la información sin tratar, proveniente directamente de los sensores del agente (en este caso, los mensajes enviados por el servidor *Soccerserver*). Tiene un ciclo de ejecución de 50 milisegundos.

Información generada:

tObjeto rc_bola : Información relativa a la posición de la pelota.

tObjeto rc_miPuerta : Información relativa a la posición de la portería de nuestro equipo.

tObjeto rc_suPuerta : Información relativa a la posición de la portería del equipo contrario.

tObjeto rc_Jugadores[22]: Array con la información de los jugadores que estamos viendo.

tObjeto Es un tipo de dato formado por lo siguiente:

`antiguo`	indica si los siguientes datos son información antigua o actual.
`equipo`	indica si es de nuestro equipo o del contrario.
`dorsal`	número de dorsal (sólo para jugadores).
`dir`	dirección del objeto con respecto a nosotros.
`dist`	distancia al objeto.
`dir_c`	cambio de la dirección en el tiempo.
`dist_c`	cambio de la distancia en el tiempo.

char *rc_playMode : cadena de texto descriptiva del modo de juego actual (ver tabla 2.1).

long nciclo : contador del número de mensajes con información perceptiva recibidos. Se usa en algunos esquemas como sello de tiempo, para saber cuándo se han actualizado los datos perceptivos.

rc_miPosX : Estimación de nuestra coordenada X en el campo, basada en la información visual.

rc_miPosY : Estimación de nuestra coordenada Y en el campo, basada en la información visual.

rc_miDir : Estimación de nuestra dirección, basada en la información visual.

Los tres últimos datos se encuentran en este esquema, a pesar de ser información calculada, porque la función *estimate_current_pos* de la biblioteca *libsclient* necesita acceder a todos los datos visuales que envía el servidor, los cuales únicamente están disponibles, de forma local, en este esquema.

avSensors

Este esquema perceptivo genera información elaborada a partir de la salida del esquema *Sensors*. Tiene un ciclo de ejecución de 50 milisegundos.

Información generada:

char rc_veoContrario : Indica si estamos viendo a algún jugador contrario.

char rc_contrarioDir : Indica la dirección al contrario más cercano.

char rc_contrarioDist : Indica la distancia al contrario más cercano.

char rc_veoAmigo : Indica si estamos viendo a algún compañero.

char rc_amigoDir : Indica la dirección al compañero más cercano.

char rc_amigoDist : Indica la distancia al compañero más cercano.

La información anterior se consigue recorriendo el array *rc_Jugadores* generado por el esquema *rc_sensors*.

char rc_enZona1 : Indica si estamos dentro de nuestra zona de referencia, es decir, si nuestra posición actual cae dentro del rectángulo delimitado por el rol que haya tomado el jugador (ver sección 4.2.2).

rc_hayQueVolver : Indica si nuestra situación y la del partido hacen necesario volver a nuestra posición de referencia. Será necesario volver cuando hayamos salido de nuestra zona de referencia y no veamos la pelota cerca, o bien cuando no estemos en el modo de juego *play_on* o en alguno que indique que tenemos que sacar nosotros.

Esquema motriz básico: Motors

Examinaremos ahora el esquema motriz básico, es decir, aquel que se comunica directamente con los actuadores (en este caso el servidor *Soccerserver*) para enviar las órdenes de movimiento que generan los demás esquemas.

Motors es el esquema motriz básico de nuestra implementación. Cuando este esquema es iniciado por primera vez, establece e inicializa la comunicación con el servidor mediante las funciones de librería *init_connection* y *send_com_init* explicadas en la sección 4.3.3. Para establecer la conexión, este esquema está parametrizado con las siguientes variables:

rc_servidor : cadena de texto que ha de indicar el nombre o la dirección IP de la máquina donde se está ejecutando el *Soccerserver*.

rc_equipo : cadena de texto con el nombre del equipo al que queremos unirnos para jugar.

Tras el establecimiento de la conexión, y aunque no es un esquema perceptivo, nos proporciona dos variables que otros esquemas pueden necesitar consultar. Estas variables son:

rc_socket : el *socket* de la conexión con el servidor. Además de por este mismo esquema, también es utilizado por *Sensors*.

rc_bando : indica si nos toca jugar en el lado derecho o izquierdo del campo.

Éste esquema tiene un ciclo de ejecución de 50 milisegundos, y proporciona una serie de variables, en las cuales pueden escribir los demás esquemas para generar un movimiento en el jugador. A continuación describiremos estas variables.

rc_dash : cuando se encuentre un valor distinto de 0, se enviará el comando **Dash** con *Power* igual al contenido de la variable. Tras hacerlo, se vuelve a poner a 0.

rc_turn : cuando se encuentre un valor distinto de 0, se enviará el comando **Turn** con *Moment* igual al contenido de la variable. Tras hacerlo, se vuelve a poner a 0.

rc_kick : cuando se encuentre un valor distinto de 0, se enviará el comando **Kick** con *Power* igual al contenido de la variable y *Angle* igual al contenido de la variable *rc_kick_dir*. Tras hacerlo, se vuelven a poner ambas a 0.

rc_kick_dir : ver descripción de *rc_kick*.

rc_turn_eye : cuando se encuentre un valor distindo de 0, se enviará el comando **Turn Neck** con *Moment* igual al contenido de la variable. Tras hacerlo, se vuelve a poner a 0.

rc_moveX y rc_moveY : cuando alguna de las dos contenga un valor distinto de 0, se enviará el comando **Move** con X e Y igual al contenido de *rc_moveX* y *rc_moveY*, respectivamente. Tras hacerlo, se vuelven a poner ambas a 0.

rc_v y rc_w : utilizadas para simular un *control en velocidad* de los comandos **Move** y **Turn**, tal y como se explica en la siguiente sección.

4.3. Implementación

Explicaremos ahora en qué consiste la implementación de JDE que hemos utilizado, para pasar a explicar a continuación cómo se programa un esquema usando la misma. Veremos también la adaptación que fue necesaria implementar sobre el esquema motriz básico para una eficiente actuación en el *SoccerServer* de los demás esquemas implementados. Por último, describiremos la biblioteca de comunicaciones que hemos utilizado.

4.3.1. Construcción de esquemas con nuestra implementación de JDE

La biblioteca de JDE que se nos ha proporcionado para la programación de este proyecto consta de una serie de ficheros de código fuente, en lenguaje C, que contienen las funciones y procedimientos que implementan la arquitectura descrita a lo largo del capítulo 3.

El código fuente de cada esquema se define en un fichero, a partir de una plantilla. Estas plantillas contienen los procedimientos que controlan la concurrencia de los esquemas, la comprobación de la función *precondition*, etc.

Primero se diferenciará si el esquema que se quiere programar es perceptivo o motriz, ya que las plantillas son distintas. Tras hacer una copia de la plantilla adecuada, procederemos a definir los parámetros del esquema y a programar las funciones contenidas en la misma.

Estas funciones son las siguientes:

preconditions : Esta función ha de comprobar las precondiciones de activación del esquema. Devolverá `verdadero` o `falso`, tras comprobar las variables que forman las salidas de los esquemas perceptivos, para indicar si es el momento de actuar.

Sólo está presente en los esquemas motrices, ya que los perceptivos siempre estarán activos, una vez despiertos.

iteration : El comportamiento en sí del esquema se implementará en este procedimiento. En cada iteración en la que el esquema esté activo (siempre, en el caso de los perceptivos y cuando gane la competición por el control, en el caso de los motrices), será invocado para que el esquema produzca una salida.

arbitration : Esta es la función de arbitraje explicada en la sección 3.1.3. Es un procedimiento que está presente en los esquema motrices con hijos y su cometido es, como ya se ha dicho, decidir cuál de sus hijos tiene que actuar en el caso de producirse un solapamiento de control o bien una ausencia del mismo.

startup : Utilizado para inicializar el esquema. Cualquier código de inicialización que sea necesario se escribirá aquí.

resume : Este es el procedimiento invocado para despertar a un esquema. Los esquemas motrices pasarán a estado `alerta` y los perceptivos a estado `activo`. Si el esquema tiene hijos que desee lanzar simultáneamente para competir por el control, llamará desde aquí a sus respectivas funciones `resume` tras ajustar sus parámetros.

suspend : Cuando se quiera dormir al esquema, éste será el procedimiento al que se llamará. Se debe encargar de poner al esquema en estado `dormido` y de dormir a su vez a los posibles hijos que tenga.

thread : Por último, este es el procedimiento central de hilo de ejecución de cada esquema. Entre otras cosas, se encarga de comprobar periódicamente el estado de la función *precondition*, llamando en su caso a *iteration*, o a la función *arbitration* del esquema padre cuando sea necesario. En principio, no es necesario cambiar nada de dicho procedimiento, dejándolo tal cual está en la plantilla.

