UNIVERSIDAD REY JUAN CARLOS

Ingeniería Informática

Escuela Superior de Ciencias Experimentales y Tecnología

Curso académico 2004-2005

Proyecto Fin de Carrera

jde+
Una plataforma de desarrollo para aplicaciones robóticas

Tutor: José María Cañas Plaza

Autor: David Lobato Bravo

8 de septiembre de 2005

A mis padres, por su paciencia

Resumen

El tema de este proyecto, es desarrollar el sistema *jde+*, una plataforma para la creación de aplicaciones robóticas, que se apoya en la arquitectura teórica de JDE [24]. Esta arquitectura ha sido desarrollada por el grupo de robótica de la URJC y plantea las aplicaciones robóticas como una colección jerárquica de esquemas concurrentes.

El sistema *jde+* pretende ser una herramienta para el desarrollo de aplicaciones robóticas. Su objetivo principal es facilitar al máximo la tarea de desarrollo de aplicaciones robóticas. Para ello el sistema *jde+* actuará como un *middleware*, entre la aplicación y el robot, aportando los mecanismos básicos y haciendo que el programador sólo tenga que preocuparse de la lógica de su aplicación.

Agradecimientos

En primer lugar a Tamara, por el apoyo moral en los momentos bajos. A mi familia, y especialmente a mis padres por su comprensión y paciencia. Y por último, a José María Cañas por la ayuda prestada aun desde la distancia.

Índice general

Índice de figuras

Índice de cuadros

Capítulo 1

Introducción

1.1 Un poco de Historia

La palabra *robot* proviene del término checo *robota*, cuyo significado es "servidumbre, trabajo forzado o esclavitud". Principalmente usado para referirse a los siervos Checoslovacos de la época feudal, que eran obligados a trabajar dos o tres días a la semana en las tierras de los nobles sin ningún tipo de retribución. Pasado el sistema feudal, siguió usándose el término, para referirse al trabajo que no era realizado de manera voluntaria. En la actualidad, el término es usado por la juventud Checa y Eslovaca para referirse al trabajo aburrido o poco interesante. Según la Real Academia Española, el término robot se refiere a:

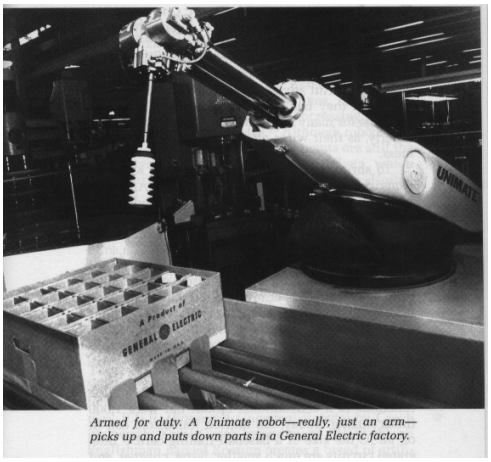
> *Máquina o ingenio electrónico programable, capaz de manipular objetos y realizar operaciones antes reservadas sólo a las personas*

Fue Karel Capek, quien utilizó por primera vez el término *robot* en su obra teatral *Rossum's Universal Robots*[11]. La obra, estrenada en 1921, narra la historia de Rossum, un científico, que consigue crear *robots* capaces de servir a la clase humana. Tras su perfeccionamiento, los *robots* se rebelan contra sus amos, destruyendo toda vida humana. El término, continúa usándose en varios escritos posteriores pertenecientes al género literario de la ciencia-ficción, pero no fue hasta años más tarde, en 1942, cuando Isaac Asimov utilizó por primera vez el termino robótica para referirse a la tecnología de los robots, en la novela *Runaround*.

Como precursores históricos de los robots actuales, tenemos diversos autómatas, diseñados y construidos por grandes mentes muy avanzadas a su época. Como ejemplos citar el Gallo de la catedral de Estrasburgo de autor desconocido en 1352, el guerrero mecánico de Leonardo Da Vinci en 1495, varios muñecos animados de Jacques Vaucanson en 1738, autor también del primer telar mecánico, o la muñeca dibujante de Henry Maillardet en 1805.

La robótica como hoy la conocemos, surge hacia la década de los 50, con la construcción de las primeras computadoras. Aunque no será hasta la década de los 70 con los primeros microprocesadores cuando comience su vertiginoso avance. Las primeras patentes aparecen en 1946, describiendo primitivos robots para el traslado de maquinaria. Será en 1954, cuando George Devol, diseñe el primer mecanismo programable aplicado a la industria, el *Universal Automation*, más tarde reducido a Unimation y que sería el corazón del primer brazo robótico industrial (figura 1.1). Junto con Joseph F. Engelberger, G. Devol formará la primera compañía comercial dedicada a la robótica, Unimation Inc., dedicada a robots industriales para cadenas de montaje.

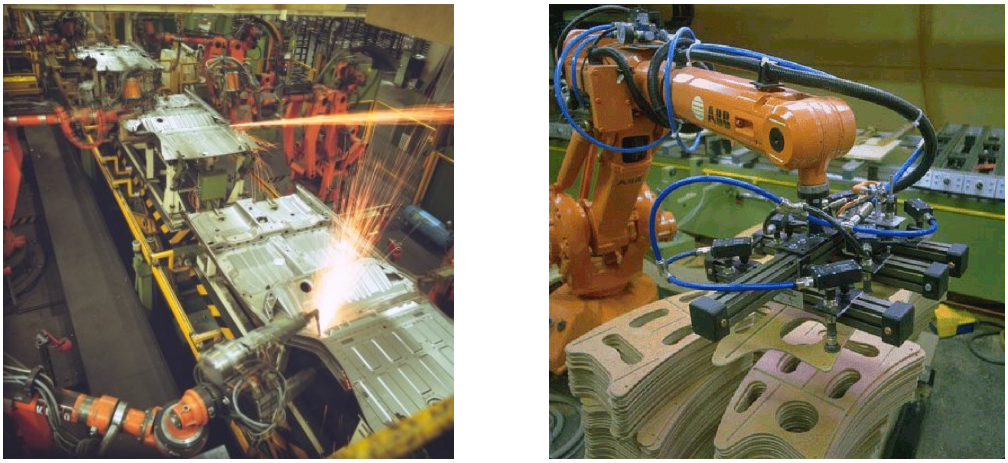
Un hito paradigmático en este escenario son los brazos articulados tipo PUMA (Programmable Universal Manipulator for Assembly). Se utilizan con frecuencia para soldar, pintar, transporte de materiales, etc. tal como ilustra la figura 1.2. Por ejemplo



Armed for duty. A Unimate robot—really, just an arm—picks up and puts down parts in a General Electric factory.

Figura 1.1: *Unimate: Primer brazo robótico industrial.*

los puntos de soldadura se le dan predefinidos en una lista con la que se programa el recorrido del brazo. Las principales ventajas que ofrece un brazo robotizado frente a un operario humano para estas labores repetitivas son la precisión y la rapidez. Desde el punto de vista del empresario también hay que considerar que los robots no hacen huelga, no se aburren y pueden trabajar 24 horas al día, todos los días del año. Además salvando cierta inversión inicial, su coste de mantenimiento puede ser menor que el equivalente para que la misma labor la realicen operadores humanos.



(a) (b)

Figura 1.2: *Robots soldadores en una factoría de coches (a) y brazo transportando materiales (b)*

Los robots también aparecen en otros escenarios fuera de la fábrica. En entornos altamente peligrosos para las personas se utilizan robots para reemplazarlas. Por ejemplo, se utilizan robots teleoperados para explorar zonas de alta radiación en el interior de reactores nucleares, para rastrear en las profundidades oceánicas, para sondear volcanes o pirámides, para inspeccionar y desactivar bombas o minas. En estos casos el robot suele estar equipado con cámaras de vídeo y es guiado remotamente por un operador humano.

En los últimos años está creciendo el uso de robots para tareas agrícolas [30]. Las tareas de fumigación o cosecha son potencialmente automatizables, porque son repetitivas y tediosas, e incluso conllevan cierto riesgo químico para el operario humano. Por ejemplo, hay robots recolectores de tomates, melones, pepinos o champiñones, que utilizan visión artificial para identificar el fruto adecuado [15]. Otra aplicación relevante en esta línea son los cosechadores automáticos de cereales, que recorren de modo

semiautónomo los inmensos campos de cereales segando y recogiendo el grano.

Las exploraciones espaciales suponen otro campo de aplicación de la robótica. Se han utilizado robots en varias misiones. Por ejemplo, la misión Pathfinder puso al robot Sojourner en la superficie de Marte en 1997. El robot tenía 6 ruedas y básicamente se movía teleoperado desde la Tierra. Tomó varias imágenes, analizó el suelo y algunas rocas del planeta rojo, enviando los resultados a la Tierra. En esta línea la NASA ha invertido y sigue invirtiendo mucho en proyectos que desarrollan robots capaces de desenvolverse en entorno hostil tan lejano (como el programa de telerobotica espacial[1]).

Uno de los campos insospechados por donde está creciendo la oferta robótica es el ocio y el entretenimiento. Por ejemplo, el perrito Aibo de Sony[2] se vende como mascota y los robots humanoides desarrollados por Honda y Sony en los últimos años avanzan en esta línea. También el ladrillo de Lego[3] se vende como juguete imaginativo. En este sentido hay que destacar también la RoboCup[4], que es una competición internacional anual en la que equipos de robots juegan al fútbol en distintas categorías.

También hay tímidos intentos de introducir los robots en el hogar para automatizar tareas domésticas. Una muestra es el aspirador robótico *Roomba*[5] que deambula por una habitación de forma autónoma sin chocar con las paredes, tragando pelusas y deslizándose debajo de las camas y los sofás. Sus sensores evitan que choque contra las paredes o los muebles, o se caiga por el hueco de las escaleras.

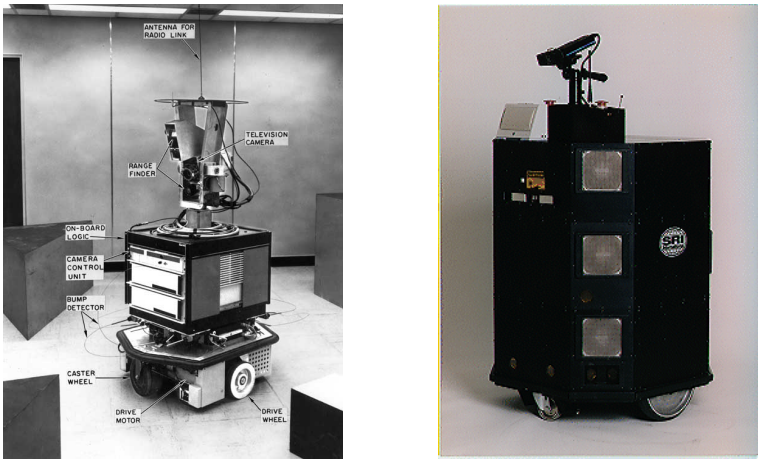
En general, el uso de los robots va creciendo paulatinamente, aumentando el rango de sus aplicaciones a medida que éstos aumentan en autonomía y funcionalidad. Precisamente la falta de autonomía es el principal cuello de botella que ha impedido el uso generalizado de robots. En la mayoría de las aplicaciones robóticas de hoy día los robots utilizados no tienen autonomía, son teleoperados o autómatas que ejecutan una sola tarea. Aunque algunos escenarios admiten teleoperación, en ciertas aplicaciones la autonomía es un requisito ineludible y en la mayoría, una característica muy ventajosa. Por ejemplo, en robots para exploraciones espaciales la teleoperación se hace demasiado lenta cuando se alejan de la Tierra y las ondas de radio tardan demasiado en llegar. En los robots de entretenimiento la gracia radica en que sean mascotas independientes, con su propias capacidades, y no meras marionetas pasivas. En las tareas agrícolas la ventaja principal se consigue al automatizar de modo completo el laboreo. Con ello se reduce al mínimo la intervención humana, quizás a una sencilla supervisión. Del mismo modo sucede con las tareas domésticas.

El camino hacia los robots móviles actuales ha sido largo. Los primeros proyectos en este área datan de principios de los años 70, cuando por primera vez se reunen en una misma plataforma sensores, motores y procesadores [27]. Desde entonces se ha ido consolidando como un campo específico dentro de la robótica, en el cual el comportamiento principal es el movimiento y donde hay una especial preocupación por la autonomía. Además la evolución de este campo ha ido definiendo sus propios problemas como la navegación, la construcción de mapas, la localización, etc.

El robot Shakey [22] (figura 1.3(a)) es un pionero dentro la robótica móvil, y fue construido en 1970 en el Stanford Research Institute (SRI). Tenía una cámara, motores y sensores odométricos. Era capaz de localizar un bloque en su entorno y empujarlo lentamente. El procesamiento de las imágenes se realizaba en un ordenador fuera del robot. Dentro del mismo instituto, un digno sucesor fue el robot Flakey (figura 1.3(b)),

[1] http://ranier.oact.hq.nasa.gov/telerobotics_page/telerobotics.shtm
[2] http://www.aibo.com
[3] http://www.legomindstorms.com
[4] http://www.robocup.org
[5] http://www.irobot.com

construido en 1984 con numerosos avances tecnológicos sobre Shakey, y que ya incluía en su interior ordenadores para realizar a bordo cualquier cómputo necesario para su comportamiento.



(a) (b)

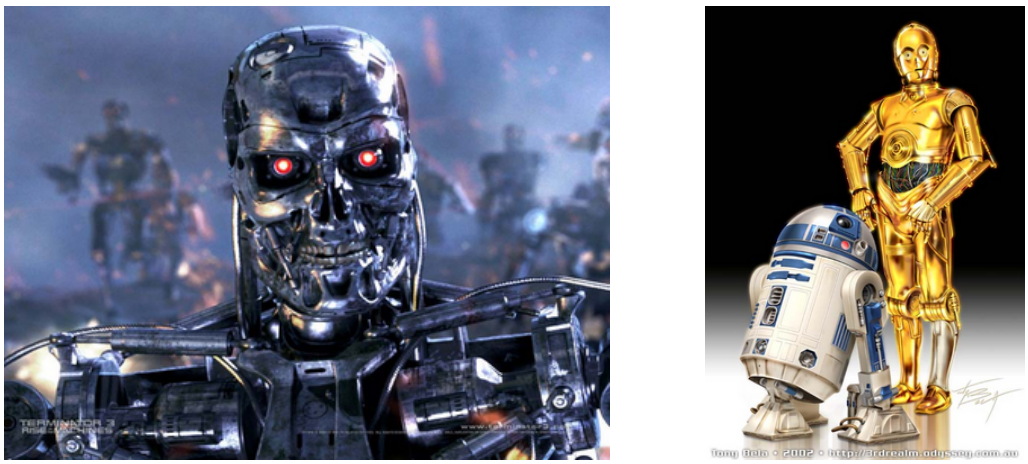
Figura 1.3: *Robots Shakey (a) y Flakey (b)*

Actualmente no es ciencia ficción ver a un robot móvil guiando a los visitantes por un museo[6], navegando por entornos de oficinas[7], o conduciendo un coche de un extremo a otro de Estados Unidos (NavLab-CMU[8]). Estos prototipos son el fruto de la investigación de varias universidades y centros situados a la vanguardia de la robótica, como Carnegie Mellon University, el Massachusetts Institute of Technology o el Stanford Research Institute.

Aunque se ha progresado mucho, aún estamos lejos de las expectativas levantadas por el cine o la literatura. Por ejemplo, robots móviles tan versátiles como C3PO y R2D2 de la Guerra de las galaxias, o los que aparecen en Blade runner o Terminator siguen siendo a día de hoy ciencia ficción. De hecho estamos muy lejos que conseguir una máquina que pase con buena nota el test de Turing [31]. En este terreno la imaginación ha ido muy por delante de la realidad y ha levantado unas expectativas demasiado prometedoras. Expectativas que no se han satisfecho, lo que provoca cierta frustración y descrédito. En este sentido las comunidades robótica y de inteligencia artificial se han encontrado con unas barreras muy complejas y difíciles de superar: el lenguaje natural, el aprendizaje, autonomía, adaptación, etc. La construcción de máquinas que se comporten de modo autónomo sigue siendo un reto y un problema abierto.

En el terreno material, el crecimiento de la robótica móvil se ha beneficiado enormemente de la revolución electrónica que hemos experimentado en las últimas décadas. Las mejoras tecnológicas y en microelectrónica han disparado la capacidad de cómputo disponible y el desarrollo de los más variados sensores, motores y dispositivos. Por ejemplo, sensores precisos como el láser, el GPS, o las cámaras en color, y motores de continua, servos, etc. aparecen frecuentemente como parte del equipamiento de los robots móviles. La velocidad de los computadores se ha multiplicado en los últimos años considerablemente y esto ha hecho factible la materialización de algunas aproximaciones a la robótica móvil que demandan gran capacidad de cálculo. Sin embargo, a pesar de estas mejoras tecnológicas, de disponer de los procesadores más rápidos, los sensores y actuadores más avanzados, el cómo combinarlos para generar comportamiento autónomo sigue siendo un problema abierto. Los progresos en el hardware

[6]Minerva: http://www.cs.cmu.edu/~minerva
[7]Xavier: http://www.cs.cmu.edu/~xavier
[8]http://www.ri.cmu.edu/labs/lab 28.html



(a) (b)

Figura 1.4: *Robótica en la ciencia-ficción: Terminator (a), androides C3PO y R2D2 (b)*

han puesto en evidencia la importancia de una buena organización interna, que se ha convertido en un factor crítico para conseguir el comportamiento autónomo robusto.

En este aspecto se ha trabajado largo y tendido desde un punto de vista teórico, describiendo como se organizan el sensado del entorno y la actuación de un robot, en busca de un comportamiento observable. Esta organización teórica, que se denomina arquitectura de control robótico, o simplemente *arquitectura*. Su aplicación práctica conduce a un problema de diseño software, que desemboca en un sistema software real. Dicho sistema permite el desarrollo de aplicaciones robóticas fundamentadas en una teoría de organización. Al diseño software de dichos sistema lo denominaremos arquitectura software, y al los sistema reales los denominaremos plataformas de desarrollo o *frameworks*.

El tema tratado en el presente proyecto, es precisamente la construcción de una plataforma de desarrollo para aplicaciones robóticas. Dicha plataforma seguirá las premisas de organización e interacción de la arquitectura de control teórica JDE[24], de la que obtendremos un diseño software, que finalmente implementaremos en el sistema *jde+*. Dicho sistema o plataforma de desarrollo, nos permitirá construir aplicaciones robóticas basadas en la teoría de JDE, brindandonos todos sus mecanismos y conceptos en pos de simplificar al máximo dicha tarea.

1.2 Arquitecturas de control

Como se vio en el apartado anterior, una de las claves para la generación de comportamientos robustos, era la correcta organización interna de los sensores y actuadores, así como del software que los gobierna. En esta sección mostraremos el estado del arte desde el punto de vista teórico de la arquitectura, mostrando las principales lineas, y desde el punto de vista práctico, veremos un par de ejemplos de arquitecturas software relevantes.

1.2.1 Arquitecturas teóricas

Desde un enfoque sistémico un robot móvil se puede contemplar como un sistema que tiene por entradas los datos sensoriales y por salidas los posibles comandos sobre los actuadores. En este ámbito, de modo informal, entendemos que arquitectura es todo lo que hay entre los sensores y los actuadores. De manera más precisa llamamos arquitectura cognitiva, teórica, o simplemente *arquitectura* de un robot autónomo

a *la organización de sus capacidades sensoriales, de procesamiento y de acción para conseguir un repertorio de comportamientos inteligentes interactuando con un cierto entorno*. La arquitectura determina los comportamientos que exhibe un robot autónomo.

La importancia de la arquitectura radica en que los robots móviles son sistemas con un alto grado de complejidad, y cuanto más complejo es un sistema más relevancia cobra el papel de la organización de sus componentes. Normalmente para tareas sencillas hay muchas maneras de abordarlas y el modo en que se resuelven, *el cómo*, casi no tiene importancia. Sin embargo al aumentar la complejidad la organización influye más en el resultado, puede incluso llegar a ser determinante: con una buena organización se consigue el objetivo y con una mala no. Por ejemplo, el papel de la arquitectura es más visible cuando la complejidad aumenta, bien porque el robot deba exhibir un repertorio completo de diferentes comportamientos, bien porque los sensores proporcionan una gran cantidad de datos del entorno, no todos relevantes. Además *el cómo* puede influir notablemente en la calidad del sistema, ya sea en su funcionamiento, en el tiempo de desarrollo necesario para construirlo o en otros aspectos. Para lograr sistemas repetibles y adaptables su organización interna ha de ser clara, accesible y fácilmente modificable. El estudio de cómo organizar estos procesos internos en robots es lo que denominamos arquitectura. Avances en la arquitectura conducen a comportamientos aun más complejos y a aumentar la autonomía de modo fiable.

El hardware de un robot en sí mismo no hace nada *motu proprio*, es inerte. Lo que da vida a estos componentes materiales es el *software*, los programas. Ellos recogen la información proporcionada por los sensores y calculan los comandos que se envían a los actuadores, determinando de este modo el comportamiento del robot. Si queremos que el robot se comporte de determinada manera tenemos que programarlo para que lo haga. Esos programas siguen cierta arquitectura software. Puede ser algo tan sencillo como un programa único en bucle infinito que ejecuta cíclicamente la secuencia: leer datos de los sensores, pensar, actuar. También puede ser varios procesos concurrentes que se comunican entre sí enviándose datos, estableciendo flujos de información entre ellos. O quizá habrá procesos que se dediquen a elaborar cierta información de salida desde sus datos de entrada, procesos que tomen decisiones, etc. Las distintas arquitecturas conceptuales se concretan en cierta arquitectura software, es decir en los programas que se ejecutan en los procesadores del robot.

En una primera aproximación generar un comportamiento autónomo se puede ver como un simple problema de control, en el que se trata que un sistema físico actúe de determinada forma en cierto entorno. Sin embargo, el problema de la arquitectura es más complejo que un problema de control. Hablamos de un controlador cuando tiene por entrada señales con la información relevante que están definidas con nitidez, cuando los objetivos no cambian de naturaleza (si acaso cambian las referencias) y cuando lo "único" que hay que hacer es elegir entre determinadas posibilidades de actuación para conseguir el objetivo. Por el contrario, hablamos de arquitectura cuando incluimos la tarea de construir o seleccionar esas señales de entrada y los sensores ofrecen una cantidad desbordante de información no específica. También cuando la naturaleza de los objetivos puede variar enormemente o cuando el robot tiene varios objetivos, con prioridades dinámicas que dependen de las necesidades actuales y de la situación del entorno. Tenemos un problema de arquitectura cuando el sistema ofrece un *repertorio* muy variado de comportamientos y hablamos de la interacción entre unos comportamientos y otros, cambio de un modo a otro, etc.

No existe *la arquitectura* válida para todos los entornos y para todos los comportamientos. Tradicionalmente el estado del arte en este campo se ha dividido en tres grandes corrientes: los sistemas deliberativos, los reactivos y los híbridos. Clasificaciones similares existen en [21, 7, 6, 12] y en muchas tesis de los años 90 [26, 27]

Sistemas deliberativos

La disciplina de la *Inteligencia Artificial* ha influido notablemente en el desarrollo de la robótica móvil. Las arquitecturas tradicionalmente consideradas como *deliberativas*, o de la escuela simbólica son las herederas de las investigaciones en IA clásica y fueron las primeras en surgir en el mundo de la robótica móvil para generar comportamiento, siendo las dominantes hasta finales de los años 80.

En general, estas arquitecturas asumen que generar comportamiento en un robot consiste en ejecutar una secuencia de acciones calculada de antemano (*plan*), que se elabora razonando sobre cierto modelo simbólico del mundo. El robot necesita una descripción simbólica del estado de su entorno (por ejemplo, un mapa de ocupación, con sus paredes, pasillos y puertas, o una relación de hechos lógicos sobre ese estado), y de su funcionamiento (por ejemplo, el efecto de sus propias acciones). La inteligencia del robot se halla principalmente en el planificador, el cual delibera sobre los símbolos del modelo del mundo y genera un plan de actuación explícito. El plan elaborado contiene el curso deseado de la acción en el futuro para conseguir el objetivo. Por ejemplo, la secuencia de operadores para conseguir el estado objetivo a partir del estado actual o la ruta óptima para llegar al punto destino.

Su inserción en robots reales, consiste en ejecutar infinitamente el bucle *Sensar-Modelar-Planificar-Actuar* [21]. En el primer paso (*Sensar*) se obtienen los estímulos del mundo exterior. En el segundo (*Modelar*) se construye una representación simbólica del mundo. En el tercero (*Planificar*) se elabora un plan, razonando sobre la representación simbólica del mundo que se obtuvo antes. Y por último (*Actuar*), se ejecutan las acciones del plan elaborado.

Las principales limitaciones de este tipo de arquitecturas son, que la planificación presenta explosión combinatoria en cuanto el problema a manejar crece en complejidad [3] y, que la idea de mundo cerrado, invariable no se ajusta a la realidad y los mecanismos lógicos no preven cambios arbitrarios en el mundo, de manera que este tipo de sistemas no maneja bien la incertidumbre del mundo real.

Como ejemplos de este tipo de arquitecturas, tenemos el robot Shakey[9] (figura 1.3(a)), el robot Hilare[10], o los sistemas SOAR y RCS.

Sistemas Reactivos

A finales de los años 80, era notable la falta de flexibilidad de los robots reales conseguidos hasta la fecha, guiados por el enfoque deliberativo. Quizá motivados por esto, varios investigadores comenzaron a replantearse el modo de generar comportamiento, y el uso de los planes, que era la pieza central del paradigma deliberativo [23] [4] [13].

Fruto de este replanteamiento nace el *enfoque reactivo*. Esta escuela reactiva, quizá liderada por los trabajos de Rodney Brooks [8] [10], provocó un giro drástico en cuanto al modo de generar comportamiento. El nuevo enfoque hace hincapié en una ligazón más directa entre los sensores y los actuadores, sin pasar por las etapas intermedias que utilizan los robots deliberativos. De esta manera la reacción a los eventos era mucho más rápida. Uno de los pasos intermedios que se consideran innecesarios es el modelo del mundo, y otro el manejo de símbolos. En este sentido el enfoque reactivo es subsimbólico, y argumenta que no es necesaria la representación simbólica, ni el razonamiento sobre símbolos para generar comportamiento. En términos de Murphy [20], este planteamiento reduce las primitivas esenciales a sensar y actuar, que están directamente ligadas, y deja a un lado la de planificar.

[9] http://www.sri.com/technology/shakey.html
[10] http://www.laas.fr/∼matthieu/robots/

Dentro de este enfoque, podemos distinguir entre sistemas reactivos puros, y sistemas basados en comportamientos. Teniendo ambos puntos en común, los segundo representan una evolución sobre los primeros.

Sistemas Reactivos Puros Los sistemas reactivos puros, se basan en la conexión de situaciones y acciones para generar comportamiento. El funcionamiento principal es un rápido bucle que chequea los valores sensoriales, busca en la situación que le corresponde y ejecuta la acción recomendada por la asociación. Lo que implica que no hay que esperar a tener el entorno completamente modelado para emitir una actuación ventajosa. Esta asociación situación-acción se almacena de algún modo dentro del sistema, ya sea con una tabla rasa, con unas reglas borrosas, manualmente, con un programador de autómatas, etc.

La principales limitaciones de estos sistemas son las siguientes. Requiere la anticipación en tiempo de diseño de *todas* las situaciones en las que se va a encontrar el robot y el cálculo de la respuesta adecuada para todas ellas, siendo incapaces de improvisar ante situaciones desconocidas. La ausencia de anticipación temporal no les permite reaccionar hasta haber detectado (sensado) una situación determinada (ej. un obstáculo).

Sistemas Basados en Comportamientos La característica principal de los sistemas basados en comportamientos (SBC) es que abogan por la distribución del control y la percepción en varias unidades que funcionan en paralelo, llamadas de diversas maneras: niveles de competencia, comportamientos, esquemas, agentes, controladores, etc. Esta descomposición por actividades contrasta con la descomposición funcional y el control monolítico típicos de las arquitecturas deliberativas. La idea subyacente es que el comportamiento complejo de un sistema es una propiedad emergente que surge de las interacciones de los componentes básicos entre sí y con el entorno.

Cada componente es un bucle rápido, reactivo, desde los sensores a los actuadores, que tiene un objetivo propio. En general cada componente unitario implementa las reacciones adecuadas ante ciertos estímulos accediendo directamente a los datos sensoriales que necesita y emitiendo control a los actuadores. Por ello, además de suponer una distribución del control, este paradigma también propone la distribución de la percepción. En este sentido no necesita representación centralizada del entorno para generar comportamiento, ni modelos simbólicos. La información necesaria está directamente en las lecturas de los sensores.

Las arquitecturas basadas en comportamientos se diferencian unas de otras en el modo en que interactúan los diferentes comportamientos y en el método elegido para coordinarlos. Desde la jerarquía y la activación por prioridades, hasta la anarquía y la activación por un autómata finito de estados que hace las veces de árbitro.

Las principales limitaciones, son la escalabilidad de estos sistemas, muy complejos por encima de 4 niveles de competencia, la falta de anticipación al no mantener una representación simbólica y la dificultas para fijar objetivos.

Sistemas Híbridos

A la luz de las limitaciones observadas en las escuelas reactiva [18] y deliberativa [10] han surgido muchas aproximaciones híbridas, que tratan de combinar las ventajas de ambos en una única arquitectura. Por ejemplo, aunar las habilidades de anticipación y orientación a objetivos que proporciona la planificación con la flexibilidad frente a imprevistos que proporciona la reactividad.

El paradigma híbrido, dentro de su heterogeneidad, combina los principios de la escuela deliberativa y de la reactiva, tratando de complementarlos y conseguir de esta

Tiempo real	Un robot esta conectado a la realidad física mediante sensores y actuadores, por lo que su software debe ser ágil y tomar decisiones con vivacidad. Por esto, el software de un robot actúe en tiempo real, si no estricto, al menos blando.
Concurrencia	Un robot debe realiza numerosas tareas de manera simultánea, leer sensores, procesarlos, generar actuaciones, ... Para conseguir esto se requieren sistemas multitarea, para repartir las tareas en múltiples flujos de ejecución independientes.
Interfaz	El software para robots suele incorporar interfaces (habitualmente gráficas) con objetivos a la depuración del sistema, o para la interacción con él.
Distribución de cómputo	Las aplicaciones para robots son cada vez más distribuidas, permitiendo crear sistemas multirobot, o bien ubicar la carga computacional en varios nodos.
Heterogeneidad	La gran variedad de sensores, actuadores y demás dispositivos usados para la robótica requieren que los sistemas robóticos sean capaces de interoperar con múltiples interfaces.

Cuadro 1.1: *Requisitos específicos de los sistemas robóticos*

manera comportamientos más robustos y complejos. La parte reactiva se ha mantenido muy próxima a los sensores y actuadores reales, dotando al sistema de flexibilidad. Sin embargo, esta componente se no escala hasta completar todo el sistema, como en los sistemas basados en comportamientos, sino que tiene por encima una capa deliberativa que modula su funcionamiento. Esta parte deliberativa incluye la representación explícita de objetivos y la planificación como motor final del comportamiento.

La separación en parte deliberativa y reactiva atiende también a un principio de ingeniería software, que tiende a agrupar las cosas similares juntas. Así, a la hora de incorporar una nueva tarea o componente en una arquitectura híbrida hay que considerar si maneja información numérica que varía continuamente, o si la información que utiliza tiene una dinámica más lenta y es más abstracta.

Dentro de este enfoque se encuentra la arquitectura software TCA de Simmons o la arquitectura software Saphira de Konolige, que comentaremos en el apartado siguiente.

1.2.2 Arquitecturas software

El desarrollo de sistemas robóticos software no difiere especialmente del desarrollo aplicado a otros ámbitos del software. El desarrollador parte con una serie de requisitos, modela un diseño y finalmente realiza una implementación. La peculiaridad es que comparten una serie de requisitos específicos, que condicionan las características de este tipo sistemas software. Varios de estos requisitos se recogen en la tabla 1.1.

Históricamente el problema del desarrollo de sistemas robóticos generalmente se abordaba con soluciones *ad-hoc*, diseñadas e implementadas para un sistema específico y difícilmente portable. Con el paso de los años, el asentamiento de los fabricantes de robots y el trabajo de numerosos grupos de investigación, han ido apareciendo las arquitecturas software o diseños software, que se plasman en plataformas de desarrollo que permiten abordar el problema de construir aplicaciones robóticas de una manera más fiable, sencilla y eficiente.

Las arquitecturas software basadas en las doctrinas deliberativas consisten general-

mente en una serie de módulos, agrupados en una librería, que se ejecutan de manera secuenciada siguiendo el bucle *Sensar-Modelar-Planificar-Actuar* [21]. El uso de la abstracción funcional hace que sea generalmente sistemas monohilo. Como ejemplos de este tipo de arquitecturas software tenemos al robot Shakey [11] [22], al robot Hilare[12], o la arquitectura SOAR [17].

Por el contrario, las arquitecturas software basadas en las doctrinas reactivas suelen acoplarse en sistemas concurrentes donde múltiples módulos se ejecutan de manera concurrente. Esto se debe a que este tipo de sistemas debe atender gran cantidad de tareas, como ya vimos. Ejemplos de sistemas de este tipo son la arquitectura de subsunción de Brooks [8, 9] o la arquitectura AuRa de Arkin [5].

Las arquitecturas software híbridas aunan los dos enfoques, aplicando lo que mejor se adapta a cada situación. Estas son las arquitecturas software más utilizadas en la actualidad, por ello mostraremos dos de las arquitecturas software más destacadas de los últimos años, TCA de Simmons [29] y Saphira de Konolige [16].

Arquitectura TCA

El *control estructurado* [29] es la propuesta de Reid Simmons para combinar en una misma arquitectura aspectos deliberativos y aspectos reactivos. La implementación de esta idea es la arquitectura TCA (*Task Control Architecture*), que propone una descomposición del sistema en módulos concurrentes que intercambian mensajes entre sí. Unos módulos utilizan la funcionalidad de otros a través de paso de mensajes que necesitan ser tratados. Cada uno de ellos emite ciertos mensajes y tiene registrados manejadores para los mensajes que es capaz de procesar. TCA imbrica la planificación y la ejecución. Además introduce explícitamente la monitorización, y con ella la planificación de percepción (una suerte de atención).

Dentro de TCA los mensajes que puede emitir un módulo son: (1) informativo (*Information Message*), para comunicar cierta información a quien le pueda interesar, tipo multidifusión; (2) petición (*Query Message*), para preguntar cierta información, el receptor elabora la respuesta y se la devuelve; (3) objetivo (*Goal Message*), para introducir un nuevo objetivo al sistema; (4) comando (*Command Message*), para solicitar cierta actuación, enviar comando a los actuadores; (5) monitor (*Monitor Message*), para programar cierta comprobación de condiciones y (6) excepción (*Exception Message*), cuando ha detectado cierta emergencia. Tal y como muestra la figura 1.5(b), hay un módulo central que se encarga de coordinar los distintos módulos del sistema encaminando los mensajes hacia aquellos que son capaces de manejarlos.

Esta arquitectura ha sido utilizada en multitud de robots, entre los que destacan Ambler y Xavier [28]. Ambler es un robot de exteriores con patas, diseñado para caminar por terrenos escabrosos. Xavier es un robot de interiores con ruedas capaz de entregar correo en entornos de oficina.

Arquitectura Saphira

Una arquitectura muy exitosa en robots de interiores es Saphira [16], creada por Kurt Konolige en SRI International. La arquitectura software asociada ha sido integrada y difundida en la plataforma de los robots comercializados por Activmedia. Recientemente se ha reescrito en C++ con el nombre de ARIA bajo licencia GPL. La orientación cliente-servidor de esta arquitectura software ha facilitado su uso en robots tan distintos como Flakey del SRI, los modelos Kephera de K-Team, los B21 de iRobot o los Pioneer de ActivMedia.

[11] http://www.sri.com/technology/shakey.html
[12] http://www.laas.fr/∼matthieu/robots/

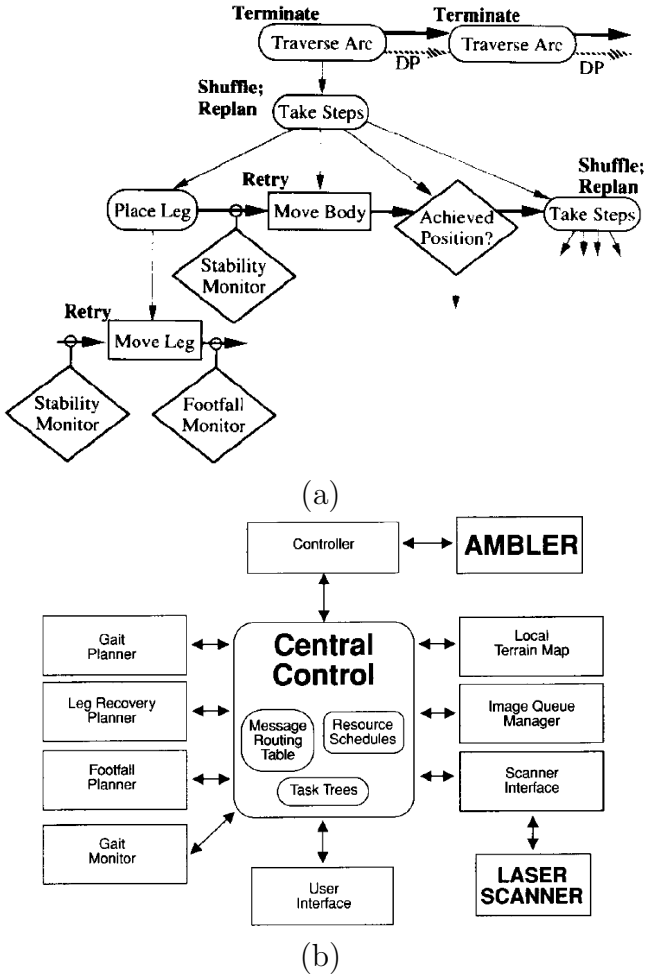


Figura 1.5: *Árbol de tareas en cierto instante para el robot Ambler (a) y la arquitectura de control como colección de módulos (b).*

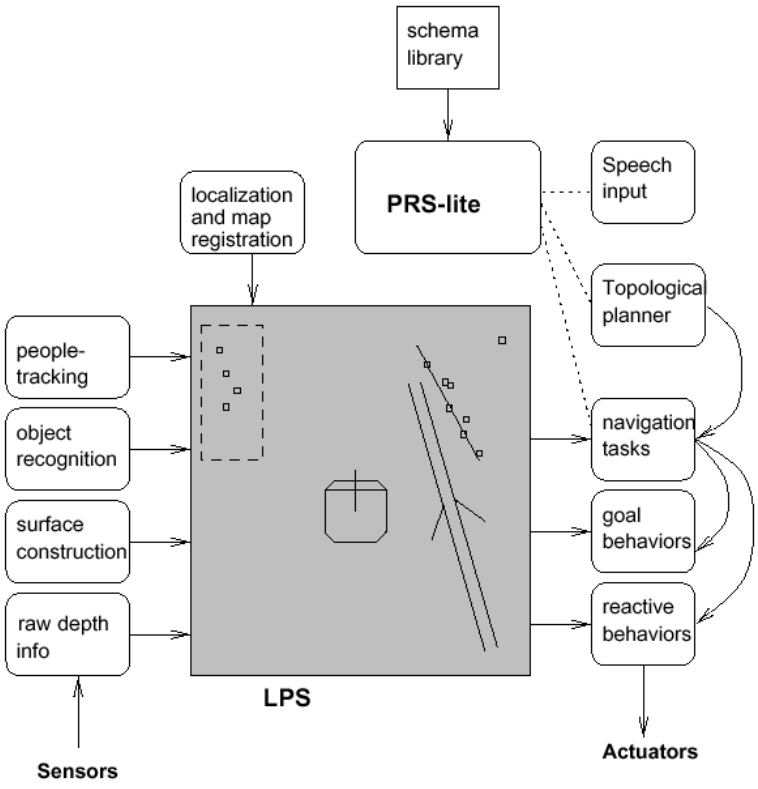


Figura 1.6: *Arquitectura Saphira*

En cuanto a la percepción, Saphira construye y actualiza constantemente una representación interna llamada *espacio perceptivo local*, en la cual fusiona información sobre el entorno próximo del robot. Este modelo está compuesto por *artifactos*, que representan internamente algún objeto real y que están en permanente correspondencia con los objetos homólogos de la realidad. Estos *artifactos* también pueden representar alguna configuración útil para guiar un comportamiento o algún objetivo instanciado. Varios procesos concurrentes mantienen ese modelo coherente, con sus símbolos *anclados* en el mundo real (*perceptual anchoring*), tal y como muestra la figura 1.6.