Cada uno de los esquemas presentados en el apartado 4.2.3 se ha implementado, por tanto, en un fichero que contiene la programación específica de estas funciones para cada esquema.

A modo de ejemplo, se adjunta el código fuente de un esquema como anexo a esta memoria.

4.3.2. Adaptación al *Soccerserver*

A la hora de implementar el movimiento del jugador, fue necesario realizar ciertas adaptaciones en el módulo motriz básico, para que funcionara eficientemente con el servidor. El problema es que el *Soccerserver* implementa un control que se podría considerar "en posición" para el movimiento de los jugadores, mientras que JDE estaba más pensado para utilizar un control "en velocidad". Esto, unido a la falta de sincronización entre los esquemas perceptivos y motrices, hizo necesario controlar el ritmo de envío de órdenes al servidor.

Primero se trató de controlar esto haciendo que los esquemas esperaran a que se actualizaran una o dos veces la información perceptiva antes de escribir nuevas órdenes en las variables de *Motors*. Esto no era eficiente, además de que suponía llevar ese control en todos y cada uno de los esquemas, así que se optó por controlarlo todo desde *Motors*.

Por un lado, se simuló un control en velocidad, añadiendo la variable *rc_v*, y haciendo que los esquemas escribieran en ella en lugar de en *rc_dash*. Cuando *Motors* encuentra un valor en *rc_v*, lo envía al servidor, pero no pone la variable a cero, dejándola tal cual está. Así, en las siguientes iteraciones, *Motors* enviará de nuevo su contenido hasta que algún esquema quiera parar los motores, escribiendo un 0 en la variable. De esta forma, los esquemas generarán como salida una velocidad lineal en lugar de una aceleración como hacían antes. Con esto se consigue que no importe en qué momento o cuantas veces se escriba un valor en la variable, porque sólo se enviará su valor cuando lo estime apropiado el esquema *Motors*. De forma análoga, se añadió la variable *rc_w* para implementar un control en velocidad de los giros (variable *rc_turn*).

Por otro lado, como se comentó en la sección 2.3.2, el servidor sólo acepta una orden motriz por cada ciclo de 100 milisegundos, descartando todas las demás que se reciban en ese tiempo. Sucedía muy a menudo que coincidían en una misma iteración de *Motors* las órdenes motrices *Turn* y *Dash*, y en ocasiones alguna otra, y hasta ese momento lo que se hacía era enviarlas todas. Esto hacía que sólo una de ellas se ejecutara y el resto fueran ignoradas, con el consiguiente efecto negativo en el control del jugador.

Para solucionar esto, se controla en qué instante se envió la última orden motriz, de tal forma que antes de enviar una nueva, se comprueba que hallan transcurrido más de 100 milisegundos, y en caso contrario, no se envía la orden aún, esperando a la siguiente iteración de *Motors* para volver a comprobarlo. Como añadido, se guarda también de qué tipo ha sido la última orden motriz que se ha enviado, de forma que se van alternando las órdenes *Turn* y *Dash*; en cuanto al resto de órdenes, se supone

que son prioritarias *Catch* y *Kick* sobre *Turn* y *Dash*, enviándose cualquiera de las dos primeras (*Catch* antes que *Kick*) en cuanto estén disponibles, sin importar que haya alguna de las dos últimas esperando.

Con todo esto, se ha conseguido un esquema motriz básico para la arquitectura JDE, que resulta eficiente para ser usado con el servidor *Soccerserver*, a la vez que cómodo de usar desde los demás esquemas motrices, a pesar de las peculiaridades de los "actuadores" que poseemos en este caso.

4.3.3. Comunicación con el servidor

Esta parte es la encargada de hacer llegar al servidor las órdenes de movimiento que se generen en el esquema de servicio *Motors*, así como recoger los paquetes de información sensorial que necesita el esquema *Sensors* para formar el espacio perceptivo.

El *Soccerserver* pone a disposición de los procesos jugadores el puerto 6000 UDP para que se comuniquen con él, utilizando para ello los mensajes que se explicaron en la sección 2.3.

Dado que esta parte de la aplicación no es de interés para el objetivo del presente proyecto, utilizaremos una biblioteca de comunicaciones externa para implementar esta funcionalidad. Esta biblioteca cliente, llamada *libsclient*, nos facilita la comunicación con el servidor, proporcionándonos funciones para establecer la comunicación, enviar mensajes con nuestras órdenes en el formato del servidor, y traducir la información que envía éste a una estructura de datos que luego podemos usar más cómodamente.

Biblioteca *libsclient*

A continuación se describen las distintas funciones de biblioteca que hemos utilizado:

init_connection Crea el *socket* que se usará en el resto de las comunicaciones.

```
Socket init_connection(char *host, int port)
```

`host` nombre o dirección IP de la máquina donde se ejecuta el servidor
`port` puerto UDP donde escucha el *Soccerserver* (6000 por defecto)

Se ha de usar antes de ninguna otra función de la librería.

send_com_init Inicializa la conexión de un jugador con el servidor.

```
InitInfo send_com_init(Socket *sock, char *teamname, int version,
int role);
```

`sock`	*socket* de la conexión con el servidor
`teamname`	cadena de texto que contiene el nombre del equipo al que deseamos unirnos
`version`	versión del lenguaje de mensajes que queremos usar con el servidor.
`role`	rol que tomará el agente. Podrá ser portero o jugador.

`InitInfo` es una estructura que contiene el lado del campo donde juega el equipo, el dorsal del jugador y el modo de juego en el que se encuentra el simulador (todo ello información enviada por el servidor).

Es la primera comunicación que hará cada jugador, y servirá para anunciarse en el juego. Cuando el servidor recibe un mensaje de este tipo, primero comprobará si está jugando algún equipo con ese nombre, si es así, asignará a ese jugador un número de dorsal consecutivo (empezando por el 1 hasta el 11). Si no existe ese equipo, y suponiendo que aún no haya dos equipos jugando, creará un equipo con ese nombre y asignará al jugador el dorsal 1. Si ya hay dos equipos jugando, y ninguno de ellos tiene el nombre que ha indicado el jugador, el servidor devolverá un mensaje de error e ignorará la petición del cliente.

receive_info Recoge un paquete de información enviada por el servidor.

```
Bool receive_info(Socket *sock, char *buffer, int size)
```

`sock`	*socket* de la conexión con el servidor
`buffer`	zona de memoria donde queremos que se guarde el mensaje del servidor.
`size`	tamaño del buffer.

Si hay algún paquete esperando en el *socket*, esta función lo copiará en el *buffer* y devolverá VERDADERO. En caso contrario devolverá FALSO y continuará la ejecución (función no bloqueante).

scan_info Extrae información sensorial de un mensaje del servidor.

```
Bool scan_info(char *buffer, SensorInfo *sinfo, char *teamname)
```

`buffer`	zona de memoria donde está almacenado el mensaje del servidor.
`sinfo`	estructura de datos que almacena múltiple información sensorial.
`teamname`	nombre del equipo para el cual queremos extraer información.