En cuanto al control, Saphira presenta dos niveles. En el nivel inferior dispone de unos comportamientos básicos escritos como controladores borrosos, es decir, como una colección de reglas borrosas con antecedentes y consecuentes. Estos comportamientos materializan las reacciones básicas, bien a los estímulos del entorno, bien a los objetivos instanciados (por ejemplo dirección de avance). Además, se engarzan con el espacio perceptivo local, de manera que los antecedentes de sus reglas hacen referencia a información almacenada en ese espacio, que se traduce a variables borrosas. La actuación efectiva se calcula promediando de modo borroso las salidas individuales de todos los comportamientos. Ese promedio es una combinación lineal, donde los pesos de cada comportamiento dependen de su prioridad, que es fija, y su adecuación a la situación actual, que se calcula con una reglas de contexto (*context-dependent-blending*). La figura 1.7 muestra un ejemplo de la combinación de dos comportamientos, en la que se ve como el cambio de contexto va variando el peso real de cada uno de ellos en la determinación de la actuación efectiva. Esta fusión permite tener en cuenta la tendencia hacia los objetivos incluso durante las maniobras reactivas.

Figura 1.7: *Arquitectura Saphira*

En el nivel superior Saphira se encarga de relacionar los comportamientos básicos con la consecución de metas intermedias y objetivos del robot. En este nivel se determina la activación y coordinación de los diferentes comportamientos básicos. El planificador PRS-Lite se encarga de ello, permitiendo la descomposición jerárquica de tareas en subtareas y ofreciendo un rico repertorio de mecanismos para especificar secuencias, despliegues condicionales de planes, ordenación de subobjetivos y de actuaciones.

1.3 *jde+*: Una plataforma de desarrollo para aplicaciones robóticas

El tema de este proyecto, es desarrollar el sistema *jde+*, una plataforma para la creación de aplicaciones robóticas, que se apoya en la arquitectura teórica de JDE [24]. Esta arquitectura ha sido desarrollada por el grupo de robótica de la URJC y plantea las aplicaciones robóticas como una colección jerárquica de esquemas concurrentes.

El sistema *jde+* pretende ser una herramienta para el desarrollo de aplicaciones robóticas. Su objetivo principal es facilitar al máximo la tarea de desarrollo de aplicaciones robóticas. Para ello el sistema *jde+* actuará como un *middleware*, entre la aplicación y el robot, aportando los mecanismos básicos y haciendo que el programador sólo tenga que preocuparse de la lógica de su aplicación.

Para obtener el sistema *jde+* inicialmente obtendremos una arquitectura software o diseño, que organice las ideas que expone JDE en conceptos implementables, y posteriormente construiremos la plataforma software o implementación del sistema. Así, en el capítulo 2 expondremos los objetivos y requisitos de este proyecto, aunque previamente introduciremos una serie de conceptos que nos ayudaran a comprenderlos. En el capítulo 3 mostraremos algunas de las herramientas software que posteriormente se usarán en la implementación. En el capítulo 4 desarrollaremos la arquitectura software y su implementación. El capítulo 5 mostrará las pruebas y ejemplos de uso que se han hecho sobre el sistema, base experimental que usaremos en el capítulo 6 para exponer las conclusiones de este proyecto. En este último capítulo también mostraremos la líneas de trabajos futuros que podrían dar continuidad a este proyecto.

Capítulo 2

Objetivos y metodología

Como hemos visto en la introducción, la creciente complejidad de las aplicaciones robóticas requiere planteamientos y herramientas cada vez más potentes para conseguir el éxito. En este contexto surgen las arquitecturas teóricas que plantean diferentes visiones acerca de la organización de los sensores y actuadores de un robot. Dichas teorías se plasman en sistemas software, sobre los que un desarrollador crea sus aplicaciones robóticas. Dichos sistemas suelen incorporar mecanismos y herramientas que facilitan la tarea de desarrollo.

El presente proyecto, *JDE+: Una plataforma de desarrollo para aplicaciones robóticas*, pretende construir un sistema software, que denominaremos jde+, que implemente una plataforma de desarrollo o *framework* para la creación de aplicaciones robóticas. La base del sistema será la arquitectura teórica JDE[24], desarrollada por el grupo de robótica de la URJC, y un cuidado diseño como pilar fundamental.

Comenzamos introduciendo algunos conceptos imprescindibles para comprender la motivación de este proyecto, tales como la teoría de JDE, y la implementación jde.c antecesora de jde+. A continuación describiremos los objetivos del proyecto de manera detallada, y enumeraremos los requisitos del sistema jde+. Y por último describiremos la metodología empleada en el desarrollo del proyecto.

2.1 JDE, una arquitectura para aplicaciones robóticas

Por aplicación robótica entendemos al programa ejecutado sobre una plataforma robótica capaz, de procesar una serie de estímulos captados por los sensores y generar comandos para los actuadores, describiendo un comportamiento observable. La creciente complejidad de los comportamientos que se pretenden desarrollar, obliga al programador de aplicaciones robóticas a organizar sus programas de una manera inteligente, para simplificar el desarrollo. Denominamos a esta organización arquitectura de control, o simplemente arquitectura, e introdujimos el estado del arte actual en la sección 1.2. JDE[24] es el nombre de la arquitectura de control ideada por el grupo de robótica de la URJC. Para JDE, una aplicación robótica tiene partes de percepción y partes de actuación, ejecutadas de manera concurrente. Cada una de estas partes, se denomina esquema, y se agrupa con otros en jerarquía para producir un comportamiento observable. Las características principales de un esquema son, que puede ser activado o desactivado a voluntad y que su funcionamiento puede ser modulado a través de parámetros. Así, unos esquemas *usan* la funcionalidad de otros activandolos y modulandolos según sus necesidades en un instante determinado. Por ello, la jerarquía (relación entre esquemas) varía dinámicamente según el contexto.

Podemos enmarcar JDE dentro de los sistemas basados en comportamientos, según

la clasificación que se hijo en la sección 1.2, aunque incluye ideas de otras tendencias como la etiología[1].

A continuación, repasaremos la teoría de JDE, profundizando en los aspectos más importantes, para construir el contexto teórico del sistema jde+. El resto de detalles puede encontrarse en [24].

Esquema: la unidad mínima

La unidad básica de JDE es el esquema, que nos permite encapsular una determinada funcionalidad, con la idea de formar comportamientos mediante la agrupación de varios esquemas. Podemos definir un esquema como *un flujo de ejecución independiente con un objetivo; un flujo que es modulable, iterativo y que puede ser activado o desactivado a voluntad*[24]. Hay dos tipos de esquemas, *esquemas perceptivos* y *esquemas motores o de actuación*. Los esquemas perceptivos se encargan de elaborar los *estímulos o percepciones sensoriales*, que pueden ser leídos por otros esquemas. Las entradas de estos esquemas pueden ser los sensores del sistema, o las percepciones de otro esquema. Los esquemas de actuación, útilizan los datos obtenidos de otros esquemas perceptivos, para generar sus salidas, que pueden ser comandos para los actuadores del sistema, o senales de activación y/o modulación para otros esquemas de nivel inferior (perceptivos o motores). La figura 2.1 lo describe gráficamente.

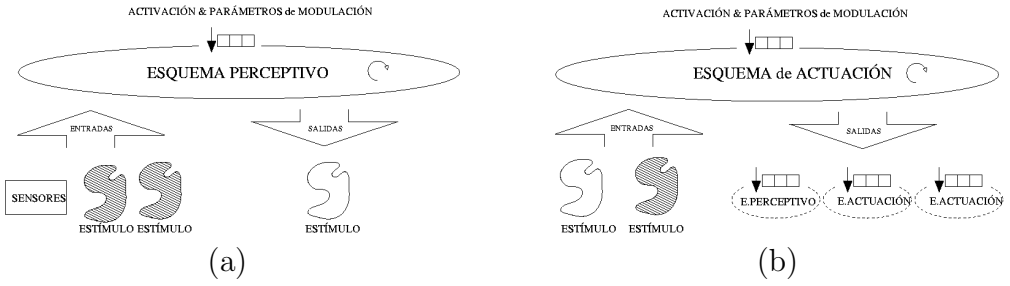


Figura 2.1: *Patrón típico de un esquema perceptivo (a) y de un esquema motor (b) en JDE*

Los esquemas, son por definición modulables, permitiendo el ajuste de su funcionamiento interno con dependencia de una serie de parámetros, variando el comportamiento de dicho esquema según lo module uno u otro esquema de nivel superior. Además, los esquemas son iterativos, realizando su misión en iteraciones que se ejecutan periódicamente. Dicho periodo, es un parámetro de modulación principal, que ajusta la cadencia de ejecución del esquema, que variará según interese su función (ej. un esquema perceptivo de visión, puede generar 10,15,20... fps, según se ajuste este parámetro). Por último, un esquema puede suspenderse dejando de producir salidas y de consumir recursos.

Esto último, se resuelve asociando estados a los esquemas, viendolos así como automatas, con varios posibles estados y transiciones entre estos. Los esquemas perceptivos, pueden estar en los estados DORMIDO y ACTIVO. En el estado DORMIDO el esquema permanece suspendido sin producir, ni consumir nada, mientras que en el estado ACTIVO, ejecuta periódicamente su iteración. La activación de un esquema perceptivo, consiste en pasar del estado DORMIDO al ACTIVO. Los esquemas motores, pueden estar en los estados DORMIDO, ALERTA, PREPARADO y ACTIVO. En el estado DORMIDO, al igual que los esquemas perceptivos, el esquema permanece suspendido. En el estado ALERTA, el esquema comprueba sus precondiciones. En el estado PREPARADO, el esquema compite por el control contra el resto de esquemas motores de

[1]Ciencia que estudia el comportamiento animal

su nivel. Y por último en el estado ACTIVO, ejecuta su iteración. La activación de un esquema motor, consiste en el paso del estado DORMIDO al estado ALERTA.

Para el caso de los esquemas motores hemos visto que tienen asociados cuatro estados : DORMIDO, ALERTA, PREPARADO y ACTIVO. Por defecto un esquema entra en la arquitectura en estado DORMIDO, es decir, sin consumir potencia de cálculo alguna. Antes de que un esquema motor pueda tomar el control real debe dar tres saltos en su estado de activación. El paso de DORMIDO a ALERTA lo determina un esquema de nivel superior, que considera conveniente preactivarlo porque puede llegar a ejecutar parte de su propio comportamiento si se dan las condiciones oportunas. En estado de ALERTA significa que el esquema está despierto, consumiendo tiempo de computador para chequear periódicamente si se dan sus precondiciones. El esquema puede estar infinitamente en ese estado latente, inhibido, a la espera de que la situación le sea favorable. Estas precondiciones evitan que un esquema se active cuando la situación es muy diferente de aquella para la que ha sido programado, con ellas se asegura que la acción se ejecuta en un contexto en el que resulta conveniente. En lugar de unas precondiciones clásicas, discretas (se-cumplen-las-precondiciones, no-se-cumplen), en JDE se utiliza un sistema más flexible, descrito en la figura 2.2.

En el esquema se almacena una cierta energía de activación, procededente de los distintos estímulos externos, estímulos y motivación internos, que hacen conveniente que ese esquema tome el control en la situación de ese instante. En la figura 2.2 se representa con el nivel mayor o menor de la zona rayada. Si esa energía es superior a cierto umbral, entonces el esquema considera que está en una situación favorable y pasa a estado PREPARADO, un avance más persiguiendo tomar el control del robot. Ese umbral de activación se representa en la figura 2.2 con la línea horizontal discontinua.

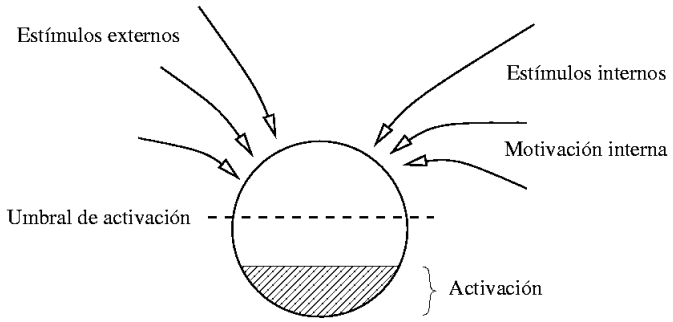


Figura 2.2: *Combinación aditiva de estímulos, umbral de activación.*

Lo siguiente es pasar al estado ACTIVO, en el que por fin, el esquema tomará el control en su nivel de actuación. JDE exige activación exclusiva, que significa que sólo debe haber un esquema motor ACTIVO por nivel en cada instante entregando comandos a sus niveles inferiores. Este enfoque evita la fusión de comandos, en caso de tener más de un esquema motor activo en el mismo nivel. Así, para conseguir el estado ACTIVO, un esquema motor PREPARADO debe competir con el resto de esquemas motores PREPARADOS de su nivel, pidiendo al esquema de nivel superior que los *arbitre* y elija un ganador.

Jerarquía dinámica: organización de los esquemas

Una vez presentada la unidad mínima, el esquema, hay muchas maneras de agruparlos para formar un sistema completo. JDE utiliza la jerarquía para regular el modo en que los esquemas se combinan e interactúan, permitiendo que unos utilicen la funcionalidad de los otros para conseguir una conducta observable. Así, la jerarquía aparece por el hecho de que los esquemas pueden aprovechar la funcionalidad de otros para

materializar la suya propia. Esto se plasma en JDE con la modulación y activación de un esquema sobre otro, y el posterior uso de la funcionalidad ofrecida por el esquema activado.

En JDE la salida de un esquema motor situado en los niveles más bajos de la jerarquía suelen ser los comandos directos que envía a algún actuador. Pero la salida también puede ser la activación y modulación de otros esquemas (esquemas hijos), los cuales en su ejecución, mientras persiguen o mantienen sus propios objetivos, están ayudando a que el que los activó (esquema padre) persiga o mantenga los suyos. Cuando un esquema es activado trata de conseguir su propio objetivo de algún modo, como una secuencia de acciones o como respuestas a ciertos estímulos del entorno, que son finalmente llevadas a cabo por los del nivel adyacente inferior. La activación del esquema inferior es la unidad de acción en ese nivel, es el vocabulario en el que se expresan sus actuaciones. Para flexibilizar esta subcontratación de funcionalidad el esquema superior puede sesgar ligeramente el comportamiento de los inferiores para adaptarlo más a su propio objetivo. En general, los esquemas ofrecen una funcionalidad específica, pero de manera flexible pues admite modulación a través de ciertos parámetros.

Este patrón de activación puede repetirse recursivamente, de modo que aparecen varios niveles de esquemas donde los de bajo nivel son despertados y modulados por los del nivel superior. Las activaciones en cadena van conformando una jerarquía de esquemas específica para generar ese comportamiento global en particular. La colección de esquemas se organiza en jerarquía para materializar cierta conducta. Esta jerarquía se reconstruye y modifica dinámicamente, según cambie el comportamiento a desarrollar o las condiciones por las que un esquema padre activa a cierto esquema hijo y no a otros.

Esto no debe llevar a pensar, que JDE es una arquitectura de activación directa, en la que un esquema padre activa a uno hijo durante un cierto tiempo a la espera de un resultado. Si nos referimos a los esquemas perceptivos, la activación es literal, el esquema se activa y empieza a emitir estímulos, que el padre puede ver. Sin embargo, en el caso de los esquemas de actuación, la activación únicamente expresa predisposición, colocando al esquema en estado de ALERTA, y dejado la activación final en manos del entorno (precondiciones) y de la competición entre hermanos motores (activación exclusiva). Esto contrasta con el enfoque funcional, en el que la ejecución de una acción requiere que el llamante se bloquee hasta el fin de la misma. En JDE, esto no sucede, ya que un esquema usa a otro, activandolo y modulandolo, para que conseguir su objetivo, pero siempre en una ejecución concurrente de ambas partes. Este enfoque aporta la necesidad primordial de una aplicación robótica, el manejo de múltiples tareas simultáneamente.

Selección de acción

Pero, ¿como influye el padre en la activación de sus hijos motores?, según hemos comentado, éste sólo expresa su deseo de que tal o cual esquema sea activado, por lo que en principio no realiza una selección de acción directa, marcando el compás de actuación. La respuesta es que el padre influye en la activación de sus hijos motores mediante la modulación, delimitando el subespacio perceptivo sobre el que se cumplen sus precondiciones, a zonas lo más disjuntas posibles (llamadas regiones de activación), intentando que sus hijos motores pretendan el estado ACTIVO en situaciones diferentes. Esto se puede ver en la figura 2.3 sobre la que se representa el espacio perceptivo, o lo que es lo mismo, el conjunto de todas las posibles situaciones que se pueden percibir. Las zonas blancas representan el subespacio perceptivo o subespacio de atención que corresponde con todos los posibles valores de los estímulos relevantes en el nivel actual. Las zonas delimitadas por linea continua, son las regiones de activación de cada esque-

ma motor de ese nivel, donde se cumplen sus precondiciones. A esto se le denomina *arbitraje de grano grueso*, basado en regiones de activación.

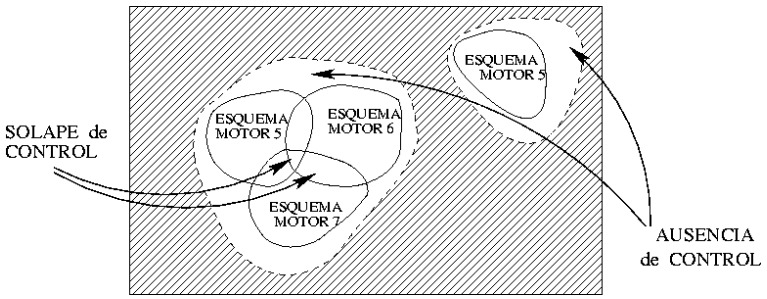


Figura 2.3: *Espacio perceptivo, subespacio de atención y regiones de activación.*

Como se puede apreciar en la figura 2.3, las regiones de activación no cubren todo el subespacio perceptivo y en algunos puntos se solapan, esto se denomina *vacío de control* y *solape de control* respectivamente, y queda fuera del alcance del arbitraje de grano grueso. En el primera caso, sucede que ningún esquema motores cumple sus precondiciones, quedando todos en el estado ALERTA, produciéndose una ausencia de control para esos estímulos en dicho instante. En el segundo caso, sucede lo contrario, varios esquemas motores ven cumplidas sus precondiciones pasando al estado PREPARADO y pretendiendo el ACTIVO. Ambos conflictos los debe resolver el padre, que conoce el contexto del problema mejor que nadie, arbitrando una coordinación muy específica y ajustada para la tarea en cuestión, cambiando la modulación de sus hijos, o sencillamente adjudicando un ganador en la competición por el control. A esto se le denomina *arbitraje de grano fino*.

El mecanismo de selección de acción de JDE se apoya en dos pilares. Por un lado, en la modulación de los hijos para que sus regiones de activación sean disjuntas (arbitraje de grano grueso), y por otro lado, en un arbitraje explícito del padre cuando surgen fallos de control, bien sean solapes o vacíos (arbitraje de grano fino). Tal y como está planteado, este mecanismo de selección de acción ofrece varias ventajas como su distribución, agilidad y escalabilidad. Por su propia naturaleza es un algoritmo distribuido, por lo que no es necesaria la existencia de un árbitro central. El sistema es homogéneo, solamente existen padres e hijos.

Para finalizar esta sección se resumen los principios de JDE en la tabla 2.1.

2.2 *jde.c*: antecedentes

jde.c se desarrolla como demostración práctica de toda la teoría contenida en la arquitectura de control JDE, su autor original es José María Cañas, aunque el sistema ha recibido muchas colaboraciones desde el grupo de robótica de la URJC. El sistema esta completamente programado en el lenguaje ANSI-C sobre una plataforma GNU/Linux, y en las últimas versiones se completa con al menos una docena de comportamientos, mas una interfaz de usuario que permite manejarlo desarrollada con XForms. Actualmente es la base de muchos proyectos del grupo de robótica de la URJC.

La unidad básica, el esquema, se representa en jde.c con una hebra de kernel, independiente de las demás. Tanto si es un esquema perceptivo, como si es de actuación, dicha hebra ejecuta periódicamente una función de iteración, a un ritmo controlado. Todas las hebras de los esquemas que no están en estado DORMIDO, se ejecutan de manera simultanea (sólo de manera virtual en un monoprocesador). Con esto se satisfacen

PRINCIPIOS	
Esquemas	
1	La unidad básica del comportamiento es el *esquema*
2	Un esquema es un flujo de ejecución independiente con un objetivo
3	Es modulable, iterativo y puede ser activado o desactivado a voluntad
4	Hay *esquemas perceptivos* y *esquemas de actuación*
5	Un esquema de actuación puede estar DORMIDO, en ALERTA, PREPARADO o ACTIVO
6	Un esquema perceptivo puede estar DORMIDO o ACTIVO
7	Un esquema aprovecha la funcionalidad de otro despertándolo y modulándolo, mientras él sigue ejecutándose concurrentemente
Jerarquía	
8	Los esquemas se estructuran en jerarquía, distinguiendo entre padres e hijos
9	En un instante dado el sistema consiste en un conjunto de esquemas, organizados en jerarquía, ejecutándose simultáneamente
10	Las jerarquías se construyen y modifican dinámicamente, y son específicas de cada comportamiento
Actuación	
11	La salida de un esquema de actuación pueden ser comandos a los actuadores o la activación y modulación de esquemas hijos
12	Mientras los esquemas hijos persiguen o mantienen sus propios objetivos, están ayudando a que el que los activó persiga o mantenga los suyos
13	El padre pone en ACTIVO a los esquemas perceptivos que buscan los estímulos relevantes para realizar su misión y pone en ALERTA a los esquemas de actuación que generan las respuestas convenientes cuando esos estímulos aparecen realmente
14	Si la situación del entorno satisface las precondiciones de un esquema en ALERTA entonces pasa a PREPARADO
15	Entre los hijos de actuación en PREPARADO se establece una competición de control, de modo que en cada instante sólo hay un único ganador que pasa a ACTIVO
16	Sólo los esquemas en ACTIVO pueden activar a otros esquemas
17	El padre monitoriza iterativamente los resultados de sus hijos, y puede cambiar su modulación e incluso cambiar de hijos cuando le convenga
Percepción	
18	La información útil para un comportamiento se expresa como una colección de estímulos, que se organiza en jerarquía
19	Un *estímulo* es una pieza de información que al menos un esquema de actuación necesita para tomar sus decisiones
20	Los estímulos pueden ser simples lecturas sensoriales, transformaciones suyas más elaboradas, o depender de otros estímulos menos abstractos
21	Los esquemas perceptivos producen estímulos y los mantienen actualizados
22	Los estímulos se materializan en variables que otros esquemas pueden leer
23	Los esquemas perceptivos se asocian a los esquemas de actuación para los cuales proporcionan información (*coordinación percepción-actuación*)
24	Los esquemas perceptivos proporcionan información de activación, de ejecución y de monitorización a los esquemas de actuación

Cuadro 2.1: *Principios fundamentales de JDE.*

los requisitos planteados para un esquema JDE, como flujo iterativo e independiente del resto.

Un esquema es generalmente una unidad de compilación C, con su archivo de cabeceras ".h" y su archivo de implementación ".c", que posteriormente se incluirá y enlazará en el sistema. Dicha unidad de compilación sigue un interfaz determinado, incluyendo una serie de funciones, que posteriormente se usarán para inicializar, activar o suspender (entre otras) el esquema desde el programa principal. Así, la construcción de un esquema consiste en programar las funciones de este interfaz, para desarrollar el comportamiento determinado del dicho esquema.

En la tabla 2.2, se muestra la interfaz típica de un esquema. El esquema `motors` se encarga de materializar un control en velocidad sobre la tracción y el giro del robot. Exporta cuatro funciones, con las que podemos parar, arrancar, suspender y despertar, respectivamente, el esquema `motors`. Además exporta tres variables, que en este caso son los parámetros de modulación que acepta el esquema. `v` modula la velocidad de avance en $\frac{mm}{s}$, `w` modula la velocidad de giro en $\frac{grados}{s}$, y `motors_cycle` modula el periodo de iteración del esquema.

```
extern void motors_shutdown();
extern void motors_startup();
extern void motors_suspend();
extern void motors_resume(int *brothers, arbitration fn);

extern float v; /* mm/s */
extern float w; /* deg/s*/
extern int motors_cycle;
```

Cuadro 2.2: *Interfaz de un esquema.*

Cada esquema, tiene asociado un identificador y una serie de variables para marcar su estado de activación. Estos datos, se implementan en el fichero fuente jde.c, que contiene una serie de arrays indexados con el identificador del esquema.

Es este fichero fuente, jde.c, el que implementa el programa principal, creando e inicializando los esquema que forman la jerarquía. La configuración de la jerarquía se lee de un fichero de configuración (por defecto jde.conf), donde se indican que esquemas se deben crear. Crear un esquema, implica crear su hebra e inicializar sus variables de estado. Los esquemas creados ingresan en la jerarquía en estado DORMIDO, pudiendo activarse posteriormente. Es importante indicar que sólo los esquemas enlazados contra el sistema en tiempo de compilación serán accesibles, y que sólo aquellos creados en tiempo de inicialización de jde.c podrán ser activados posteriormente.

Como se ha comentado, un esquema "nace" en estado DORMIDO. En este estado, un esquema no debería consumir ningún tipo de recurso en el sistema. Para implementar esto, se usan variables condición asociadas a cerrojos (mutex), que permiten bloquear una hebra a la espera de una señal. La señal, esperada se emite en la implementación de la función "resume" del interfaz comentado anteriormente, que coloca al esquema en un estado activo, y que será llamada por el esquema padre en el momento de despertar a ese hijo. De la misma manera, un padre dejará de nuevo bloqueado a un hijo en su variable condición, usando la llamada "suspend". Con estas funciones se consigue que cualquier hebra, cualquier esquema, pueda activar y desactivar a otras. Se pone con ello de manifiesto que las hebras no pertenecen a ningún nivel a priori. Dependiendo de quién las active pueden aparecer ahora en un nivel de la jerarquía y en otro diferente un instante después. Se materializa así el principio enunciado en la sección anterior de que la jerarquía es dinámica, se reconfigura en el tiempo, y de que los esquemas no se

adscriben a ningún nivel en concreto.

En el sistema jde.c, toda la comunicación relativa a la percepción y modulación entre esquemas, se realiza a través de memoría compartida, utilizando variables globales, que todas las hebras son capaces de ver. Este mecanismo tiene la ventaja de la sencillez de uso, aún siendo necesarios mecanismos de sincronización para el acceso concurrente.

En primer lugar, los estímulos perceptivos se materializan en variables, que los esquemas perceptivos definen y se encargan de refrescar periódicamente, y que los esquemas de actuación pueden leer cuando les interese. Así pues, la influencia que los esquemas perceptivos ejercen en los de actuación se da a través de variables comunes. La visibilidad está garantizada porque todas las hebras forman parte del mismo proceso y por ello comparten el espacio de memoria virtual donde se almacenan todas las variables. Del mismo modo, los esquemas perceptivos de un nivel inferior vuelcan sus resultados en ciertas variables que los esquemas perceptivos superiores leen para elaborar sus propios estímulos.

En segundo lugar, los parámetros de modulación que acepta un esquema se materializan en variables que el propio esquema define. Los esquemas superiores que activen y quieran modular el funcionamiento de un esquema inferior escriben los valores adecuados en esas variables parámetro, refrescándolas a su ritmo. El esquema inferior se encarga de leer con su propia cadencia esa variable para comportarse de un modo o de otro dentro de la funcionalidad que implementa.

El esqueleto típico de un esquema es un bucle infinito que típicamente ejecuta una función de iteración en cada ciclo del bucle. En realidad, lo que haga en cada ciclo depende del estado de activación en que se encuentre y si es un esquema de actuación, del resultado de la competición por el control en ese momento y en ese nivel.

```
for(;;)
   {
      if (state[SCH]==slept) pthread_cond_wait(condition[SCH]);
      else
         {
          gettimeofday(&a);
          perceptive_iteration();
          gettimeofday(&b);
          diff = (b-a);
          next = perceptive_cycle-diff;
          usleep(next);
         }
   }
```

Cuadro 2.3: *Pseudocódigo del esqueleto típico de un esquema perceptivo*

En la tabla 2.3 se ha puesto el bucle infinito de un esquema perceptivo a modo de ejemplo. En cada iteración se chequea primeramente si algún otro esquema decidió dormir al esquema. En ese caso la propia hebra se queda bloqueada sobre su variable condición, con la instrucción `pthread_cond_wait`. En caso contrario, toma el tiempo de sistema con `gettimeofday` y ejecuta una iteración de su función genuina (`perceptive_iteration` en la tabla 2.3). Al terminar esa función vuelve a tomar el tiempo del sistema. Finalmente la hebra se detiene un cierto intervalo, invocando a la llamada al sistema `usleep`. Gracias a esta pausa se garantiza el ritmo periódico de ejecución de la hebra que enuncia JDE. Esa cadencia suele ser un parámetro que la propia hebra admite, en el caso de la tabla 2.3 sería la variable `perceptive_cycle`. La doble llamada al cronómetro local sirve para descontar el tiempo de ejecución de

```
for(;;)
   {
   if (state[SCH]==slept) pthread_cond_wait(condition[SCH]);
   else {
      gettimeofday(&a);
      if (preconditions() <threshold)
         { /* my preconditions DONT match current situation */
         if (brothers_ready_or_active>1)
            { /* preconditions of another brother do match */
            state[SCH]=alert);
            }
         else
            { /* there is control absence */
            callforarbitration();
            if (state[SCH]==active) actuator_iteration();
             }
         }
      else
         {/* my preconditions match current situation */
         if (brothers_ready_or_active==0)
            {state[SCH]=winner)
            actuator_iteration(); }
         else
            { /* there is control overlap */
            callforarbitration();
            if (state[SCH]==winner) actuator_iteration(); }
         }
      gettimeofday(&b);
      diff = (b-a);
      next = actuator_cycle-diff;
      usleep(next);
      }
   }
```

Cuadro 2.4: *Esqueleto típico de un esquema de actuación, incluyendo la selección de acción*

la iteración genuina, que puede no ser despreciable, con lo que se ajusta más al ritmo periódico exacto.

En la tabla 2.4 se muestra también el esqueleto típico de un esquema de actuación. Es similar al de un esquema perceptivo, pero incluye la parte de selección de acción.

Tal y como muestra la cabecera de la función `miesquema_resume` en la tabla 2.2, cuando un padre despierta a un esquema hijo le pasa la lista de hermanos con la que tendrá que competir por el control. También le pasa a cada hijo una función propia de arbitraje explícito a la que invocar cuando éste detecte problemas en la competición por el control. Al despertarlos, el padre provoca que todos sus hijos pasen del estado DORMIDO a ALERTA. Esta selección refleja que el padre considera que sólo los esquemas despiertos pueden colaborar a conseguir su objetivo, y representa una vertiente finalista de la competición por el control en ese nivel. Hay cierta predisposición a que esos esquemas despiertos, y no otros, tomen el control.

En cada iteración del hijo, si el padre ha decidido dormirlo (por ejemplo porque él mismo ha perdido la competición por el control en su nivel), entonces el hijo se duerme a si mismo en su variable de condición. Si por el contrario el padre no ha desactivado a sus hijos, cada hijo comprobará el estado de sus precondiciones. Como vimos en la

sección 2.1 cada esquema de actuación tiene sus propias precondiciones, que miden el grado en que es aplicable a la situación actual. En esa función `preconditions()` de la tabla 2.4 es donde se implementa la consulta a los estímulos externos o internos y su adición siguiendo la suma heterogénea de estímulos. Si la activación resultante supera cierto umbral entonces el esquema promociona de ALERTA a PREPARADO. De algún modo ese nivel de activación continua representa la motivación a disparar este esquema, que ahora lleva la impronta de la situación del entorno. Los estímulos del entorno hacen subir la motivación de los esquemas de actuación adecuados a la situación actual.

Si las precondiciones del hijo se satisfacen, entonces él pasa a estado PREPARADO y chequea el estado de activación de todos sus hermanos, buscando alguno que también esté PREPARADO o ACTIVO. Si no lo encuentra entonces es el único hijo PREPARADO en ese instante, así que promociona a ACTIVO y realmente ejecuta su función de iteración genuina. Si encuentra otros hermanos con intenciones de tomar el control entonces ha detectado un *solape de control* e invoca a la función de arbitraje explícito que le dio el padre. La función de arbitraje le mantendrá su estado en PREPARADO si no resulta elegido ganador. O bien modificará su propio estado a ACTIVO si él resulta elegido ganador, en cuyo caso habrá que ejecutar la función de iteración.

Si las precondiciones del hijo no se satisfacen, entonces chequea el estado de activación de todos sus hermanos, buscando alguno que sí esté PREPARADO o ACTIVO. Si lo encuentra entonces este hijo no hace nada en esta iteración, pero si no lo halla, entonces ha detectado un *vacío de control*, e invoca también a la función de arbitraje explícito que le dio el padre. La función de arbitraje modificará su propio estado a ACTIVO si él resulta forzado a tomar el control, o lo mantendrá en ALERTA si no tiene que hacer nada.

La función de arbitraje explícito devuelve a aquél que la invoca si es el elegido para tomar el control o no, bien sea por obligación en el caso de vacíos de control, bien por selección cuando haya solapes de control. Dentro de esta función se comprueba el estado de activación de todos los hermanos y con ello se detecta si hay que resolver un solape o un vacío de control. Adicionalmente, el padre puede consultar todos los estímulos que haya activado con idea de poder resolver el conflicto, quizá atendiendo a alguna medida sensorial o el valor concreto de algún estímulo.

El algoritmo de selección de acción implementado es distribuido, y se simplifica por el hecho de que cada hijo puede comprobar en exclusión mutua el estado de activación de sus hermanos. Cada vez que un hijo ejecuta una iteración suya se está evaluando si el arbitraje es el correcto en ese nivel. El algoritmo utiliza a los hijos como detectores de solapes y vacíos de control. El padre no ejecuta ningún código de arbitraje, realmente lo ejecutan los hijos, cualquiera de ellos, cuando detectan algún problema, aunque el código esté escrito en el fichero que contiene al esquema padre. De este modo, el código de arbitraje explícito queda centralizado, muy compacto y ordenado, porque es el padre el que tiene el contexto y la información relevante para resolver el conflicto. Pero su ejecución está distribuida entre todos sus hijos.

Las precondiciones configuran la región de activación del esquema y normalmente estarán moduladas por el padre para que siempre se active un hijo y sólo uno a la vez. En el caso normal ocurre literalmente eso y la competición se resuelve directamente, de modo muy eficiente, sin necesidad de invocar a la función de arbitraje explícito.

Además de los esquemas de actuación y percepción, el sistema incluye varios esquemas, que no atienden a las descripciones realizadas. Los denominamos esquemas de servicio, pues su función es dar servicio al resto de esquemas del sistema. En la implementación actual existen tres esquemas de servicio. El primero maneja una interfaz gráfica, con la que es posible depurar el estado del resto de esquemas, mostrando representaciones gráficas de sus variables internas y demás información útil del sistema. En la figura 2.4 se puede ver una captura de dicha interfaz. Los otros dos son

`sensationsotos` y `sensationsoculo`, que se encargan de leer o escribir las variables base de los sensores y actuadores de la plataforma robótica. Al conjunto de estas variables se le denomina *api* de jde.c, y representa la abstracción de la plataforma sobre la que se ejecuta jde.c, haciendo que los esquemas sean independientes de dicha plataforma. Así, jde.c puede leer o escribir las variables del *api*, bien de una plataforma real, local o remota, o bien de una plataforma simulada.

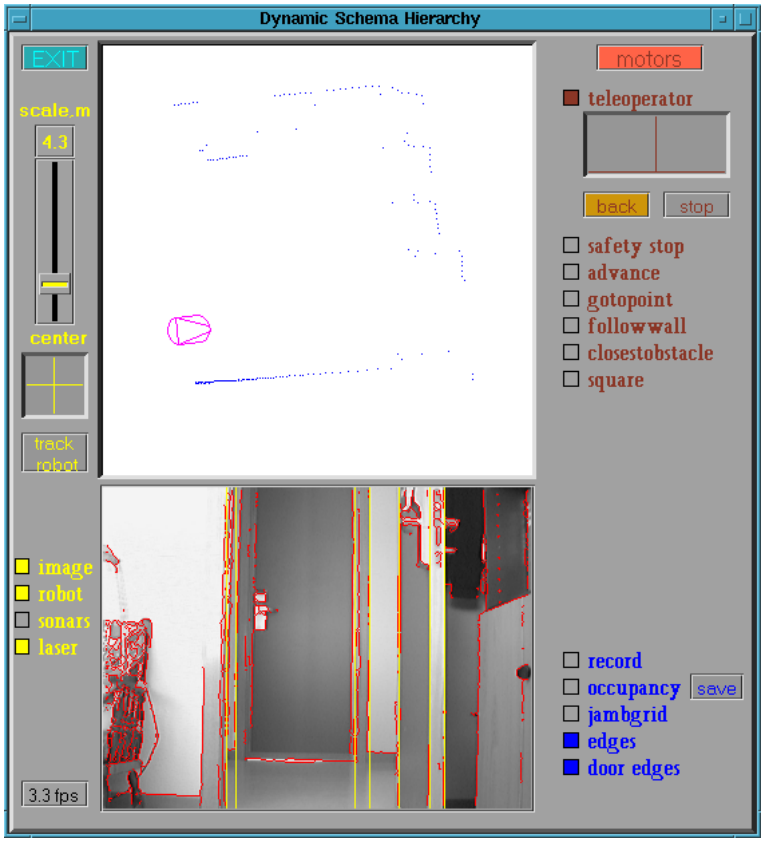


Figura 2.4: *Interfaz de depuración de la plataforma JDE*

Resumiendo, las principales ventajas de jde.c son:

- Implementa esquemas concurrentes.
- El algoritmo de arbitraje es distribuido.
- La implementación es muy eficiente y rápida.
- El *api* de variables abstrae a los esquemas de la plataforma sobre la que se ejecuta el sistema.

2.3 *jde+*: motivaciones

Aún habiendo demostrado su validez, a lo largo de una gran cantidad de proyectos, el sistema jde.c carece de ciertas características que lo hagan mas manejable, escalable y flexible. jde+ pretende ser una nueva implementación de la arquitectura teórica JDE, con nuevos enfoques y un diseño totalmente renovado, que solventen las limitaciones de jde.c.

Las principales limitaciones encontradas es jde.c son:

- Sólo permite una instancia de cada esquema.
- Integración de nuevos esquemas difícil.

- Mecanismos de comunicación poco sofisticados.
- Diseño difícilmente distribuible en varios nodos de computo.

El sistema jde.c, solamente permite **una instancia por esquema** representada por una hebra y un grupo de variables compartidas. En cuanto a la hebra, no habría problema en crear tantas como fuesen necesarias, ya que simplemente deberíamos indicar la función que ejecutarán, obteniendo varias hebras con la misma funcionalidad. El problema se encuentra en la manera en que los esquemas comparten sus datos, variables compartidas, que hacen inviable que mas de una hebra actúe sobre ellos. Así, jde.c no permite que existan múltiples instancias de un mismo esquema en diferentes partes de la jerarquía.

El enfoque de jde+, son esquemas que permiten *instanciación múltiple*, viendo un esquema como un componente (desde el punto de vista de la ingeniería del software) y al sistema como el *framework* sobre el que se ejecuta. Así, un esquema, es un ente compacto y autocontenido, que encapsula toda la funcionalidad e información requerida, y el sistema aporta los mecanismos para su creación, comunicación y ubicación.

La **integración de nuevos esquemas** en el sistema, es una parte básica, pues el sistema en si mismo no es nada sin los esquemas. El sistema jde.c, no aporta ninguna facilidad para ello. La inserción de un nuevo esquema consiste, en programar el módulo correspondiente (siguiendo uno de los esqueletos presentados anteriormente), modificar el programa principal para incluirlo y crearlo, y recompilar el sistema completo. Adicionalmente, puede que sea necesario modificar la rutina que lee la configuración del fichero jde.conf, para que identifique las opciones del nuevo esquema. Esto es así, porque el sistema y los esquemas están totalmente acoplados.

El enfoque que aporta jde+ a este problema es la *carga dinámica de esquemas*. Mediante este mecanismo, el sistema carga en tiempo de ejecución los esquemas que necesita para desarrollar el comportamiento especificado. La carga se hace de manera dinámica y bajo demanda, del mismo modo que los sistemas operativos cargan sus librerías dinámicas. Esto permite desacoplar el sistema de los esquemas, pudiendo compilar cada parte por separado.

Los **mecanismos de comunicación** de percepciones y modulaciones entre esquemas de jde.c, son simplemente asignaciones de variables globales. La protección contra condiciones de carrera la debe aportar el usuario, y en el caso más simple implica, el bloqueo del esquema hasta la entrega de los datos. Implementar un mecanismo de alerta ante la llegada de nuevos datos no resulta trivial. Tampoco existe una limitación en cuanto al alcance de las comunicaciones, ya que cualquier esquema puede escribir en las variables de otro. Esto, implica que la idea de jerarquía tal y como se describe en JDE, sólo existe en la cabeza del programador, y no en el sistema, lo que puede llevar a jerarquías malformadas.

Para estos aspectos jde+, implementa un mecanismo de comunicación mediante *paso de mensajes*, que por un lado, permite desarrollar diferentes políticas de comunicación, y por el otro, restringe las comunicaciones, permitiendo únicamente que se realicen entre padres e hijos, exactamente como se describe en la arquitectura teórica. Otra posibilidad que puede desarrollarse con estos mecanismos es la traza las comunicaciones para la depuración del sistema.

A medida que se desarrollan nuevos comportamientos, cada vez más complejos, se vislumbra a medio plazo, la necesidad de un sistema que permita la **distribución de computo**. jde.c, en principio no fue diseñado para ello, aunque como se ha programado con hebras, podría mejorarse su rendimiento haciéndolo correr sobre un multiprocesador. Sin embargo, una distribución estilo cluster, en la que se pudiese ubicar uno o varios esquemas en nodos de computo separados, no es posible.