Devuelve VERDADERO si hay algún mensaje en *buffer* que haya podido analizar, FALSO en caso contrario. La información extraída del mensaje se almacena en la variable *sinfo*, la cual almacena el tipo de información que contiene (visual, auditiva o introspectiva), mas una estructura con la información en sí, que tendrá el siguiente formato:

Información auditiva:

`int time`	ciclo de simulación en el que se produce el mansaje.
`SenderType sender`	emisor del mensaje (*ST_Referee*, *ST_Player*, *ST_Self*).
`double direction`	dirección de donde viene el mensaje.
`char message[BUFSIZE]`	cadena de texto con el mensaje.

Información introspectiva:

`int time`	ciclo de simulación en el que se produce el mansaje.
`ViewWidth viewwidth`	cono de visión (*VW_Narrow*, *VW_Normal* o *VW_Wide*).
`ViewQuality viewquality`	nivel de detalle de la visión (*VQ_high* o *VQ_low*).
`double stamina`	nivel de *Stamina* del jugador.
`double effort`	valor de *effort* del jugador.
`double speed`	velocidad a la que se está moviendo el jugador.
`int kick`	número de comandos *kick* que ha ejecutado el jugador.
`int dash`	número de comandos *dash* que ha ejecutado el jugador.
`int turn`	número de comandos *turn* que ha ejecutado el jugador.
`int say`	número de comandos *say* que ha ejecutado el jugador.

Información visual:

En el caso de la información visual, el servidor envía todos los objetos que ve el jugador en un sólo paquete. Así, tenemos una primera estructura que contiene el número de ciclo de simulación, el número de objetos que vemos en ese ciclo, y después un array de objetos con la información de cada uno:

`ObjectType type`	tipo (*OT_Unknown*, *OT_Ball*, *OT_Player*,*OT_Goal*, *OT_Flag*, *OT_Line*)
`Bool inViewConeP`	indica si se vé el objeto con la cámara o con el sensor de proximidad.
`double dist`	distancia del objeto.
`double dir`	dirección del objeto (ángulo con respecto al nuestro).
`double dist_chng`	cambio de la distancia en el tiempo.
`double dir_chng`	cambio de la dirección en el tiempo.
`double face_dir`	dirección hacia donde mira (solo jugadores).
`id`	especifica el objeto en particular que vemos.

La información de variable *id* dependerá del tipo de objeto. Para las porterías (*OT_Goal*) indica el bando, para jugadores (*OT_Player*) el equipo y el dorsal, y para banderas y líneas del campo, especifica cuál es de entre las existentes (fig. 2.2).

En nuestra implementación, esta función es la que nos proporciona toda la información perceptiva que necesitamos. Una vez la información está en estas estructuras, extraemos aquello que nos puede ser relevante para nuestros esquemas, guardándolo en las variables correspondientes.

estimate_current_pos Calcula una estimación de la posición y la dirección del jugador usando su información visual de los objetos fijos del terreno de juego (ver fig. 2.2).

```
Bool estimate_current_pos(SeeInfo *sinf, Side side, PosState
*pstate)
```

`sinf`	puntero a la estructura con la información visual.
`side`	bando del jugador.
`pstate`	puntero a la variable donde escribirá la posición y el ángulo.

Devuelve VERDADERO si ha sido capaz de calcular la posición del jugador usando la información disponible en la variable *sinf*, FALSO en caso contrario.

Además de las comunicaciones con el servidor y la traducción de órdenes e informacion sensorial, la librería nos proporciona esta función para poder localizarnos en el campo. Por desgracia, la estimación que nos proporciona no siempre es del todo fiable, pero por regla general resulta de utilidad.

A continuación se explican las funciones de la biblioteca *libsclient* que podemos usar para enviar los comandos motrices descritos en la sección 2.3.2.

send_com_move Envía la orden **Move** al sevidor.

```
Bool send_com_move(Socket *sock, double x, double y)
```

`sock`	*socket* de la conexión con el servidor
`x`	coordenada X del campo a donde queremos movernos.
`y`	coordenada Y del campo a donde queremos movernos.

send_com_turn Envía la orden **Turn** al sevidor.

```
Bool send_com_move(Socket *sock, double moment)
```

`sock`	*socket* de la conexión con el servidor
`moment`	angulo que queremos girar.

send_com_dash Envía la orden **Dash** al sevidor.

```
Bool send_com_dash(Socket *sock, double power)
```

`sock`	*socket* de la conexión con el servidor
`power`	fuerza con la que queremos acelerar el movimiento.

send_com_kick Envía la orden **Kick** al sevidor.

```
Bool send_com_kick(Socket *sock, double power, double dir)
```

`sock`	*socket* de la conexión con el servidor
`power`	fuerza con la que queremos golpear la pelota.
`dir`	dirección (con respecto a nosotros) hacia la que queremos golpear la pelota.

send_com_catch Envía la orden **Catch** al sevidor.

```
Bool send_com_catch(Socket *sock, double dir)
```

`sock`	*socket* de la conexión con el servidor
`dir`	dirección hacia la que queremos tratar de coger la pelota.

Capítulo 5

Experimentación

Se han realizado pruebas tanto a nivel de agentes individuales como de juego en conjunto. En el caso de las pruebas de los agentes, se han ido realizando a medida que se programaban y depuraban los distintos comportamientos implementados, para lo cual se ha programado un cliente especial permitido por el *SoccerServer*, como se explica a continuación. Utilizando dicho cliente, se han preparado escenarios de prueba donde se observa el comportamiento de agentes individuales ante distintas situaciones. Por último, y una vez completo el árbol de comportamientos, se ha podido probar el juego de un equipo formado por estos jugadores, contra equipos participantes en anteriores ediciones de la *RoboCup*.

5.1. El *Coach* o entrenador

El servidor del juego permite la existencia de clientes especiales, a los que se denomina *coach* o entrenador, que pueden realizar acciones especiales, no permitidas para los clientes jugadores normales, ya que sirven para manipular los distintos objetos y parámetros de la simulación sobre la marcha.

Hay dos tipos de *coach*. Uno de ellos puede estar presente en los partidos oficiales, haciendo las funciones de entrenador del equipo, y pudiendo únicamente observar el juego y dar instrucciones a los jugadores mediante mensajes de tipo auditivo. El otro tipo sólo se puede usar para ayudar en el desarrollo y prueba de los equipos, ya que puede realizar acciones como mover objetos a cualquier posición del campo, cambiar el modo de juego a voluntad, restaurar el cansancio de los jugadores, etc. Nosotros hemos programado un *coach* de este último tipo para ayudarnos durante el desarrollo.

Consta de un pequeño programa que conecta con el servidor y permite al usuario escribir órdenes, en el formato del servidor, que son enviadas a medida que se van escribiendo. Permite también almacenar conjuntos de órdenes en ficheros de

texto, las cuales pueden ser invocadas en cualquier momento para que se ejecuten secuencialmente. Usando esta característica es como se han implementado los escenarios de prueba que se detallan a continuación.

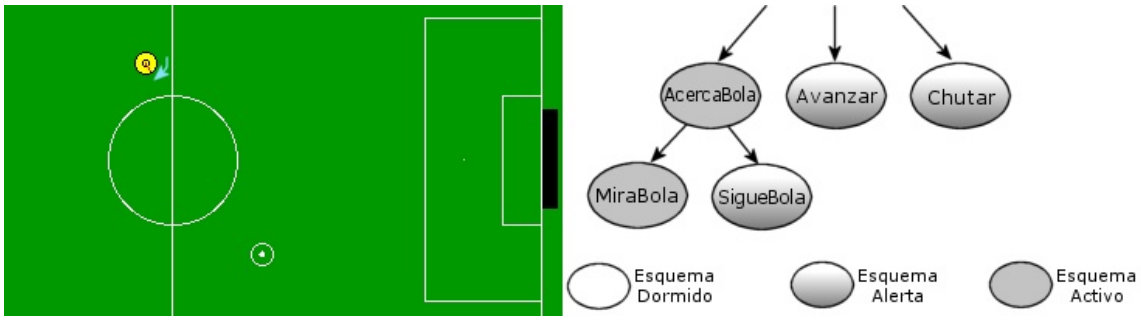
5.2. Escenarios de prueba

En esta sección se detallan los escenarios de prueba que se han utilizado durante el desarrollo y las pruebas finales. En ellos se prueba el comportamiento de un agente individual ante una situación específica del campo. En cada uno de ellos, se colocan la pelota y una serie de jugadores estáticos de uno u otro equipo en lugares determinados, se ubica al agente que queremos probar y se inicia el juego (modo `play_on`).