La solución de jde+, es un diseño que incorpora esta posibilidad, aunque no será implementada ya que escapa al propósito de este proyecto.

Para el desarrollo de estas mejoras, y del sistema en general, se pretende utilizar una metodología orientada a objetos que permita una mayor abstracción de las partes.

2.4 Objetivos

Habiendo mostrado las motivaciones de este proyecto, en esta sección se enumeran los objetivos perseguidos, que deben desembocar en la implementación del sistema jde+. Estos serán el hilo conductor de todo el proceso de desarrollo junto con los requisitos marcados en la sección siguiente. Los organizamos según dos enfoques, diseño e implementación. Desde el enfoque de diseño, marcamos las ideas o características generales que se pretenden incluir en el sistema *jde+*, visto como sistema abstracto (o sistema de diseño). Desde el enfoque de implementación, marcamos la metodología y características concretas del sistema *jde+*, visto ahora como un programa.

El objetivo principal que persigue este proyecto, es construir el sistema *jde+*, una plataforma para el desarrollo de aplicaciones robóticas basada en JDE, que facilite al máximo las tareas de programación de robots móviles. Este pretende ofrecer una serie de mecanismos para que el desarrollador de aplicaciones robóticas, sólo tenga que preocuparse por la lógica de su comportamiento, y no de la interacción con el sistema, el robot (sus sensores o actuadores), o demás comportamientos existentes en el sistema. En este sentido, *jde+* pretende ser una herramienta para el desarrollo de aplicaciones robóticas, con fuertes conceptos teóricos en su concepción, pero enfocado para ser muy fácil de usar, permitiendo al desarrollador de aplicaciones robóticas concentrar sus esfuerzos en el fondo y no en la forma.

Para la consecución del objetivo principal de este proyecto, se deben alcanzar múltiples metas o subobjetivos. En lo que resta de sección describimos dichos subobjetivos, desde los enfoques comentados al inicio, numerándolos para poder referirnos a ellos en lo sucesivo.

Como subobjetivos enfocados al diseño de *jde+*, se pretende un sistema que permita la incorporación de nuevos esquemas de manera sencilla y que ofrezca mecanismos de comunicación potentes, como principales facilidades para el desarrollador de aplicaciones robóticas. También se desea que sea posible que coexistan múltiples instancias de un esquema, tanto simultaneas como en distintos instantes de tiempo, como principal limitación a superar de jde.c. Un subobjetivo secundario del diseño es incorporar ideas, que en un futuro próximo simplifiquen la ejecución distribuida del sistema en varios nodos de computo. Enumerándolos tenemos:

O-1 Incorporación de esquemas más sencilla.

O-2 Mecanismos de comunicación potentes.

O-3 Múltiples instancias.

O-4 Sistema distribuible.

Con *O-1* se pretende superar la dificultad de crear e insertar nuevos esquemas, patente en el sistema jde.c, que requiere conocimientos avanzados del funcionamiento del sistema, la modificación de al menos el programa principal, y por supuesto la recompilación de todo el sistema, para cada nuevo esquema creado. Con este objetivo se busca que el procedimiento para crear nuevos esquemas sea lo más simple posible, no requiera ninguna modificación del núcleo del sistema para su inserción y sólo necesite la compilación de las partes nuevas. Adicionalmente, el sistema cargará los esquemas

bajo demanda, dependiendo del comportamiento a desarrollar y de las necesidades de cada esquema.

O-2, por su parte, pretende plasmar la interacción entre esquemas con mecanismos de comunicación potentes, basados en paso de mensajes, que permitan desarrollar diferentes políticas de comunicación y limiten el alcance de las comunicaciones a lo expuesto en JDE. Siempre sin descuidar la simplicidad de uso.

En *O-3* se invierte la principal limitación de jde.c, la existencia de múltiples instancias. Este objetivo se relaciona con *O-2*, ya que parte de la solución se encuentra en él. Esta nueva característica permitirá crear nuevas jerarquías, que en jde.c no eran posibles, aunque abre las puertas a nuevas problemáticas no tratadas hasta ahora, como los bucles de dependencias entre esquemas o el uso masivo de recursos en jerarquías muy complejas.

Y por último, con *O-4* intentaremos esbozar las líneas maestras que simplifiquen la tarea de transformar el sistema *jde+* en un sistema distribuible, que permita ubicar esquemas en diferentes nodos de computo. Decimos esbozar, porque este subobjetivo sólo influirá en el diseño y no se plasmará con una implementación, ya que sale del alcance de este proyecto.

Enfocando ahora los subobjetivos desde el punto de vista de la implementación del sistema, se pretende un sistema orientado a objetos, eficiente y que funcione sobre plataformas GNU/Linux. Y enumerandolos tenemos:

O-5 Orientado a objetos.

O-6 Eficiente.

O-7 Implementado para plataformas GNU/Linux.

Con *O-5* se quieren aprovechar las facilidades de abstracción que brinda el paradigma de orientación a objetos a la hora de desarrollar un sistema complejo como *jde+*. Este subobjetivo, también lo podríamos ver desde la óptica del diseño, pues se aplicará el mismo paradigma. Con esto, la traducción del diseño a la implementación será directa.

El subobjetivo *O-6* pretende que el sistema sea lo más ligero posible, tanto computacionalmente, como espacialmente, para reservar la mayor parte de los recursos disponibles a la lógica de los esquemas que incorpore el sistema, generalmente compleja cuando hablamos de robótica. Así, se debe hacer un esfuerzo extra en la implementación para optimizar las tareas de gestión del sistema y el manejo de los esquemas que actúen en una jerarquía concreta.

El último subobjetivo, el *O-7*, indica que el sistema se implementará para las plataformas GNU/Linux, usando sus librerías y compiladores, aunque se realizará de manera que la tarea de portado a otros sistemas sea lo más simple posible, usando mecanismos estándar y herramientas que faciliten esta tarea.

2.5 Requisitos

Los requisitos del sistema vienen fijados, por la arquitectura JDE y por los objetivos marcados en la sección anterior. Estos requisitos se aplican al desarrollo del sistema, traduciéndose primero al diseño y posteriormente a la implementación, ambos temas tratados en el capítulo 4.

Requisitos impuestos por JDE

Estos son los requisitos impuestos por la arquitectura teórica JDE[24], que rigen como es un esquema, como se comporta y como interactúa con otros esquemas.

R-1 Un esquema tiene un flujo de ejecución independiente.

R-2 El flujo de ejecución de un esquema es modulable, iterativo y puede ser activado o desactivado a voluntad.

R-3 Existen dos tipos de esquemas, perceptivos y motores.

R-4 Los esquemas perceptivos tienen dos estados DORMIDO y ACTIVO.

R-5 Los esquemas motores tienen cuatro estados DORMIDO, ALERTA, PREPARADO y ACTIVO.

R-6 Los esquemas se estructuran en jerarquía, distinguiendo entre padres e hijos.

R-7 Un esquema puede usar la funcionalidad de otros para conseguir su objetivo.

R-8 Las jerarquías se construyen y modifican dinámicamente, y son específicas de cada comportamiento.

R-9 En cada nivel de la jerarquía debe haber sólo un esquema motor ACTIVO, el resto de situaciones producen conflictos de control.

R-10 La detección de conflictos de control se realiza en los hijos, la solución la provee el padre.

Requisitos genuinos de jde+

Estos son los requisitos genuinos del sistema jde+, rigen el funcionamiento del sistema, y fijan algunas restricciones con respecto a JDE, en el comportamiento de un esquema, para eliminar posibles ambigüedades. Por ejemplo, se fijan las direcciones en las que fluye la comunicación.

R-11 El sistema debe permitir activar un comportamiento.

R-12 El sistema debe permitir desactivar un comportamiento.

R-13 El sistema obtiene los esquemas mediante el mecanismo de carga dinámica, evitando la recompilación del sistema completo para la incorporación de nuevos esquemas.

R-14 Un esquema se materializa en el sistema a través de una o múltiples instancias, tanto simultáneas como en distintos instantes de tiempo.

R-15 Un esquema procura un objetivo, y define dependencias con otros esquemas.

R-16 El sistema crea una instancia bajo demanda, por dependencia de un esquema de nivel superior.

R-17 El sistema debe consumir la menor cantidad de recursos computacionales del sistema.

R-18 Una instancia de esquema interacciona con otras mediante paso de mensajes.

R-19 La comunicación estará gestionada por el sistema.

R-20 Existen cuatro tipos de mensaje: percepción, modulación, cambio de estado y petición de arbitraje.

R-21 Los mensajes de percepción y petición de arbitraje siempre van de un hijo a su padre.

R-22 Los mensajes de modulación y cambio de estado siempre van de un padre a uno de sus hijos.

2.6 Metodología de desarrollo

Todo proyecto software debe estar guiado por una metodología de trabajo o proceso de desarrollo software. Dependiendo de las características de dicho proyecto software deberemos aplicar uno u otra metodología. Este proyecto, es en definitiva un desarrollo software y por tanto debemos utilizar una metodología que guíe el proceso.

La metodología de desarrollo debe definir los productos que se deben obtener en el proceso de desarrollo. Los productos mas comunes son la documentación y el código fuente que trasformaremos en los programas. También se debe fijar un plan de trabajo, marcando hitos y fechas de entrega.

Este proyecto tiene las siguientes características:

Dos integrantes . En el desarrollo de este proyecto tan solo están involucradas dos personas, el tutor y el autor.

Fuertes raíces teóricas . La base del proyecto tiene un origen teórico (JDE[24]), por lo que parte de sus requisitos están fijados de antemano.

Complejidad media . La complejidad del sistema a desarrollar es media, ya que aunque no será demasiado grande incorporará una serie de mecanismos complejos (concurrencia, carga dinámica, ...).

Plazo de entrega cerrado . El plazo de entrega esta fijado para Septiembre de 2005.

El número de integrantes reducido de este proyecto implica que la documentación a realizar no será tan estricta, ya que la comunicación será constante y directa. La base teórica implica que parte de los requisitos están fijados de antemano, por lo que tan solo aparecerán una serie de requisitos genuinos del sistema que recojan los aspectos novedosos de este proyecto. La complejidad media del proyecto requerirá un desarrollo incremental, que vaya incorporando poco a poco las diferentes funcionalidades. Y por último, dado que el plazo de entrega esta cerrado se debe preparar un plan de trabajo que permita finalizar el proceso en la fecha indicada.

Por estas razones, la metodología que se eligió es el desarrollo en espiral, y los productos esperados son esta memoria y el sistema *jde+*. El desarrollo en espiral nos permite realizar un desarrollo incremental, obteniendo versiones funcionales en cada iteración. El plan de trabajo consiste en obtener versiones preliminares de ambos productos en las fechas marcadas por el tutor del proyecto. Para la coordinación de tareas se realizan reuniones semanales, bien presenciales, bien por mensajería instantánea.

En cada iteración del proceso se analizan los requisitos del sistema, se diseña e implementa una versión que incorpore dichos requisitos, se realizan pruebas y se planifica la siguiente versión. En total se han completado cuatro iteraciones, al final de cada una se obtuvo lo siguiente:

1. Versión 0.1 de *jde+*. Incorporaba los mecanismos de carga dinámica.
2. Versión 0.2 de *jde+*. Incorporaba parte de la funcionalidad del isc.
3. Versión 0.3 de *jde+*. Funcionalidad completa a falta de la reutilización.
4. Versión 0.4 de *jde+*. Versión final del sistema.

El desarrollo de esta memoria se intercala con el del sistema *jde+* a partir de la segunda iteración.

Capítulo 3

Herramientas y mecanismos software

Una vez mostradas las lineas base, que justifican el nuevo sistema jde+, describiremos algunas de las herramientas utilizadas en el desarrollo del sistema. Dichas descripciones se enmarcarán dentro del contexto del sistema, mostrando su uso dentro de este. Así, en este capítulo enumeramos las herramientas y mecanismos software que se han empleado en jde+, más las decisiones tomadas en caso de existir varias alternativas.

Comenzaremos hablando del lenguaje de programación elegido, seguiremos con el mecanismo de carga dinámica de código, mostraremos también las herramientas para programación concurrente y por último hablaremos de las utilidades usadas para la gestión del proyecto.

3.1 Lenguaje de programación C++

La primera elección, fue el lenguaje de programación a usar. El sistema jde.c, esta programado en C, y cuenta con mucho soporte (dispositivos, comportamientos, etc.) por lo que el lenguaje elegido debe permitir portar parte de este soporte de la manera más simple posible. Además, el tipo de aplicación a desarrollar, requiere un lenguaje versátil y eficiente.

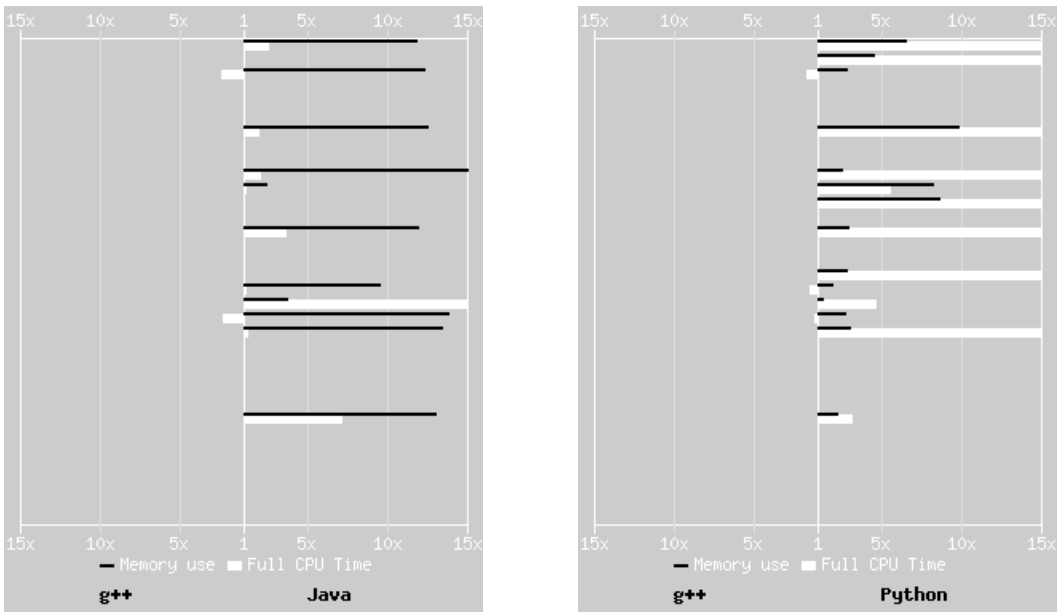
Con estas premisas, el abanico de posibilidades se reduce. Aspirantes a la elección son el lenguaje C, por supuesto, su evolución C++, Java y Python.

La decisión de diseño de usar el paradigma de programación de orientación a objetos, descarta el lenguaje C de la lista de aspirantes, ya que aunque podría emularse la funcionalidad, complicaría el código y lo haría ilegible. El resto de aspirantes, soporta de manera nativa dicho paradigma.

C++, es un lenguaje compilado, que nació con el *estigma* de ser C con clases. Es cierto que su sintaxis, es similar, y que a diferencia de C, es orientado a objetos, pero incorpora muchas más cosas que lo hacen merecedor de algo más que una simple evolución. Entre las más destacadas, se encuentra el manejo de excepciones y la programación genérica mediante plantillas (de la que hace gala la librería estándar STL). No es un lenguaje orientado a objetos puro, sino que mezcla los tipos que ya existían en C con las clases. Permite herencia múltiple y clases abstractas.

Java, también es un lenguaje compilado, pero a diferencia de C++ se ejecuta sobre una máquina virtual, que a su vez se ejecuta sobre el sistema operativo de la máquina física. Dicha máquina cuenta con versiones para los sistemas operativos más distribuidos, haciendo que Java sea portable de manera nativa[1]. El lenguaje mismo se inspira

[1]Siempre que se usen los recursos estándar de la máquina virtual.



(a) (b)

Figura 3.1: *C++ vs Java y Python*

en la sintaxis de C++, pero su funcionamiento es más similar al de Smalltalk (todo son objetos). Incorpora sincronización y manejo de tareas en el propio lenguaje (similar a Ada) e incorpora interfaces como un mecanismo alternativo a la herencia múltiple de C++.

Python, por su parte, es un lenguaje interpretado e interactivo, usado en sus inicios como lenguaje de scripting. Hoy en día, se han programado sistemas complejos con él, como son los proyectos de software libre Mailman (gestor de listas de correo) o Zope (gestor de contenidos). Su principales ventajas son, que no se necesita compilar nada y que puede ejecutarse de manera interactiva sobre un interprete, probando el código a medida que lo escribimos. Es muy versátil, e incluye gran cantidad de módulos.

Una vez presentados los aspirantes, comparamos las premisas con las características de cada lenguaje. La primera premisa, requería facilidad a la hora de portar el código de jde.c. En este aspecto, los tres lenguajes son similares, aunque evidentemente destaca C++, ya que sus sintaxis es un superconjunto de la de C. Java tiene una sintaxis muy similar a C++, y el porte no debería ser muy complicado. Por su parte Python nos permite embeber código C de manera nativa.

En cuanto a la versatilidad, volvemos a obtener un empate, ya que los tres lenguajes son multipropósito. Java y Python por su parte incluyen soporte para multiprogramación, comunicaciones y tipos de datos avanzados. C++ puede usar de manera nativa las librerías del sistema objetivo, GNU/Linux con posibilidades prácticamente ilimitadas, la librería STL incluye todo tipo de estructuras de datos.

Es en la última premisa, donde vemos la superioridad de C++, la eficiencia. La figura 3.1[1] muestra la comparativa de uso de CPU y uso de memoria de C++ contra Java y Python. Cada par de barras del eje vertical (blanca: uso CPU y negra: uso memoria), representa un test (ackermann, arboles binarios, conteo de palabras, generación aleatoria, creación de hebras, etc.). El lado hacia el que crece la barra implica que el lenguaje colocado en ese lado es menos eficiente, y la longitud indica cuanto menos eficiente es. Por ejemplo, en la comparativa 3.1(a), en el primer test Java es casi 2.5 veces menos eficiente en uso de CPU que C++.

Así, concluyendo, por la facilidad de reutilización del software programado para jde.c y por la superior eficiencia con el resto de aspirantes, el lenguaje de programación elegido ha sido C++. Dado que el sistema operativo sobre el que se realiza el desarrollo

es GNU/Linux, utilizaremos para su compilación g++, compilador de libre distribución (GPL) con amplio soporte en la comunidad UNIX.

3.2 Carga dinámica de código

Una de las partes más interesantes que presenta el sistema jde+, es la posibilidad de cargar los esquemas de manera dinámica en tiempo ejecución. Un esquema, en el fondo no es más que código ejecutable, por lo que jde+ debe ser capaz de ejecutar dicho código, que no fue compilado a priori en el sistema. Este mecanismo, lo vemos en los sistemas operativos, en la carga de *librerías dinámicas*, o en los navegadores web, en la carga de *plugins*.

Con este mecanismo, podremos mantener el sistema jde+ intacto, mientras desarrollamos nuestros esquemas, que más tarde se insertarán de manera dinámica. Otra posibilidad abierta con este mecanismo, es tener un sistema 24x7, que no requiera apagarse o reiniciarse nunca, actualizando la implementación de los esquemas en tiempo de ejecución.

Para conseguir esta funcionalidad en GNU/Linux usamos la librería `dl`[2] (dynamic load). Dicha librería tiene las siguientes funciones:

- void *dlopen(const char *filename, int flag);
- char *dlerror(void);
- void *dlsym(void *handle, const char *symbol);
- int dlclose(void *handle);

Con `dlopen` abrimos la librería dinámica `filename`, y obtenemos un manejador para acceder a ella. Con `dlerror`, obtenemos una cadena de caracteres que describen el último error que se produjo. `dlsym`, nos permite acceder a los símbolos contenidos en la librería a través del manejador que obtuvimos con `dlopen`. Por último `dlclose` cierra la librería y la descarga de memoría.

En la figura 3.2 se muestra un ejemplo de uso simple. Consiste en cargar la librería del sistema `libm`, que contiene funciones matemáticas, y hacer uso del símbolo `cos`, que es la función del coseno.

3.3 Paralelismo con `Pthreads`

La utilidad de la multiprogramación en la robótica, se hace imprescindible dado la gran cantidad de tareas que se deben hacer incluso en el comportamiento más simple (leer sensores, construir representación, refrescar pantalla, decidir actuación, ...). Por ello, cualquier software que acompaña a un robot incluye facilidades para la multiprogramación, véase el caso de `Ray-Scheduler` del robot B21 [24], las `Task` de Saphira [16] o `RoBios` para el robot EyeBot [19]. El sistema jde.c, usa un mecanismo de multiprogramación estándar, Pthreads, potente, sencillo de usar y muy bien documentado. Este mecanismo, día a día, se consolida en los sistemas operativos modernos, y nos ofrece hebras de kernel eficientes y robustas frente a bloqueos. jde+, también hace uso de Pthreads.