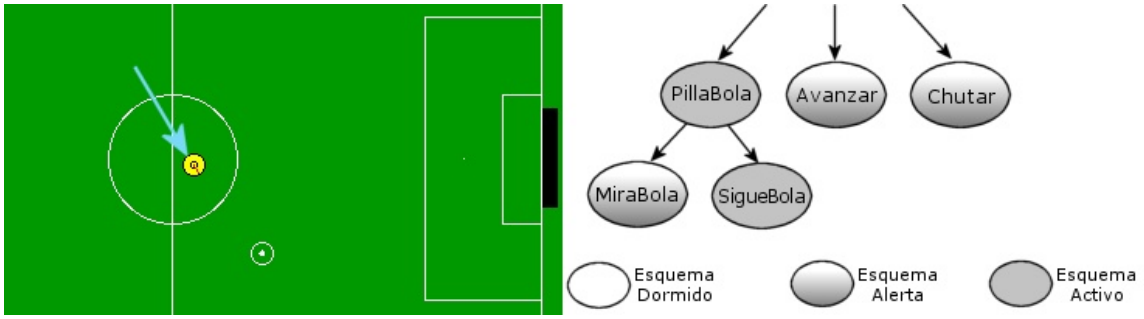
Para cada escenario, visualizaremos el comportamiento del agente paso por paso, mientras vamos mostramos el estado del árbol de esquemas JDE. Para mayor claridad, sólo se mostrarán los esquemas motrices, hijos de *Atacar*, que intervienen en cada escenario. Tampoco se muestra el estado de los esquemas perceptivos, ya que estos se encuentran siempre en ejecución.

5.2.1. Escenario *Marcar Gol*

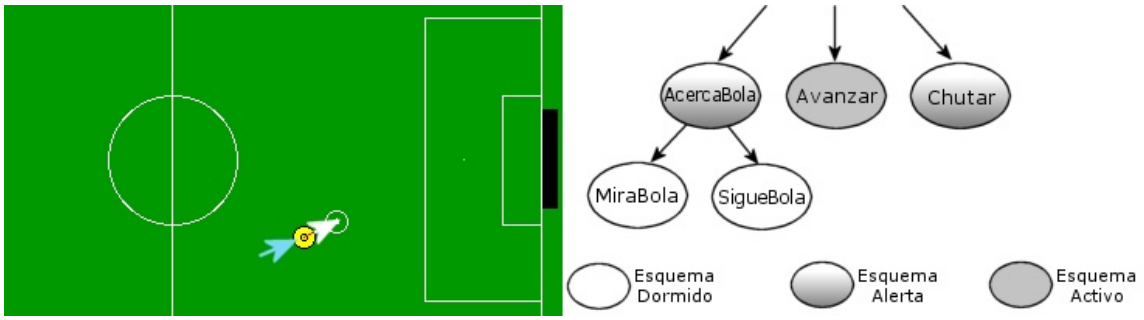
Un escenario sencillo donde probar los comportamientos de buscar la pelota, llevarla hasta la portería contraria y chutar a puerta. Están implicados, por tanto, los esquemas *AcercaBola*, *Avanzar* y *Chutar*. Los esquemas *Pasar* y *Driblar*, presentes también en este nivel del árbol, no se muestran al no intervenir para nada, encontrándose en estado `alerta` durante todo el escenario.



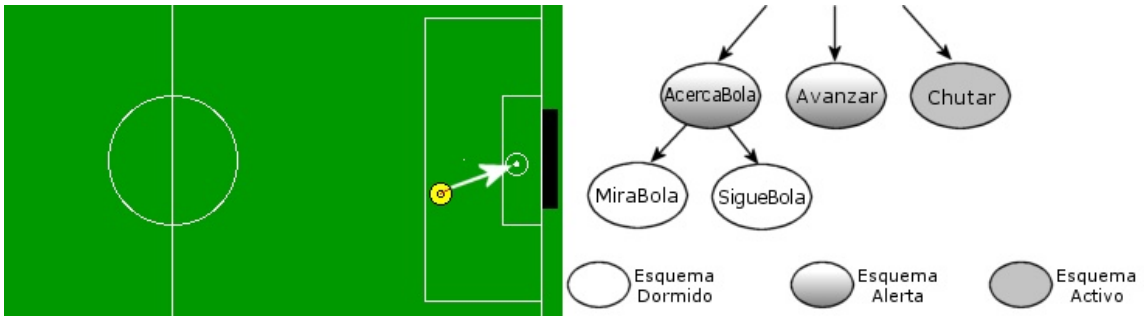
Al no tener la pelota de frente, el primer comportamiento que se activa es *MiraBola*, esquema hijo de *AcercaBola*, para buscarla.



Una vez el jugador tiene la pelota localizada, se activa el comportamiento *SigueBola*, hijo también de *AcercaBola*, para acercarse a ella



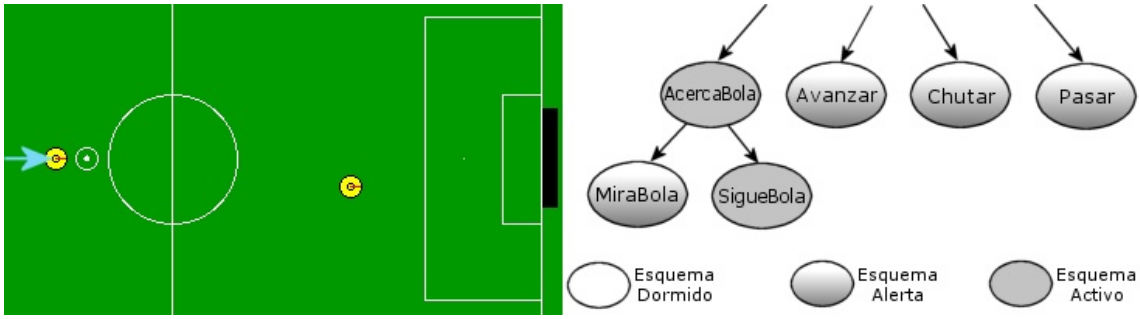
Cuando el jugador está a menos de 1 metro de la pelota, se activa *Avanzar*, para llevar la pelota a la portería contraria.



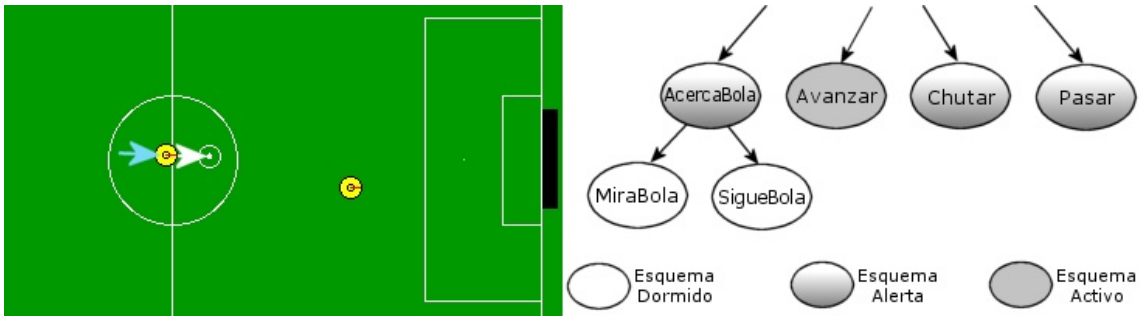
Por último, una vez el jugador está cerca de la portería contraria con la pelota, se activa *Chutar* para tratar de marcar gol.