Una hebra, técnicamente, puede definirse como un flujo de instrucciones independiente, que puede ser planificado por el sistema operativo [2]. En un sistema UNIX una hebra existe dentro de un proceso, utilizando sus recursos (descriptores de fichero,

[2]Ver dlopen(3) del manual de programación de Linux

```
#include <stdio.h>
#include <dlfcn.h>

int main(int argc, char **argv) {
      void *handle;
      double (*cosine)(double);
      char *error;

      handle = dlopen ("libm.so", RTLD_LAZY);
      if (!handle) {
         fprintf (stderr, "%s\n", dlerror());
         exit(1);
      }

      dlerror();
      *(void **) (&cosine) = dlsym(handle, "cos");
      if ((error = dlerror()) != NULL)  {
         fprintf (stderr, "%s\n", error);
         exit(1);
      }

      printf ("%f\n", (*cosine)(2.0));
      dlclose(handle);
      return 0;
}
```

Figura 3.2: *Ejemplo de uso de* `dl`

espacio de memoria, etc.). Su flujo de ejecución está separado del de su proceso padre. Un proceso puede contener varias hebras que comparten sus recursos. Dichos flujos terminan cuando lo hace el proceso padre, o antes.

Las ventajas principales de utilizar multiprogramación basada en hebras en vez de múltiples procesos son:

1. El coste añadido a la creación de un nuevo flujo de ejecución, es mucho menor que la creación de procesos nuevos. La creación de una hebra requiere de estructuras de datos muy sencillas y pequeñas (pila, registros, propiedades de planificación), mientras que un proceso requiere de una estructura más compleja y costosa de inicializar.
2. Los mecanismos de comunicación entre hebras son mucho más eficientes, al utilizar un espacio de memoria compartido, y fáciles de usar que los mecanismos de comunicación entre procesos (IPC).

En la tabla 3.1 se muestra una comparación de tiempos de las rutinas `fork()` y `pthreads_create()`, encargadas de crear procesos y hebras respectivamente. Cada experimento crea 500000 procesos/hebras, el tiempo, en segundos, se ha medido con la utilidad time. El código de este experimento se puede encontrar en [2].

Sistema Linux 2.4	**fork()**			**pthread_create()**		
	real	**user**	**sys**	**real**	**user**	**sys**
Athlon 1800+ 256Mb Memoria	56.24	10.24	45.46	22.35	4.13	18.19
Pentium II 400Mhz 128Mb Memoria	201.31	30.83	170.45	78.67	17.48	61.13
AMD K6-3 400Mhz 128Mb Memoria	239.20	18.13	201.23	95.36	4.24	11.75

Cuadro 3.1: *Comparativa de tiempos fork() vs. pthread_create()*

Históricamente, cada plataforma implementaba su propia versión de software para multiprogramación. De esta manera, el desarrollo de aplicaciones portables era muy difícil, dadas las diferencias en las implementaciones. Por ello, y probada la eficiencia de la multiprogramación basada en hebras se creo una interfaz de programación estándar. Dicho estándar se especifica en el IEEE POSIX 1003.1c de 1995. Las implementaciones que siguen este estándar, se llaman hebras POSIX, o `Pthreads` (*Posix threads*).

`Pthreads` se define como un conjunto de tipos y funciones en el lenguaje de programación C. Las funciones incluidas en el API (alrededor de sesenta) permiten la gestión de hebras (creación, eliminación, etc.), el manejo de estructuras para exclusión mutua (*mutex*) y funciones para el uso de variables condición. El acceso concurrente a una variable compartida por varias hebras, puede producir lo que se conoce como condiciones de carrera (*race conditions*), conduciendo la ejecución a estados impredecibles. Uno de los mecanismos para el control del acceso concurrente, es el clásico semáforo, que permite el acceso en exclusión mutua a una hebra para modificar el valor de una variable compartida, manteniendo bloqueadas las demás (*locked*). Este mecanismo, también permite la sincronización de varias hebras, aunque `Pthreads` incluye un mecanismo más sofisticado, las variable condición, con las que podemos controlar si una hebra pasa a ejecutarse o no, dependiendo del valor de una de estas variables.

En GNU/Linux, la implementación de Pthreads se encuentra en la librería `pthread`. Su uso con el lenguaje C++, es exactamente igual que con el lenguaje C, salvo cuando se quiere usar sobre clases y objetos. En concreto, la función que nos permite crear nuevas hebras en Pthreads es `pthread_create`, y admite entre otros parámetros un puntero a la rutina a ejecutar en la nueva hebra. El tipo de esa función debe ser `void* (*f)(void*)`. En C, únicamente podemos pasar funciones en este parámetro, pero en C++, además tenemos los métodos de los objetos, y es aquí donde surge el problema.

Supongamos que tenemos la siguiente clase ejemplo:

```
class A{
 public:
   void* m1(void* a);
   ...
   ...
};
```

Si ahora queremos crear una hebra, que ejecute el método `m1`, haríamos:

```
#include <pthread.h>
#include <A.h>

int main(){
   A objetoA;
   ...
   ...
   pthread_create(...,objetoA.m1,...);
   ...
}
```

, donde ignorando el resto de parámetros, pasamos a `pthread_create`, la rutina que queremos que ejecute. Sin embargo, aunque aparentemente es correcto, al compilar este código, resulta que el tipo del método `m1`, no encaja con lo que espera `pthread_create`. Esto es, porque el tipo real del método `m1` es `void* m1(A* this,void* a)`, ya que el compilador incluye automáticamente el parámetro `this`, en el que pasa un puntero al propio objeto.

La solución a este problema tiene dos vertientes, la primera, simplemente consiste en crear una función que se encargue de llamar al método en cuestión. Para que esta función pueda funcionar con cualquier objeto de la clase, pasaremos un puntero al objeto que queremos invocar (tercer parámetro de `pthread_create`. Un ejemplo de función para la clase A sería:

```
#include <pthread.h>
#include <A.h>

void* f(void* objeto){
   A* pA = dynamic_cast<A*>(objeto);

   pA->m1(argumentos del método);
}

int main(){
   A objetoA;
   ...
   ...
   pthread_create(...,f,(void*)(&objetoA));
   ...
}
```

Esta solución, rompe el modelo de objetos, ya que el método `m1` podría ser privado, caso en que esta solución no valdría.

La segunda solución, consiste en declarar un método de clase (en C++ un método `static`), al que el compilador no antepone el puntero al objeto. El método sería similar a la función que vimos en la primera solución:

```
class A{
 public:
   void* m1(void* a);
   static void* f(void* objeto);
   ...
   ...
};

void* A::f(void* objeto){
   A* pA = dynamic_cast<A*>(objeto);

   pA->m1(argumentos del método);
}

int main(){
   A objetoA;
   ...
   pthread_create(...,objetoA.f,(void*)(&objetoA));
   ...
}
```

Así, con esta solución se puede acceder a cualquier parte del objeto, pública o privada.

En jde+, los esquemas son objetos, y cada uno contendrá una hebra, que usando la técnica mostrada, podrá acceder al estado interno del esquema. El sistema contiene más hebras, para tareas de servicio, que se apoyan también en esta técnica. Se entrará en más detalle en el capítulo 4.

3.4 Mecanismos de sincronización

Además de las hebras, jde+ usa otros mecanismos de programación paralela, también contenidos en Pthreads. Son los llamados mecanismos de sincronización, que permiten el acceso concurrente de forma segura a datos compartidos.

El uso de estos mecanismos es por lo general complejo, en la medida en que producen errores difíciles de diagnosticar, como son los interbloqueos o las condiciones de carrera, que además no suelen ser reproducibles. En este aspecto, el interfaz ofrecido por Pthreads no ayuda en absoluto. Un ejemplo, son los cerrojos, para los que existe una función que bloquea y otra que desbloquea. Un error típico, es olvidar desbloquear un cerrojo al olvidar la segunda función, produciendo un bloqueo en el sistema en cuanto se intente bloquear de nuevo. Si bien, es cierto que en las últimas implementaciones se incorporan chequeos de los casos más simples, aunque no es suficiente para la mayoría de los problemas.

Para abordar esto, en jde+, se opta por embeber estos mecanismos primitivos, en clases recubrimiento (`wrappers`), que ayuden en la medida de lo posible a evitar algunos problemas, haciendo mucho más sencillo su uso. Por otro, lado también se incorpora la notificación de errores mediante excepciones.

Se han implementado tres clases para esta misión:

Clase mutex : Recubre el tipo `mutex_t` de pthreads, y ofrece los métodos `lock()`, `unlock()` y `trylock()`.

Clase scoped_lock : Fácilita el uso de la clase anterior, automatizando las llamadas a `lock()` y `unlock()` dentro de un ámbito (o alcance)

(en C++ limitado por).

Clase condition : Recubre el tipo `condition_t` de pthreads, y ofrece los métodos `notify_one()`, `notify_all()` y `wait()`.

Un ejemplo comentado de su uso se puede ver a continuación. Se trata de un ejemplo típico de lectores y escritores, en el que varias hebras acceden de manera simultánea a unos datos para modificarlos. Al entrar en ambas funciones, se declara un objeto de la clase `scoped_lock`, al que se le pasa en el constructor, un objeto `mutex`. En ese momento, se ejecuta el método lock() del `mutex`, accediendo en exclusión mutua (solo uno a la vez) a la región crítica (donde se modifican los datos). Una vez dentro, se opera sobre los datos compartidos. En el caso de la función lector, se comprueba si hay datos en la cola, en caso negativo, usa la variable condición para bloquearse, hasta la llegada de datos. En el caso de la función escritor, inserta los nuevos datos en la cola, y usa la variable condición para notificar a un lector la llegada de datos para desbloquearlo. En ambas funciones, al finalizar, y salir del ámbito (o alcance) del objeto `scoped_lock` sl, se invoca automáticamente su destructor, momento en el cual se desbloquea el cerrojo.

```
#include <mutex.h>
#include <scoped_lock.h>
#include <condition.h>
#include <queue>

/*datos compartidos*/
queue<int> cola_enteros;

/*mutex y condicition para sincronizar*/
mutex m_global;
condition c_global;

void* lector(void*){
   scoped_lock sl(m_global);/*comienzo región critica. Lock automático*/

   if (cola_enteros.empty())
      c_global.wait(sl);/*esperamos hasta que en
                          cola_enteros haya algo*/
   consumir(cola_enteros);
   /*fin región crítica. Unlock automático al
     salir del ámbito de sl*/
}

void* escritor(void*){
   scoped_lock sl(m_global);/*comienzo región crítica. Lock automático*/

   producir(cola_enteros);
   c_global.notify_one();/*despertamos a un lector*/
   /*fin región crítica. Unlock automático al
     salir del ámbito de sl*/*/
}

/*hebras ejecutando concurrentemente las funciones lector y escritor*/
....
....
....
```

Sin embargo, estos mecanismos son muy primitivos, y construir un sistema concurrente complejo como jde+, se vería muy simplificado usando mecanismos más abstractos.

Jde+ es un sistema orientado a objetos, en el que unos objetos interactúan con otros de manera concurrente. Para este problema existen dos mecanismos específicos, descritos por los patrones de diseño *monitor* y *objeto activo*[25]. El *monitor* aplica la semántica "solo uno cada vez", al acceso a los métodos de un objeto. Así, se secuencia el acceso al objeto manteniendo el estado del objeto siempre en exclusión mutua. El Objeto Activo mantiene la semántica "solo uno cada vez", pero esta vez encolando las peticiones. De esta manera, el que invoca el método nunca se bloquea, como pasa en el *monitor*. Una hebra interna se encarga de procesar las peticiones posteriormente. Para nuestro sistema el *monitor* será suficiente.

Una implementación típica de un *monitor* usando las primitivas descritas se puede ver a continuación. Los métodos 1 y 2 ejecutan en exclusión mutua sus implementaciones ("metodoX_i"), cumpliendo así la semántica "solo uno cada vez". Una vez dentro de la región crítica se podrían añadir chequeos de condiciones, dando muchísima más flexibilidad al *monitor*.

```
#include <mutex.h>
#include <scoped_lock.h>
#include <condition.h>

class MonitorA{
 public:
   void metodo1();
   void metodo2();
   ...
   ...
 private:
   void metodo1_i();
   void metodo2_i();
   ...
   ...

   mutex monitor_lock;
   ...
   ...
}

void MonitorA::metodo1(){
   scoped_lock sl(monitor_lock);

   método1_i();
}

void MonitorA::metodo2(){
   scoped_lock sl(monitor_lock);

   metodo2_i();
}

MonitorA::metodo1_i(){
```

```
    ....
    ....
}

MonitorA::metodo2_i(){
    ....
    ....
}
```

3.5 Patrones de diseño

Se debe tener en cuenta en todo momento que no se trata de "reinventar la rueda", por lo que se debe intentar reconocer en la medida de lo posible los escenarios ya tratados por patrones de diseño típicos, y en esos casos aplicar la solución que propone dicho patrón. El diseño de *jde+* incorpora diferentes patrones de diseño adaptados a su caso particular. En esta sección describimos de forma genérica estos patrones.

En total se han aplicado cuatro patrones de diseño, el patrón *monitor*[25] descrito en la sección 3.4, y los patrones *factoría abstracta*, *state* y *singleton*[14] que pasamos a describir en esta sección.

3.5.1 Factoría Abstracta

Este patrón de diseño se utiliza para crear objetos de los que no se conocen sus métodos constructores, o incluso el objeto entero, salvo su nombre o identificador y su clase base. A través de una clase, denominada clase factoría, que tiene un interfaz conocido (típicamente un método `create_instance()`), se crean instancias de ese objeto.

El patrón de diseño *factoría abstracta* en su forma más genérica, utiliza las clases que aparecen en la figura 3.3. La clase `classfactory`, es la base de las factorías y define el método abstracto `create_instance` que devuelve un objeto. Con `object` representamos el objeto más genérico. Como ejemplo tenemos la factoría `factoryA`, que es capaz de instanciar objetos de la clase `A`. Su uso consiste en la existencia de varias factorías, probablemente indexadas de algún modo, de las que el usuario selecciona una e instancia un objeto por mediación de ésta.

3.5.2 State

La motivación de este patrón de diseño es agrupar todo el comportamiento condicional en una serie de clases que representarán los diferentes estados de operación. Estas clases se agrupan bajo una clase abstracta que define su interfaz. Dicho interfaz incluye las operaciones condicionadas por el estado, que cada una de las clases derivadas implementa según el estado que representa.

En la figura 3.4 vemos un ejemplo, en el que la clase `TCPConnection` representa una conexión tcp que actúa en función de su estado. El estado lo representa la clase abstracta `TCPState` de la que se derivan los posibles estados de una conexión tcp (para simplificar se presentan tres), `TCPEstablished`, `TCPListen` y `TCPClosed`. La clase abstracta define los métodos `Open`, `Close` y `Acknowledge` que cada clase derivada implementa atendiendo a su estado. Con esto la clase `TCPConnection` representa su estado con una de las clases derivadas de `TCPState` y en la ejecución de sus métodos

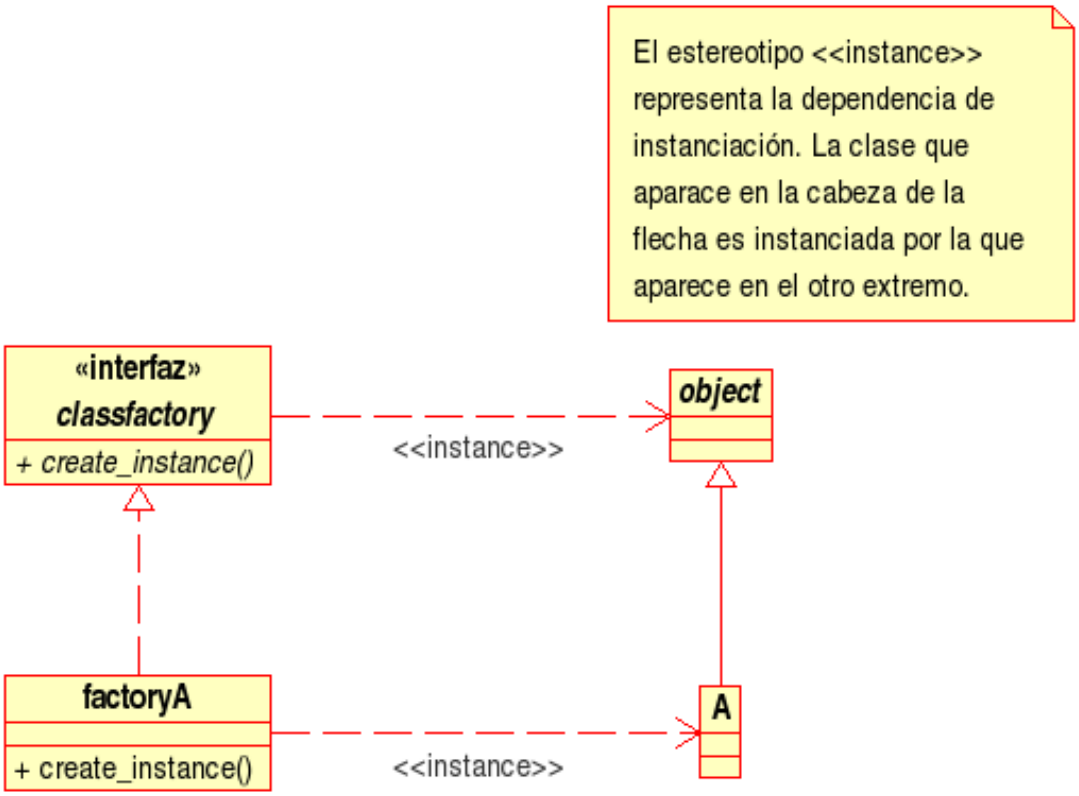


Figura 3.3: *Patrón de diseño* factoría abstracta

invoca al concreto del estado actual. El cambio de estado se puede realizar en la clase `TCPConnection`, o mejor aun permitiendo que cada clase estado defina su sucesor.

3.5.3 Singleton

Este patrón de diseño describe una clase de la que sólo existe una o determinadas instancias. Esto se lleva a cabo con miembros de clase, que controlan la creación de instancias de esa clase. La figura 3.5 muestra una clase que implementa este patrón de diseño. Los miembros que aparecen subrayados indican miembros de clase.

Al ejecutar el miembro de clase `instance` (en C++ singleton::instance()) obtenemos la única instancia de dicha clase.

3.6 Gestión del proyecto: `GNU Autotools`

La gestión de un proyecto, relativamente complejo, como jde+, con cerca de 4000 líneas de código distribuidas en cerca de 30 ficheros fuente, se ve enormemente facilitada con el uso de herramientas que permitan automatizar tanto la compilación como la distribución a terceros.

Una opción consiste en usar algún tipo de IDE[3], que automatice todas las tareas de gestión. En nuestro caso, se optó por usar el editor `emacs`, por sus fantásticas características para escribir código fuente, y una serie de herramientas, a parte, para la gestión del proyecto. Dichas herramientas son conocidas en el mundo del software libre como `GNU Autotools`, y permiten mediante macros automatizar las tareas de gestión. Las herramientas usadas son `Autoconf` y `Automake`.

[3]Integrated Development Environment: entorno de desarrollo integrado.

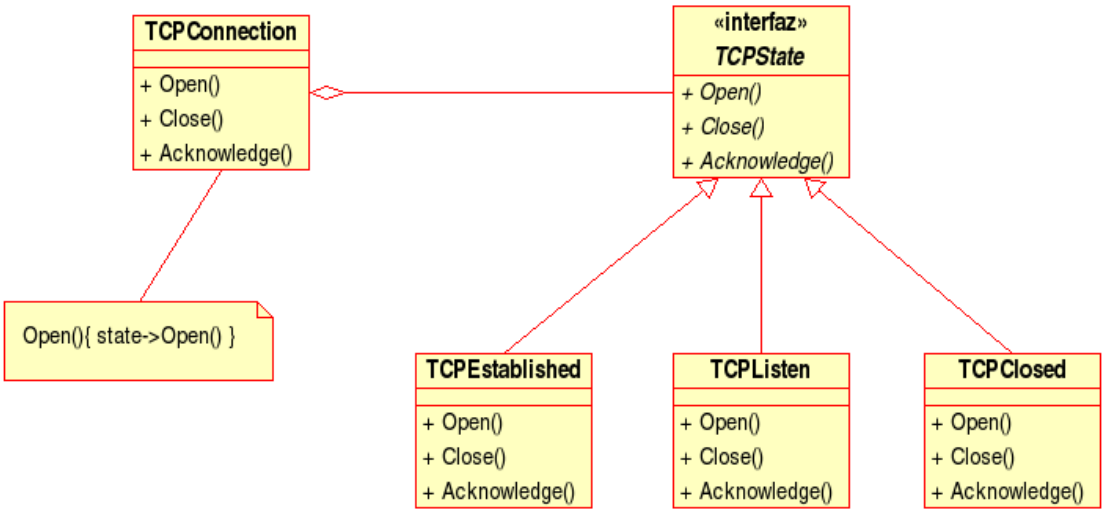


Figura 3.4: *Patrón de diseño* state

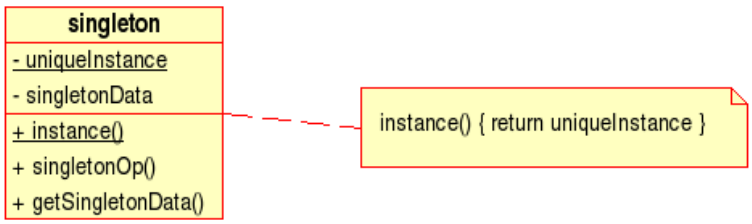


Figura 3.5: *Patrón de diseño* singleton

`Autoconf`, nos permite automatizar la configuración del proyecto, detectando los requisitos (librerías, cabeceras y otros programas) sobre el sistema en el que se pretende construir el proyecto (compilación generalmente). Para ello generaremos el fichero de macros `configure.in`, en el que se indican los requisitos del sistema. Posteriormente, chequearemos que los requisitos se cumplen ejecutando el script `configure`. Si el sistema sobre el que ejecutamos dicho script, cumple los requisitos, se generará la configuración del proyecto de manera automática, y podremos proceder a la construcción del mismo.

`Automake`, permite generar reglas de construcción (*Makefiles*) de manera automática, a partir de macros. Estas macros se indican en el fichero `Makafile.am`. `Automake` será invocado de manera automática por el script `configure`, para generar las reglas de construcción del proyecto.

Así, usando estas herramientas, la construcción del proyecto es tan simple como ejecutar lo siguiente:

```
$> ./configure
checking for a BSD-compatible install... /usr/bin/install -c
checking whether build environment is sane... yes
checking for gawk... gawk
checking whether make sets $(MAKE)... yes
checking build system type... i686-pc-linux-gnu
checking host system type... i686-pc-linux-gnu
checking for style of include used by make... GNU
checking for gcc... gcc
checking for C compiler default output file name... a.out
```

```
checking whether the C compiler works... yes
checking whether we are cross compiling... no
checking for suffix of executables...
checking for suffix of object files... o
....
....
$> make
$> make install
```

El uso de `GNU Autotools`, además nos permite la construcción condicional, en función del tipo de sistema operativo, o de la versión de determinada librería, permitiendo así construir proyectos portables de manera sencilla.

Capítulo 4

Descripción Informática

Una vez presentados los conceptos, precedentes y lineas base en las que se enmarca este proyecto, pasaremos a describir todo el proceso de desarrollo del sistema *jde+*. Comenzaremos con una descripción de alto nivel, donde veremos las ideas fundamentales y las partes básicas, que forman la génesis del sistema *jde+*. Seguiremos con un descripción de bajo nivel, donde desarrollaremos los detalles del diseño y todas las decisiones que los sustentan. En la siguiente sección, describiremos la implementación realizada de *jde+*, mostrando de nuevo todos los detalles relevantes, y los mecanismos de construcción (compilación) empleados para cada parte del sistema.

Como ya se comentó en el capítulo 2 es un objetivo del presente proyecto obtener un diseño basado en la metodología de orientación a objetos, por ello, en todo momento hablaremos de clases y objetos como las partes mínimas del sistema, y nos centraremos en describir sus interacciones. Para esto, usaremos el lenguaje de modelado UML, con el que describiremos el sistema *jde+* gráficamente, mediante diagramas de clases, diagramas de estado o incluso diagramas de secuencia.

4.1 Análisis

En esta sección desarrollaremos una descripción de alto nivel del sistema, donde mostraremos las ideas originales que nos conducen al diseño que mostraremos más adelante. Así, analizaremos el cometido y el porqué de cada parte, de lo que denominamos génesis o núcleo del sistema.

Para iniciar el análisis, recopilaremos las ideas comentadas sobre JDE y sobre *jde+*. Sobre JDE, vimos que describía una arquitectura de control robótica, en la que se hablaba del esquema como la parte mínima y de jerarquía como la manera en que estos se organizaban. Sobre *jde+*, indicamos la idea de tratar a los esquemas como entes compactos y autocontenidos que utilicen los mecanismos del sistema para interaccionar entre ellos. Con estas ideas intentaremos obtener las partes relevantes que debería incluir el sistema, siempre desde un punto de vista muy abstracto, sin entrar en detalles relacionados con la implementación. A estas partes las denominaremos clases de análisis, de las que describiremos su funcionalidad en el sistema, y sus relaciones con las demás. Una vez hecho esto, habremos comprendido mejor el sistema que pretendemos desarrollar, y podremos entrar en más detalles, ya en la sección de diseño.

Según estas ideas el sistema debe constar de algo que represente a un esquema, de algo que represente la relación entre ellos o jerarquía, y algo que represente el mecanismo de comunicación. Así, definimos las siguientes clases de análisis[1] que forman el núcleo del sistema:

[1]Definimos los nombres en inglés, ya que posteriormente la implementación se hará en inglés para facilitar su acceso a los no hispano-hablantes

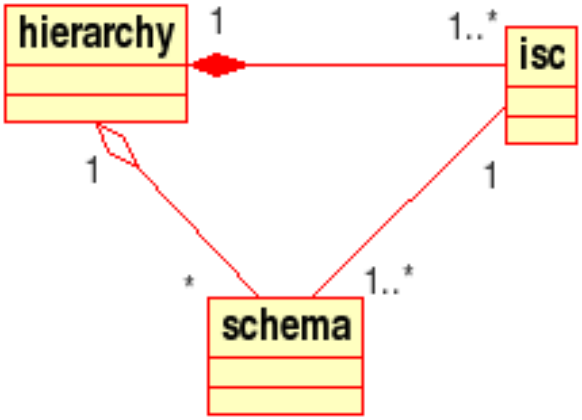


Figura 4.1: *Diagrama de clases de análisis*

hierarchy : Representa la jerarquía, contiene los esquemas y gestiona la relación entre ellos. Permite extraer información de la jerarquía, como quién es el hijo de tal esquema, o quienes los hermanos. También ofrece las capacidades de carga dinámica y reutilización de esquemas.

schema : Representa al esquema, y ofrece las funcionalidades descritas en JDE, un flujo iterativo controlable (activable/desactivable), un estado, capacidades para manejar percepciones o modulaciones, los mecanismos de selección de acción, más la capacidad de interactuar con el resto de esquemas mediante paso de mensajes.

isc : Representa al comunicador (**i**nter-**s**chema **c**ommunicator). Forma parte de la funcionalidad ofrecida por la clase `hierarchy`, aunque por la entidad de los mecanismos que ofrece, se declara como una clase propia. Su funcionalidad principal es la gestión de la comunicación entre esquemas, recogiendo y entregando los mensajes que éstos envíen. También, como funcionalidad secundaría, permite el registro de comunicaciones con fines de depuración en el desarrollo de nuevos comportamientos.

Presentadas las clases, definimos sus relaciones en el diagrama de clases de la figura 4.1. La clase `hierarchy` es la clase principal, y se relaciona con la clase `isc` mediante composición, ya que ésta forma parte de `hierarchy`, aunque por su complejidad se presenta como una clase aparte. También se relaciona con la clase `schema`, esta vez mediante agregación, mostrando a la clase `hierarchy` como un contenedor de esquemas. La relación entre `schema` e `isc`, indica una relación de colaboración entre ambas clases, dado el uso que una hace de la otra (envío y entrega de mensajes).

La cardinalidad de la relación `hierarchy`→`isc` es de uno a infinito, pudiendo existir múltiples comunicadores en el sistema. En cuanto a la relación `hierarchy`→`schema`, la cardinalidad es de cero a infinito, ya que partimos con un sistema sin esquemas, vacío, que crece de manera dinámica, teóricamente sin límites. Las relaciones inversas, en ambos casos, tienen cardinalidad uno, ya que sólo existe una jerarquía en el sistema en cada instante. En cuanto a la relación `schema`→`isc`, la cardinalidad es uno, indicando que un esquema usa un único comunicador en el sistema. Y por último, en la relación `isc`→`schema` la cardinalidad es de uno a infinito, ya que el comunicador se relaciona con un conjunto de esquemas.

Mostradas las clases y sus relaciones, pasamos a una descripción detallada de cada clase, con todas las ideas que reflejan cada una de ellas.

Clase de análisis hierarchy

La clase de análisis `hierarchy` es el centro del sistema, ya que alberga al resto de componentes, coordinando sus acciones. Alberga a los comunicadores y a los esquemas, y gestiona la información acerca de la jerarquía. Es la clase encargada de desencadenar toda la funcionalidad del sistema, simplemente indicando el comportamiento que se desea. Con esto, se activará el primer esquema y resolviendo las relaciones implícitas entre padres e hijos se irán desplegando las distintas jerarquías según la situación.

La idea de tratar al esquema como un ente compacto y autocontenido que se relaciona con otros a través del sistema (enfoque de paso de mensajes), implica que un esquema no interaccione con otro del modo habitual, es decir, que no invoque los métodos de otro para comunicarse con él (enfoque funcional clásico). Esto introduce la necesidad de un mecanismo de nombrado, para que un esquema pueda referirse a otro de alguna manera. Para ello, se usará lo que hemos denominado SID de **S**chema ID, que la clase `hierarchy` se encarga de generar para cada esquema que contenga.

Las funcionalidades que incluye esta clase son:

- Identificación de esquemas mediante SID, para poder referirnos a un esquema determinado de una manera estandarizada dentro del sistema.
- Extracción de información sobre la jerarquía de esquemas en un instante determinado. Esto nos permite obtener información acerca de los "familiares" de un esquema (padre, hijos, hermanos), como sus estados o sus SID's.
- Carga dinámica de esquemas, para permitir la inclusión de nuevos esquemas en el sistema sin necesidad de recompilarlo o retocarlo en modo alguno.
- Resuelve las dependencias de los nuevos esquemas, cargando bajo demanda los esquemas requeridos.
- Reutilización de esquemas en la jerarquía, permitiendo que esquemas en estado inactivo suplan la necesidad de crear nuevos esquemas con las mismas características. Esto permite el ahorro de recursos en jerarquías grandes con esquemas duplicados en su interior.

En cuanto a las colaboraciones, la clase `hierarchy` interactúa con `schema` para extraer su información relevante y para gestionarlo dentro del sistema, y con `isc` para iniciar, configurar y parar los comunicadores.

Clase de análisis schema

Como ya hemos comentado, la clase de análisis `schema` reune toda la funcionalidad esperada de un esquema, tal y como se describe en JDE, más algunos mecanismos fijados como objetivos del sistema *jde+*. Debemos recordar que hay dos tipos de esquemas, los esquemas perceptivos y los motores, aunque de momento no nos fijaremos en este detalle, y hablaremos de lo general de un esquema.

La clase `schema` será la base de los esquemas programados por el usuario, por ello debe ofrecer la funcionalidad general de un esquema, con el objetivo de simplificar la tarea de crear un nuevo esquema. La creación de un nuevo esquema consistirá en usar esta clase como base para heredar sus mecanismos, e implementar tan solo las partes características (iteración, arbitraje, precondiciones,...). Así, apoyandonos en los mecanismos de herencia los nuevos esquemas programados por el usuario obtendrán las capacidades del sistema de manera automática.

Detallando, la clase `schema` incluye las siguientes funcionalidades generales:

- Un flujo iterativo controlable, que ejecute periódicamente la funcionalidad particular del esquema en cuestión. Dicho flujo debe ser controlable, permitiendo ser activado y desactivado.
- Un estado, que marque las acciones a realizar en cada instante. Los estados posibles, ya comentados, son DORMIDO, ALERTA, PREPARADO y ACTIVO.
- Manejo de los datos de un esquema, percepciones y modulaciones, para facilitar al usuario las tareas de acceso a los mismos.
- Cumple un objetivo.
- Describe las dependencias de otros esquemas (posibles esquemas hijos).

Además de estos mecanismos, debemos tener en cuenta que debe ser posible crear múltiples instancias de un mismo esquema, por lo que cada instancia de la clase `schema` mantendrá un estado propio.

En cuanto a las colaboraciones con el resto de clases del sistema, `schema` interactúa con `hierarchy` para obtener información de los esquemas cercanos (hermanos,hijos y padre) y para saber qué comunicador debe usar. También colabora con `isc`, para comunicarse con el resto de esquemas del sistema, enviando mensajes a su padre e hijos.

Clase de análisis isc

La clase de análisis `isc` engloba la funcionalidad de lo que hemos denominado comunicador, esto es, la parte del sistema que se encarga de la comunicaciones entre esquemas, gestionando el mecanismos de paso de mensajes. Por la entidad de sus funciones, `isc` forma una clase, aunque sus instancias residen en el interior de la clase `hierarchy`, como miembros de ésta.

Las funcionalidades que incluye esta clase son:

- Envío de mensajes entre esquemas indicando el origen y el destino mediante identificadores SID.
- Flujo independiente para el procesado de mensajes, que desacopla el envío de la recepción.
- Registro de comunicaciones, para facilitar las tareas de depuración de comportamientos en desarrollo.

Las colaboraciones de esta clase se hacen con `schema` en el envío y entrega de los mensajes, y con `hierarchy` para la localización de los esquemas conocido su SID. La cardinalidad de la relación `hierarchy`→`isc`, comentada más arriba, es uno a infinito, lo que implica que el sistema puede tener más de un comunicador. El uso de múltiples comunicadores podría ser interesante en jerarquías muy grandes, donde podríamos diferenciar varios grupos de comunicación, cada uno con su `isc`, aunque por simplicidad sólo se usará uno de aquí en adelante, y no se explorará esta posibilidad.

4.2 Diseño

Una vez mostrada una descripción de muy alto nivel del sistema desde un punto de vista analítico, pasamos a enfocar los aspectos mencionados bajando a un punto de vista de diseño, donde describiremos con gran detalle cada punto relevante del sistema. Además, se comentarán las decisiones tomadas y los motivos que haya detrás.

En la sección anterior hemos enunciado las partes de las que se compone el sistema *jde+*, y aunque lo hemos hecho desde un nivel de abstracción alto, se puede ver claramente la estructura del sistema. Como hemos visto, todo gira en torno a la clase `hierarchy`, que contiene objetos de clase `schema` e `isc`. Hemos comentado la funcionalidad que debe incorporar cada uno, y cómo se relacionan entre sí para conseguir la funcionalidad que esperamos. Ahora, usando este análisis, vamos a traducir todo esto a algo, que posteriormente podamos implementar directamente. Para ello, cogeremos las clases de análisis, y manteniendo la funcionalidad que se espera de ellas, obtendremos las clases de diseño, que incluirán muchos más detalles y estarán más cerca del lenguaje de implementación. Al hacer esta traducción, llamada realización en diseño por la ingeniería del software, es posible que una clase de análisis, generalmente muy abstracta, se realice en varias clases de diseño, ahora mucho más especificas. Esto es debido, a que es preferible repartir la complejidad en varias clases, que no tener una super clase muy compleja, que aglutine todos los conceptos recogidos en una clase de análisis. Este proceso de traducción, o proceso de diseño, consiste en obtener las clases de diseño, describir su funcionalidad, definir sus miembros (atributos y métodos de clase) y mostrar las relaciones existentes entre las clases, todo ello, con el suficiente detalle como para que el paso a la implementación sea una mera transcripción (al menos en el aspecto declarativo). Es destacable que el proceso de diseño está influido por el lenguaje de implementación que se ha elegido, en nuestro caso C++, y que algunas decisiones tendrán en cuenta esto.

Para describir todo el proceso de diseño que se ha realizado, comenzaremos con un repaso rápido de todas las clases que forman el sistema, comentando de forma básica su funcionalidad y relación con las demás. Se tratará de guiar este repaso por las funcionalidades que se esperan del sistema para introducir cada clase dentro de su contexto. Una vez visto el sistema completo, se profundizará en los aspectos más importantes, mostrando las soluciones planteadas y las decisiones tomadas. Los patrones de diseño utilizados se han descrito en la sección 3.5, por lo que tan solo incluiremos referencias a ellos y su aplicación al diseño concreto de *jde+*.

4.2.1 Diseño en un vistazo

En este apartado enumeraremos las clases de diseño que componen el sistema, intentando seguir las funcionalidades que ofrecen. No debemos olvidar que ya hemos esbozado el sistema en el análisis, por lo que debemos seguir esta línea en el proceso de diseño.

Según el análisis, incorporábamos la funcionalidad de un esquema JDE en la clase `schema`. En diseño desglosamos esta en varias clases de diseño. Las principales son `schemainstance` y `schemaimplementation`, que representan *lo general* y *lo particular* de un esquema respectivamente. Con *lo general* nos referimos a toda la funcionalidad común que incluyen todos los esquemas: un flujo iterativo, un estado, un identificador y una serie de relaciones con otros esquemas (hijos y padre). Con *lo particular* nos referimos a las partes específicas de cada esquema: el código de su iteración, sus precondiciones concretas, el manejo de conflictos de control (arbitraje), y en definitiva aquello que el programador de aplicaciones robóticas debe desarrollar para cada esquema de un comportamiento. Con esta separación hacemos que la clase `hierarchy`, ahora en diseño, interactúe con *lo general* (`schemainstance`) para las tareas de gestión de esquemas, sin preocuparse de lo que haya por debajo. Es la clase `schemainstance` la que actúa sobre *lo particular* cuando sea necesario invocar la funcionalidad de un esquema (ejecutar una iteración, arbitrar un conflicto,...). Así, para crear un nuevo esquema, crearemos una nueva clase que herede de la clase `schemaimplementation` y que implemente una serie de métodos abstractos de esta con la funcionalidad es-

pecífica del nuevo esquema. De aquí en adelante nos referiremos a la parte específica de un esquema como su objeto `schemaimplementation`, aunque en realidad el objeto pertenecerá a una clase derivada de `schemaimplementation` que definirá sus métodos abstractos. También usaremos los términos "parte específica" o "lo particular".

En el análisis veíamos que `hierarchy` aportaba los mecanismos de carga de esquemas, ahora en el diseño esta funcionalidad se reparte en una serie de clases. La carga de esquemas, ya hemos mencionado que se debe hacer sin retocar el sistema, para ello se usará el mecanismo de carga dinámica de código descrito en la sección 3.2. Añadir esta funcionalidad al sistema nos permite cargar en tiempo de ejecución partes no incluidas a priori, justo lo que perseguimos con los esquemas. Pero debemos tener en cuenta que la carga de un esquema implica además resolver sus dependencias, es decir, cargar los esquemas hijos que se requieren para alcanzar el objetivo. Y para esto, debemos poder obtener las dependencias de un esquema y poder distinguir que esquemas las realizan. Para resolver esto aparece la clase de diseño `schemainterface`.

La clase de diseño `schemainterface` define *lo que hace* un esquema y *cómo usarlo*. La funcionalidad de un esquema se definirá de manera abstracta, mediante un nombre que represente dicha funcionalidad. El cómo usar un esquema se refiere a qué parámetros de modulación acepta y qué percepciones es capaz de generar. Así, para describir la *funcionalidad del esquema* "E" que denominamos "X", modulada con los parámetros "M" y/o que genera las percepciones "P" usaremos el *interfaz* "X(M,P)". Entonces diremos que el esquema "E" *implementa el interfaz* "X(M,P)". Pero además de definir la funcionalidad o capacidad de un esquema, podemos usar los *interfaces* para definir sus *dependencias*. De la misma manera, si el esquema "E" requiere la funcionalidad "Y(M',P')", diremos que "E" *usa el interfaz* "Y(M',P')". Así, para definir las dependencias de un esquema también usaremos *interfaces*. Para referirnos informalmente a los *interfaces*, en vez de usar la notación mostrada, usaremos un nombre característico generalmente asociado a su funcionalidad. Por ejemplo, si el *interfaz* describe un sensor láser, usaremos el nombre "láser", o si describe un control en velocidad, usaremos "control_velocidad". Este nombre informal lo denominaremos apodo, y en la instanciación de un esquema lo podremos indicar para posteriormente referirnos a él. Esto será muy útil en caso de que un esquema requiera dos hijos idénticos, representados por el mismo interfaz. Para poder distinguirlos asociaremos un apodo único a cada interfaz.

El uso de los *interfaces* para referirnos de manera genérica a una funcionalidad, oculta la implementación subyacente, característica que nos permitirá crear diferentes implementaciones bajo un mismo *interfaz*.