5.2.2. Escenario *Pasar*

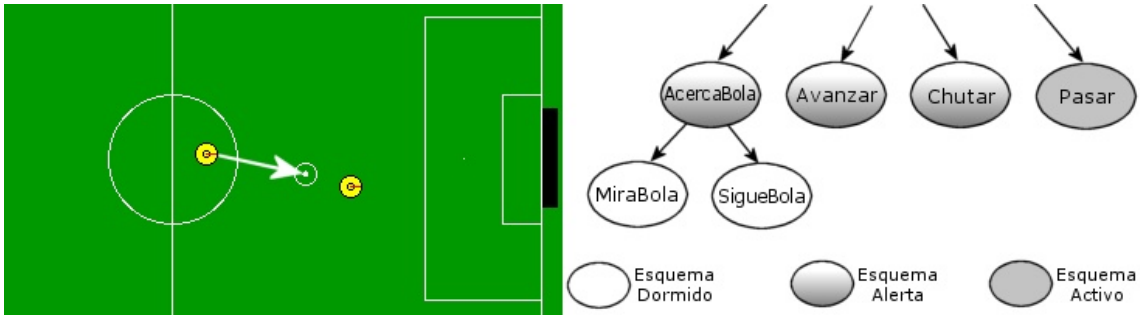
Se programa este segundo escenario para probar el esquema *Pasar*. Este esquema se activa cuando el agente que lleva la pelota detecta, de camino a la portería contraria, un jugador de su mismo equipo a cierta distancia por delante de él mismo. No se muestra el estado del esquema *Driblar*, que se encontrará constantemente en `alerta`, pues no interviene en este escenario.



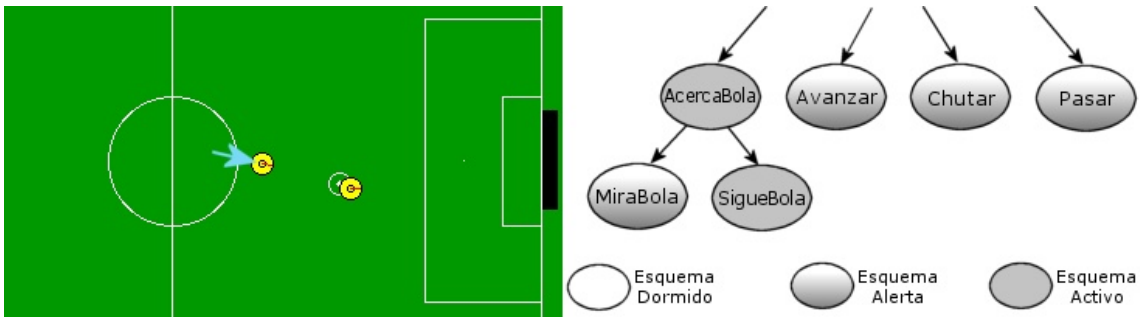
Comienza el escenario con la pelota enfrente del jugador, por lo que se activa *SigueBola*, y el jugador se dirige hacia ella.



Cuando el jugador llega hasta la pelota, comienza a *Avanzar* con ella hacia la portería contraria.



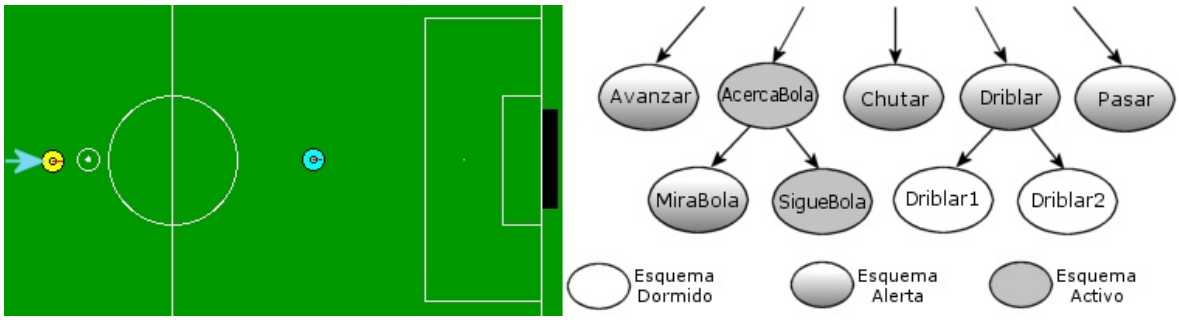
El avance se mantiene hasta que el jugador detecta que hay un compañero por delante suya, a menos de una determinada distancia y un determinado ángulo (ambos valores, parámetros de *Pasar*), momento en que pasa a activarse *Pasar*, y el jugador chuta hacia su compañero con una fuerza proporcional a la distancia a éste.



El agente sigue ejecutando su código, por lo que se vuelve a activar *AcercaBola* para llegar hasta la pelota, pero el escenario ya está completo: se ha realizado el pase con éxito.

5.2.3. Escenario *Driblar*

Aquí probaremos el comportamiento *Driblar*, que hace que el jugador trate de esquivar a los jugadores contrarios con los que se encuentre de camino a la portería.

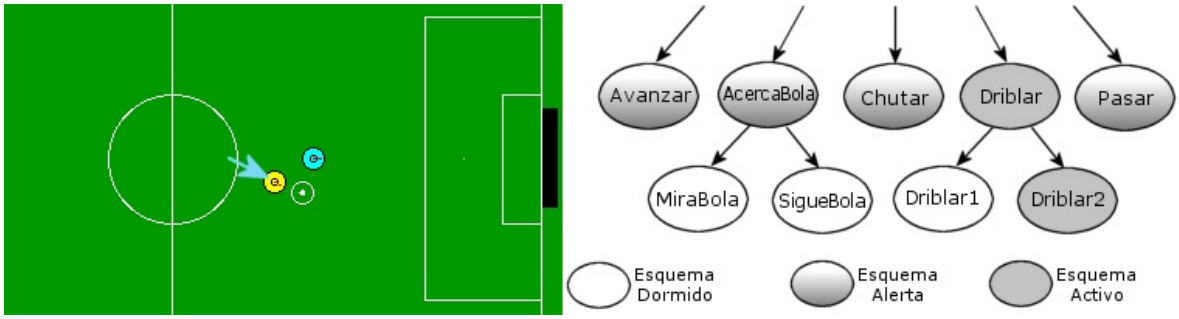


Como en el escenario anterior, comenzamos con la pelota enfrente del jugador, por lo que se activa *SigueBola*, y el jugador se dirige hacia ella.

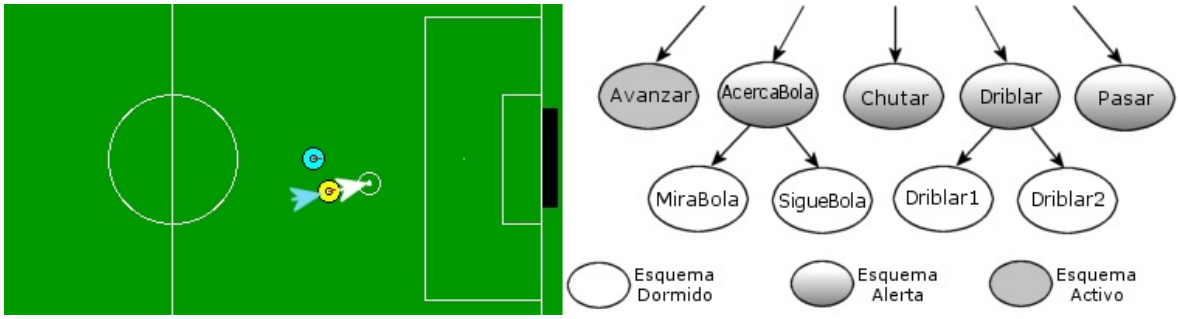
Cuando el jugador llega hasta la pelota, comienza a *Avanzar* con ella hacia la portería contraria.

El avance se mantiene hasta que el jugador del equipo contrario que hay por delante está a una cierta distancia (parámetro de *Driblar*) y se activa el esquema *Driblar*. A

diferencia de *AcercaBola*, *Driblar* no despierta a sus dos hijos para que compitan por el control, ya que lo que se quiere implementar es un comportamiento secuencial. Primero se activa a *Driblar1*, que chuta la pelota a cierto ángulo del jugador contrario y gira el cuerpo hacia el mismo sitio. Esta situación se mantendrá hasta que se tenga constancia de que la acción se ha ejecutado, es decir, hasta que se detecte que la pelota se ha alejado del jugador.



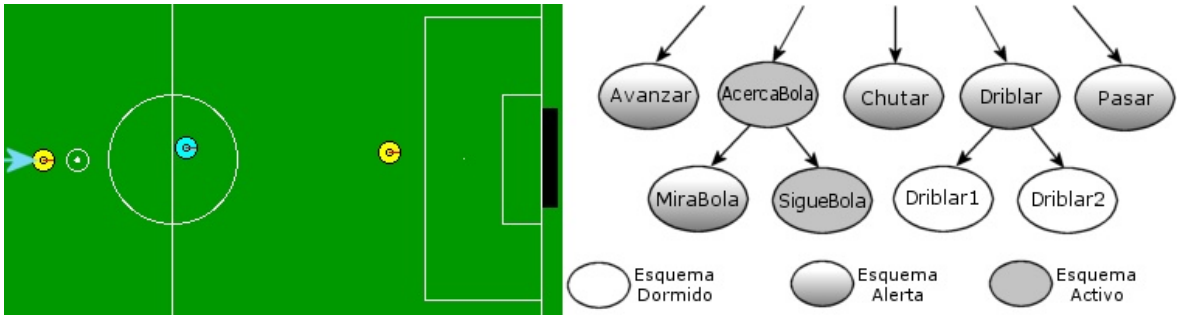
En este momento, el esquema padre *Driblar* duerme a *Driblar1* y pasa al siguiente paso: activar a *Driblar2*, que se encarga de comandar a los "motores" una velocidad muy alta (que, por su alto gasto de *Stamina*, no puede ser mantenida durante mucho tiempo) haciendo las necesarias correcciones de rumbo, hasta que la pelota vuelva a estar a menos de un metro de distancia.



Una vez el jugador tiene de nuevo la pelota, y no existiendo ya el estímulo del jugador contrario cerrándonos el paso, *Driblar* ya no necesita activarse, por lo que el control recae de nuevo en *Avanzar* y el jugador prosigue su camino hacia la portería contraria.

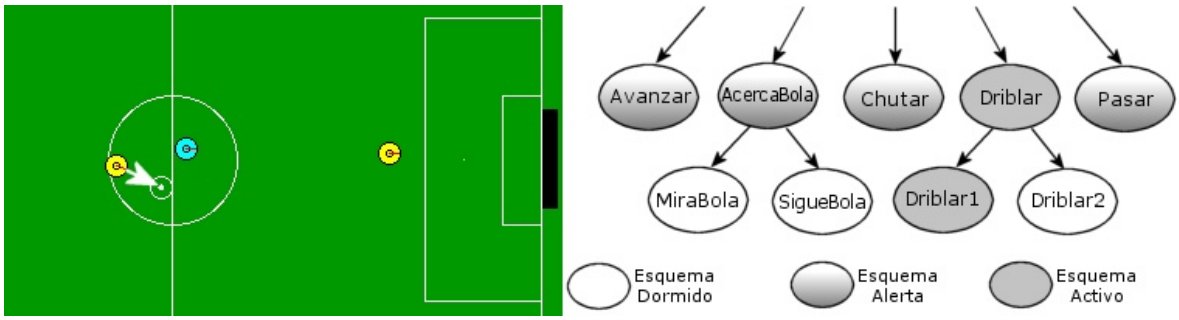
5.2.4. Escenario *Driblar y Pasar*

Por último, se prueba este escenario que combina los dos anteriores, donde el agente ha de esquivar a un jugador contrario para después pasar la pelota a su compañero de equipo. Se prueba de esta forma la capacidad de la jerarquía de esquemas para reaccionar correctamente ante distintos estímulos, mostrando un comportamiento inteligente.



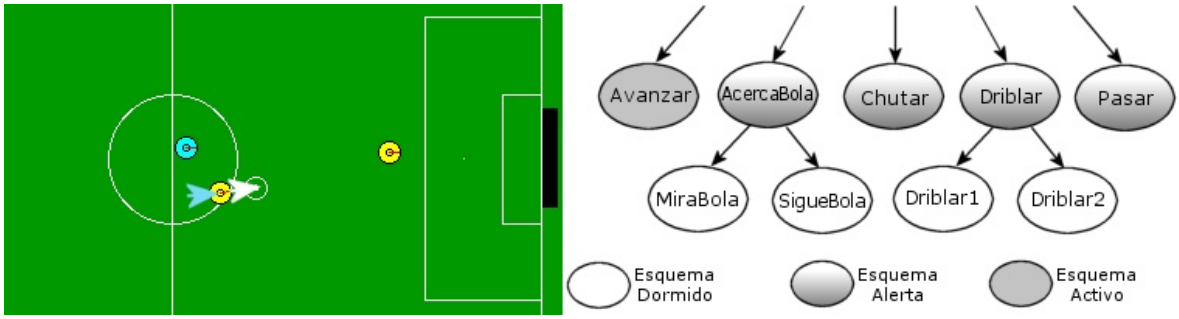
Como en el escenario anterior, comenzamos con la pelota enfrente del jugador, por lo que se activa *SigueBola*, y el jugador se dirige hacia ella.

Cuando el jugador llega hasta la pelota, comienza a *Avanzar* con ella hacia la portería contraria.

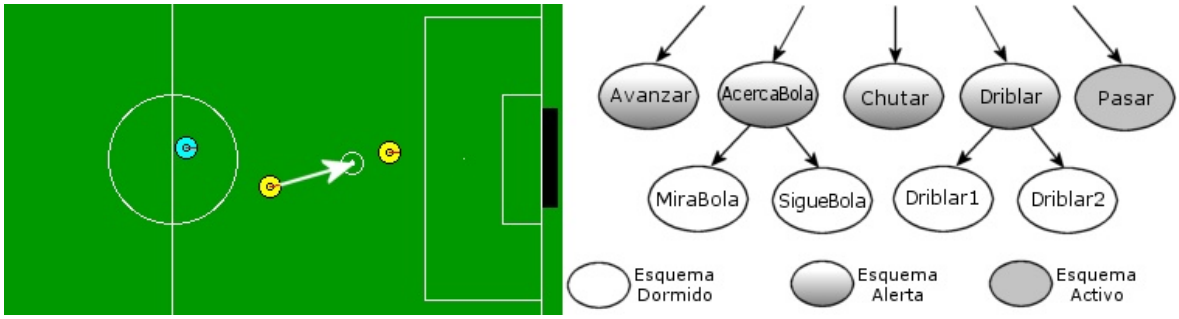


De nuevo, al ser detectado el jugador contrario bloqueando el camino a la portería, se activa el esquema *Driblar*, en su primer paso: esquema hijo *Driblar1*.

Acto seguido se activa *Driblar2* para recuperar la pelota.



Y con la pelota "a sus pies", el jugador prosige, avanzando hacia la portería.



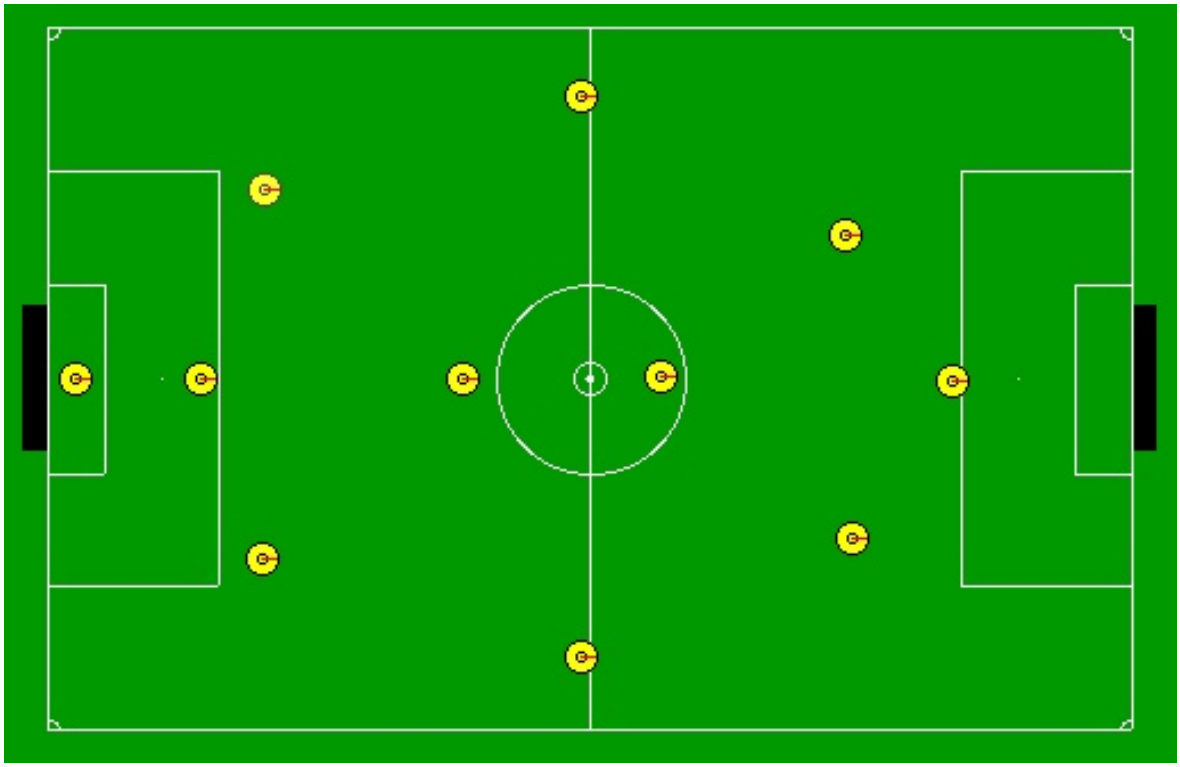
Aunque, como detecta que hay un compañero por delante suya, decide pasarle la pelota, activándose el esquema *Pasar*.

Cabe aclarar que en realidad la transición entre estados no es tan sencilla como se muestra en estas figuras, en las que se ha simplificado el proceso. En efecto, el efecto de avanzar se consigue alternándose la activación de los esquema *Avanzar* y *AcercaBola*, y la activación de este último se produce en múltiples momentos durante el escenario, en aquellas ocasiones en que la dirección de pelota se aleja del centro del cono de visión del jugador o se la pierde de vista.

5.3. Partidos

Tras las pruebas individuales de los agentes, y tras la inclusión de las características de parametrización de los jugadores descritas en la sección 4.2.2, se realizaron pruebas de juego en conjunto, formando equipos de once procesos jugadores.

Debido a los pases entre jugadores, el equipo es capaz de avanzar de forma eficiente con la pelota y marcar gol con ella. Sin embargo, el rendimiento es inferior al de los equipos contra los que se le ha enfrentado, cuyos jugadores realizan predicciones de trayectorias o tienen una mayor "conciencia" de juego en conjunto.



Donde mayores carencias se observan es a la hora de recuperar la pelota, una vez la ha tomado el equipo contrario, lo que es especialmente evidente en la actuación de los defensas y el portero. Esto no es sorprendente, puesto que el árbol de esquemas que se ha diseñado está orientado casi exclusivamente a atacar.

Capítulo 6

Conclusiones

Finalizamos esta memoria analizando los resultados obtenidos con la realización del presente proyecto. Haremos un repaso a continuación de los objetivos que se han cumplido y, para terminar, veremos qué posibles mejoras se podrían aplicar para conseguir un comportamiento más eficiente.

6.1. Consecución de objetivos

El objetivo inicial del proyecto era la consecución satisfactoria de un comportamiento complejo, utilizando la arquitectura JDE, en el entorno de la liga simulada de la *RoboCup*. Como se ha podido comprobar en las pruebas del capítulo 5 este objetivo se ha cumplido.

La consecución del objetivo se ha producido en tres fases: un entorno de actuación inicial, construcción del comportamiento de jugador de fútbol y la formación de un equipo de procesos jugadores.

Entorno de actuación inicial

Primero, y apoyándose en la biblioteca de comunicaciones *libsclient*, explicada en la sección 4.3.3, se programaron los esquemas *Sensors* y *Motors* que nos proporcionan las bases sensorial y motriz respectivamente, que luego son usadas por el resto de los esquemas de la jerarquía.

Así mismo, para depurar y probar estos dos esquemas de servicio, se programó un esquema de prueba, no comentado en el apartado de la implementación, al que se denominó *Teclado*. Este era un esquema motriz pensado para que un usuario manejara a un jugador mediante el teclado. Gracias al mismo se probó la validez inicial de los esquemas *Sensors* y *Motors*.

Comportamiento "jugador de fútbol"

Teniendo una primera base sensorial y motriz válidas, se comenzó la programación de los demás esquemas. A medida que se iban añadiendo esquemas a la jerarquía, nos encontramos con los problemas de control explicados en la sección 4.3.2.

La depuración del esquema *Motors*, añadiendo el control en velocidad y el control del ritmo de envío de órdenes al servidor, ha sido una constante durante la programación y prueba de la jerarquía de comportamientos.

Formación del equipo

Una vez completo el árbol que implementa el comportamiento de nuestro jugador de fútbol virtual, el siguiente paso era formar un equipo de jugadores, utilizando para ello once procesos de este tipo. Para que los procesos jugadores se comportaran como un equipo, se optó por la solución de las zonas de referencia, tal y como se explica en la sección 4.2.2. Para implementar esto, se programó un nuevo esquema, al que se llamó *Volver*, para hacer que los jugadores regresaran a su zona según fuera necesario, conservando así sus posiciones en el campo.

Esta última fase ha sido quizá la menos satisfactoria ya que, aunque se puede observar un comportamiento emergente de juego en equipo (debido a los pases entre jugadores), los bajos resultados obtenidos al jugar contra otros equipos nos indica que aún quedaría mucho por afinar en el comportamiento de los jugadores.

6.2. Conclusión

Es obvio, dados los resultados obtenidos, que la arquitectura JDE ha resultado válida para ser utilizada en este entorno.

Gracias a la facilidad para integrar nuevos comportamientos en la jerarquía, ha sido posible programar el comportamiento complejo de un jugador de fútbol de una forma incremental, ayudando esto considerablemente en el desarrollo. Así mismo, la separación existente entre esquemas facilita la depuración y prueba de comportamientos individuales.

Ha habido también algún inconveniente en el uso de JDE, siendo el más importante la difícil integración del modo de generar salidas de movimiento por parte de los esquemas, con la forma de funcionar de los actuadores. Sin embargo, una vez resuelto este problema de la forma descrita en el apartado 4.3.2, la creación de nuevos comportamientos es una tarea relativamente sencilla, permitiendo utilizar con gran comodidad la arquitectura JDE dentro de este entorno.

6.3. Mejoras y trabajos futuros

Una de las mejoras más importantes sería la formación de jugadores heterogéneos. El comportamiento de jugador de fútbol programado es bastante conveniente para el rol de delantero y quizá de medio campo, pero es relativamente deficiente para un defensa y, en el caso del portero, absolutamente inadecuado. La programación de otro árbol de comportamientos distinto para el rol de portero (aprovechando gran parte de los esquemas ya programados) es una línea inmediata para continuar con el trabajo realizado en este proyecto fin de carrera.

Por otra parte, la eficiencia del juego podría mejorarse aún mucho, creando nuevos esquemas de comportamientos (saques de banda, buscar el pase, etc...) así como la inclusión de alguna forma de predicción de la trayectoria de la pelota que mejorara el juego en el aspecto defensivo.

En cualquier caso, utilizando la base perceptivo/sensorial que se ha programado, puede implementarse cualquier nuevo comportamiento que se nos ocurra utilizando JDE (o cualquier otra arquitectura), en el entorno de la liga simulada de la *RoboCup*.

Bibliografía

[1] Capek, Karel: *"R.U.R. Rossom's Universal Robots"*, 1921.

[2] Varios autores: *"RoboCup Soccer Server, Users Manual"*, 2002

[3] Cañas Plaza, JoseMaría: *"Dynamic schema hierarchies for an autonomous robot"*, Iberamia 2002.

[4] Arbib, M., Liaw, J.S.: *"Sensorimotor transformations in the worlds of frogs and robots"*. Artificial Intelligence, 72 (1995)

[5] Arkin, R.: *"Motor Schema-Based Mobile Robot Navigation"*. The International Journal of Robotics Research, 8(4) (1989)

[6] RoboCup federation: http://www.robocup.org

[7] Álvarez Rey, Jose María.: *"Implementación basada en lógica borrosa de jugadores para la Robocup"*, Proyecto Fin de Carrera, Universidad Rey Juan Carlos, 2001.

[8] Grupo de Sistemas y Comunicaciones Universidad Rey Juan Carlos, *"Robótica"*, http://gsyc.escet.urjc.es/docencia/asignaturas/robotica, 2003.

[9] Saffiotti, Alessandro y Wasik, Zbigniew: *"Using Hierarchical Fuzzi Behaviors in the RoboCup domain"*

[10] STRELB, M. D., TURNER, M.: *"Linux"*, Prentice Hall, 2000.

Anexo

Ejemplo de código fuente de un esquema: Chutar a puerta

```
#include "jde.h"

#include <math.h>

int rc_chutaPuerta_cycle=150; /* ms */

arbitration rc_chutaPuerta_callforarbitration;
int rc_chutaPuerta_brothers[NUM_SCHEMAS];

/* Schema Parameters, default values */
float rc_chutaPuerta_kick = 80;
float rc_chutaPuerta_limite = 12;

/* kick! */
void rc_chutaPuerta_iteration()
{
  if (debug[SCH_RC_CHUTAPUERTA])
    printf("rc_chutaPuerta: Chutando (%.2f)\n", rc_puertaDir);

  rc_v = 0;                         /* Stop the motors */
  rc_kick_dir = rc_puertaDir;       /* Direction: goal */
  rc_kick = rc_chutaPuerta_kick;    /* Power of the kick */
}

/* we must have the ball and be close enough to the goal */
float rc_chutaPuerta_preconditions()
{
  return ( rc_veoBola &&
           rc_bolaDist <= 1.0 &&
           rc_puertaDist <= rc_chutaPuerta_limite );
}

void rc_chutaPuerta_suspend()
{
  pthread_mutex_lock(&mymutex[SCH_RC_CHUTAPUERTA]);
  state[SCH_RC_CHUTAPUERTA]=slept;
```

```
  pthread_mutex_unlock(&mymutex[SCH_RC_CHUTAPUERTA]);
}

void rc_chutaPuerta_resume(int *brothers, arbitration fn)
{
  int i;

  pthread_mutex_lock(&mymutex[SCH_RC_CHUTAPUERTA]);
  for(i=0;i<NUM_SCHEMAS;i++) rc_chutaPuerta_brothers[i]=-1;
  i=0;
  while(brothers[i]!=-1) {rc_chutaPuerta_brothers[i]=brothers[i]; i++;}
  rc_chutaPuerta_callforarbitration=fn;

  put_state(SCH_RC_CHUTAPUERTA,notready);
  pthread_cond_signal(&condition[SCH_RC_CHUTAPUERTA]);
  pthread_mutex_unlock(&mymutex[SCH_RC_CHUTAPUERTA]);
}


void *rc_chutaPuerta_thread(void *not_used)
{
  struct timeval a,b;
  long diff, next;
  int i,other=0, forcedbrother=-1;

  for(;;)
    {
      pthread_mutex_lock(&mymutex[SCH_RC_CHUTAPUERTA]);
      if (state[SCH_RC_CHUTAPUERTA]==slept)
{
  printf("rc_chutaPuerta off\n");

  pthread_cond_wait(&condition[SCH_RC_CHUTAPUERTA],&mymutex[SCH_RC_CHUTAPUERTA]);
  printf("rc_chutaPuerta on\n");
}
      else
{
  gettimeofday(&a,NULL);
  if (rc_chutaPuerta_preconditions()==0.)
    {  /* my preconditions DON'T match current situation */
      i=0; other=0;
      while(rc_chutaPuerta_brothers[i]!=-1)
{
  if ((state[rc_chutaPuerta_brothers[i]]==ready)
      || (state[rc_chutaPuerta_brothers[i]]==winner)) other++;
  i++;
}

      if ((other>1)||
  ((other==1)&&(state[SCH_RC_CHUTAPUERTA]==notready)) ||
```

```
  ((other==1)&&(state[SCH_RC_CHUTAPUERTA]==forced)))
{
  /* preconditions of another brother do match */
  put_state(SCH_RC_CHUTAPUERTA,notready);
}
      else
{ /* there is control absence */
  if ((state[SCH_RC_CHUTAPUERTA]!=forced)&&(state[SCH_RC_CHUTAPUERTA]!=notready))
    put_state(SCH_RC_CHUTAPUERTA,notready);
  rc_chutaPuerta_callforarbitration();
  if (state[SCH_RC_CHUTAPUERTA]==forced) rc_chutaPuerta_iteration();
}
    }
  else /* my preconditions match current situation */
    {
      i=0; other=0;
      while(rc_chutaPuerta_brothers[i]!=-1)
{
  if ((state[rc_chutaPuerta_brothers[i]]==ready)
      || (state[rc_chutaPuerta_brothers[i]]==winner)) other++;
  if (state[rc_chutaPuerta_brothers[i]]==forced) forcedbrother=i;
  i++;
}

      if ((other==0)||
  ((other==1)&&(state[SCH_RC_CHUTAPUERTA]==winner)) ||
  ((other==1)&&(state[SCH_RC_CHUTAPUERTA]==ready)))
{
  if (forcedbrother!=-1) put_state(forcedbrother,notready);
  if (state[SCH_RC_CHUTAPUERTA]!=winner) put_state(SCH_RC_CHUTAPUERTA,winner);
  rc_chutaPuerta_iteration();
}

      else { /* there is control overlap */
if ((state[SCH_RC_CHUTAPUERTA]!=winner)&&(state[SCH_RC_CHUTAPUERTA]!=ready))
    put_state(SCH_RC_CHUTAPUERTA,ready);
rc_chutaPuerta_callforarbitration();
if (state[SCH_RC_CHUTAPUERTA]==winner) rc_chutaPuerta_iteration();
      }

    }

  gettimeofday(&b,NULL);
  diff = (b.tv_sec-a.tv_sec)*1000000+b.tv_usec-a.tv_usec;

  next = rc_chutaPuerta_cycle*1000-diff-10000;
  /* discounts 10ms taken by calling usleep itself */
  if (next>0) usleep(rc_chutaPuerta_cycle*1000-diff);
  else {printf("time interval violated: rc_chutaPuerta\n");
  usleep(rc_chutaPuerta_cycle*1000);}
```

```
}
      pthread_mutex_unlock(&mymutex[SCH_RC_CHUTAPUERTA]);
    }
}


void rc_chutaPuerta_startup()
{
  pthread_mutex_lock(&mymutex[SCH_RC_CHUTAPUERTA]);
  printf("rc_chutaPuerta schema started up\n");
  put_state(SCH_RC_CHUTAPUERTA,slept);
  pthread_create(&schema[SCH_RC_CHUTAPUERTA],NULL,rc_chutaPuerta_thread,NULL);
  pthread_mutex_unlock(&mymutex[SCH_RC_CHUTAPUERTA]);
}
```