Retomando el mecanismo de carga de esquemas, nos planteabamos como indicar la capacidad y los requisitos de un esquema, para que el sistema fuese capaz de resolver en tiempo de ejecución que esquemas asociar. La solución planteada es usar *interfaces*, que permiten indicar la capacidad de un esquema y las capacidades requeridas para cumplir su objetivo. Por otro lado, el problema de la carga de esquemas consiste en obtener de entre todos los esquemas disponibles, aquel que implementa la capacidad que se requiere. Para ello, inicialmente se requiere una carga de las capacidades de todos los esquemas disponibles y de la forma de crearlos, y posteriormente mediante algún mecanismo de busqueda localizar el que se necesita y crearlo. Para abordar este escenario, aplicamos el patrón de diseño *factoría abstracta*[14] descrito en el apartado 3.5.1. La carga de factorías se realiza con el método de carga dinámica comentado en la sección 3.2. Dicha funcionalidad se incorpora en una nueva clase, la clase `loader`. Esta clase es genérica, y tiene la propiedad de cargar cualquier clase que se le indique en su parámetro genérico. Para nuestro caso su parametro genérico será la clase `schemafactory` (lo indicaremos con `loader<schemafactory>`), por lo que podrá cargar cualquier objeto derivado de esta clase. Resumiendo, la clase `hierarchy` tiene un cargador de factorías, representado por la clase `loader<schemafactory>`. Al inicio del

sistema, se registran las factorías de los esquemas disponibles, mediante las cuales se extraen todas las capacidades (los *interfaces* implementados). Y con ello, el sistema es capaz de localizar los esquemas que se requieren, e instanciarlos. Hablaremos con más detalle de estos mecanismos posteriormente.

Nos resta hablar de como resolver la comunicación entre esquemas. Para ésta tarea ya hablamos de la clase de análisis `isc` encargada del mecanismo de paso de mensajes. Al diseño pasa con el mismo nombre y realiza las mismas tareas ya comentadas. Como indicamos en los requisitos (sección 2.5) existen cuatro tipos de mensajes, envío de percepción, envío de modulación, cambio de estado y petición de arbitraje. Los dos primeros incluyen datos, que los esquemas generan y por tanto de tipo desconocido a priori. La solución más simple para abordar este problema, es usar un buffer de envío en bruto y que los extremos se encarguen del *marshalling* y el *unmarshalling* de los datos. En nuestro caso, la solución es un poco más elaborada, y se incluye en la clase de diseño `data`. Esta clase contiene un buffer para albergar los datos de tamaño indicado por el usuario, un identificador de tipo, que permite el chequeo en las asignaciones, y una serie de métodos para su manejo. Así, un objeto `data` puede albergar casi cualquier tipo de datos, y usarse como se haría con una variable normal, asignarse a otras, pasarse como parámetro, ..., todo ello con un chequeo de tipos que nos ayudará a detectar situaciones erróneas.

Pero, ¿dónde residen estos datos?, ¿cómo sabemos qué datos podemos enviar a un esquema, o cuáles podemos recibir?. Desde el punto de vista del programador del esquema, la solución es simple, estudia los esquemas que va a usar y determina cuales son sus modulaciones y percepciones. Pero desde el punto de vista del sistema, se deben proporcionar mecanismos genéricos, que sean capaces de interoperar con cualquier esquema sin conocimiento previo para mantener la integridad en las comunicaciones entre esquemas. Es decir, el sistema debe obtener cierta información sobre *como usar* un esquema de manera introspectiva, para averiguar si un esquema actúa sobre otro esquema (a través del `isc` con envío de mensajes) correctamente o notificar el intento de uso erroneo. En cuanto a *como usar* un esquema, nos referimos a que parámetros de modulación acepta, y que percepciones es capaz de generar. Debido a que el lenguaje que se usará en la implementación, C++, no tiene mecanismos de introspección, que nos permitan descubrir los miembros de un determinado objeto en tiempo de ejecución con los que el sistema podría "interrogar" a un esquema en busca de sus modulaciones y percepciones. Para abordar este problema, debemos incorporarlos de alguna manera. Como dejamos indicado cuando hablábamos de la clase `schemainterface`, esta también definiría *como usar* un esquema. Pues bien es aquí donde incorporaremos los mecanismos de introspección, que consistirán en un par de funciones (programadas por el desarrollador de un interfaz) que devuelvan los nombres de las modulaciones y las percepciones de dicho `schemainterface` y por tanto del esquema que lo use. Además las estructuras de datos para dichos parámetros también residirán en el `schemainterface`. Con esto, el esquema que lo use, automáticamente incorpora las estructuras de datos para albergar sus modulaciones y percepciones (su copia). E incluso, al indicar los requisitos de un esquema con `schemainterface`'s, resulta que se incluyen también las estructuras de datos para albergar los parámetros de los esquemas usados o hijos.

La figura 4.2 muestra dos esquemas, EM1 y EP1 (uno motor y otro perceptivo). EP1 implementa la capacidad del interfaz I_B, que ampliado en la imagen muestra varias modulaciones y una percepción. EM1 por su parte, incluye el interfaz I_B como requisito, ya que usa al esquema hijo EP1 para conseguir su objetivo. EM1 ha decidido modular a su hijo EP1, enviando un nuevo valor para la modulación M_1. El sistema puede chequear que EP1 tiene esta modulación gracias a las funciones de introspección del interfaz, y simplemente copia el dato en la estructura reservada al efecto.

Presentado el diseño del sistema en un vistazo rápido, con los principales puntos,

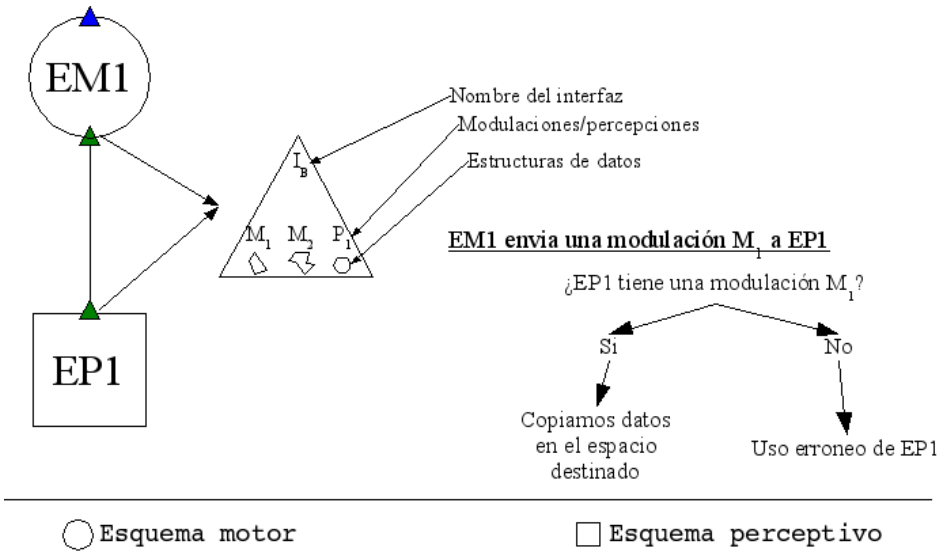


Figura 4.2: *Uso de* `schemainterface`

mostramos en la figura 4.3 el diagrama de clases completo. Hay que indicar que aunque algunas clases aun no han sido presentadas, el diagrama nos ayudará a guiar el resto de la sección. Llegado el momento se presentarán dichas clases. En la figura 4.3 hemos seleccionado tres zonas, cada una de ellas representa la realización en diseño de las clases presentadas en el análisis. Como se puede ver, salvo la clase `isc`, las demás han distribuido su funcionalidad en varias clases de diseño. La clase de diseño `schemainstance` queda entre dos regiones, debido a que parte de su funcionalidad pertenece a una región y parte a la otra como veremos luego.

4.2.2 Desarrollo de un esquema

Uno de los objetivos principales que persigue el desarrollo de *jde+* es simplificar al máximo el desarrollo de nuevos esquemas (ver O-1 en la sección 2.4). En este apartado detallamos las soluciones aportadas en el diseño del sistema con más detalle de lo expuesto en el apartado anterior. Hablaremos de las clases involucradas y de su relación para la consecución de este subobjetivo.

Como hemos presentado en el apartado anterior la creación de un nuevo esquema consiste en heredar de la clase abstracta `schemaimplementation`, definiendo los métodos abstractos con la funcionalidad del nuevo esquema, también debemos desarrollar el interfaz (una clase que herede de `schemainterface`) que presenta su capacidad y modo de uso. Para entrar en detalles, la figura 4.4 muestra las clases de diseño que intervienen y sus miembros más relevantes. Los miembros aparecen con diferentes tipografías y símbolos su significado es el siguiente:

Tipografía cursiva : El miembro mostrado así es un método abstracto, es decir, no implementa funcionalidad, si no que declara un método que las clases derivada deberán implementar. Una clase con métodos abstractos se denomina clase abstracta y no se permite crear objetos de clases abstractas, siendo su finalidad ser "moldes" para derivar clases que comparten su estructura.

Miembro precedidos de + : El miembro es público, y todo el mundo puede usarlo.

Miembro precedidos de # : El miembro es protegido, lo que implica que sólo la propia clase y las clases derivada podrán usarlo.

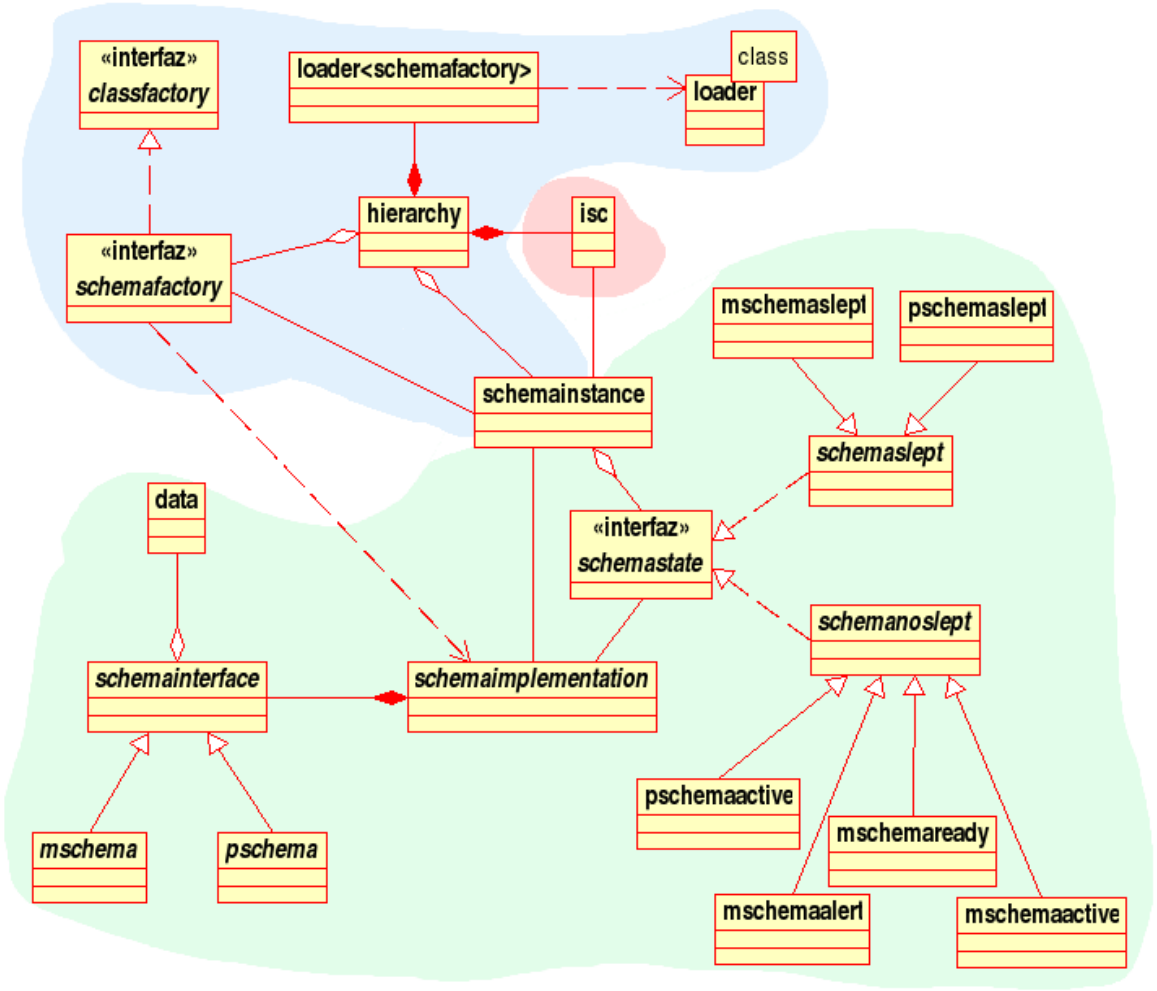


Figura 4.3: *Diagrama de clases de diseño*

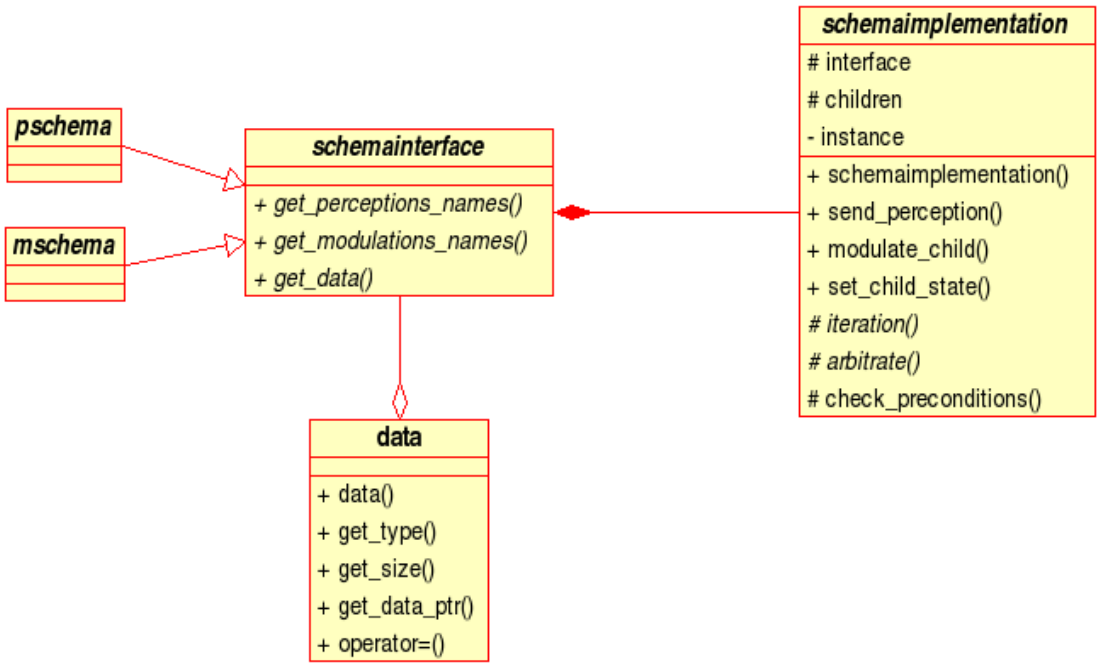


Figura 4.4: *Clases relacionadas con el desarrollo de un esquema*

Miembro precedidos de - : El miembro es privado, y sólo la clase que lo contiene puede usarlo.

De aquí en adelante se usará esta notación, y si fuese necesario se introducirán nuevas indicaciones.

Como podemos ver en la figura 4.4, la clase `schemaimplementation` contiene varios métodos destinados a facilitar el uso del sistema, permitiendo el envío de mensajes a los esquemas cercanos (padre e hijos) de manera simple. Estos métodos son `send_perception`, `modulate_child` y `set_child_state`, con los que podemos enviar una percepción al padre, modular un hijo o cambiar el estado de un hijo, respectivamente.

En cuanto a la funcionalidad específica de un esquema, tenemos los métodos abstractos `iteration` y `arbitrate` que un esquema debe implementar con la funcionalidad de la iteración y la resolución de un conflicto de control. El método `check_preconditions` se puede redefinir con el chequeo de precondiciones del esquema en cuestión. Y con estos tres métodos, se cubren las funcionalidades particulares de un esquema, el resto *lo implementa el sistema*.

Pero como ya hemos comentado (y como se puede ver en la figura 4.4), un esquema debe mostrar su capacidad y sus requisitos mediante interfaces usando la clase `schemainterface`. Otra tarea a desempeñar en el desarrollo de nuevos esquemas, es el desarrollo de sus interfaces, a menos que estemos reutilizando uno ya existente. Crear un nuevo interfaz es simplemente crear una nueva clase que herede de `schemainterface` e implemente sus métodos abstractos. Dichos métodos son los mecanismos de introspección que el sistema usará para analizar un esquema. También será necesario incluir las estructuras de datos que alberguen las modulaciones y percepciones de un esquema e implementar el método abstracto `get_data`, que sea capaz de transformar estas estructuras en objetos `data`.

Opcionalmente puede ser interesante implementar clases recubrimiento (*wrappers*) para los datos que manejará el esquema, para facilitar el uso de los objetos data, demasiado genéricos y un tanto difíciles de manejar en crudo. Así, podemos crear clases que faciliten el acceso, la asignación y demás acciones que un esquema pueda

realizar sobre sus datos, y automaticen la tarea de transformación de y hacia un objeto `data`.

4.2.3 Carga dinámica de esquemas

El mecanismo de carga dinámica también es digno de ser comentado con más detalle. Junto con el desarrollo de esquemas, persigue el subobjetivo O-1 mostrado en la sección 2.4, en este caso en el aspecto de "inserción de esquemas".

Como ya se ha comentado, en el diseño de este mecanismo se usa la carga dinámica de código descrita en la sección 3.2 y el patrón de diseño *factoría abstracta*[14] descrito en el apartado 3.5.1. Desarrollaremos ambos temas, mostrando como se materializan sobre el diseño de *jde+*.

La figura 4.5 muestra las clases que intervienen en el desarrollo de este mecanismo en *jde+*. El patrón de diseño *factoría abstracta* se aplica en nuestro caso con ligeras variaciones. El papel de la clase `hierarchy` es el de contenedor de las factorías de esquemas. Dichas factorías son de la clase `schemafactory`, una derivación de `classfactory` que incluye métodos para obtener información de los objetos que son capaces de instanciar, en su caso objetos `schemaimplementation`. Esta información es precisamente la capacidad y requisitos de un esquema, información que usará `hierarchy` para indexar las factorías y poder resolver las dependencias que se planteen en cada jerarquía.

En cuanto a la funcionalidad de carga dinámica, hemos visto que se encuentra localizada en la clase de diseño `loader`. También hemos comentado que esta clase es genérica y que en nuestro caso se especializa con la clase `schemafactory`. El lenguaje C++ tiene soporte nativo para tipos genéricos a través del uso de clases plantilla (*templates*) que permiten definir clases con tipos como parámetros. Al sustituir los parámetros decimos que instanciamos o especializamos la plantilla. La clase `loader` la expresamos usando una clase plantilla que aceptará un parámetro, la clase a cargar dinámicamente. Lo representaremos con el nombre de la clase plantilla y entre los símbolos <> la clase que actúa como parámetro. En nuestro caso diremos `loader<schemafactory>`.

La clase `loader<schemafactory>` es parte de la clase `hierarchy`, ya que es uno de sus miembros, y esta la usa como cargador de objetos de clase `schemafactory`. Como se puede apreciar en la figura 4.5, la clase loader tiene un método mediante el cual es capaz de obtener un objeto de una clase cargada dinámicamente desde un fichero. En nuestro caso se obtienen objetos de la clase `schemafactory` que `hierarchy` almacena e indexa según su *interfaz* para su uso posterior.

Recapitulando, tenemos que la clase `hierarchy` carga con `loader<schemafactory>` las factorías disponibles en las ubicaciones que se le hayan indicado (típicamente con rutas de directorios en un fichero de configuración) a modo de bibliotecas dinámicas. Tras indicarle que inicie el comportamiento "CX", que se identifica con el *interfaz* "X", se inicia la instanciación de los objetos `schemaimplementation` (lo particular de un esquema), primero del que implementa el interfaz "X", posteriormente de sus dependencias, y de manera recursiva de las dependencias de sus dependencias. Este proceso se detalla en el apartado siguiente. La posibilidad de realizar diferentes implementaciones del mismo *interfaz* requiere que a la hora de cargarlas nos decidamos por una. La política elegida es que la primera cargada es la que prevalece, así bastará incluir las localizaciones de las factorías por orden de preferencia.

4.2.4 Instanciación de un esquema: creación retardada

En esta sección nos interesamos por la instanciación en si de los esquemas, es decir, del momento justo en que aparecen en escena. En el apartado anterior, mostrábamos los mecanismos que permitían la carga dinámica de esquemas en el sistema, es decir, *el*

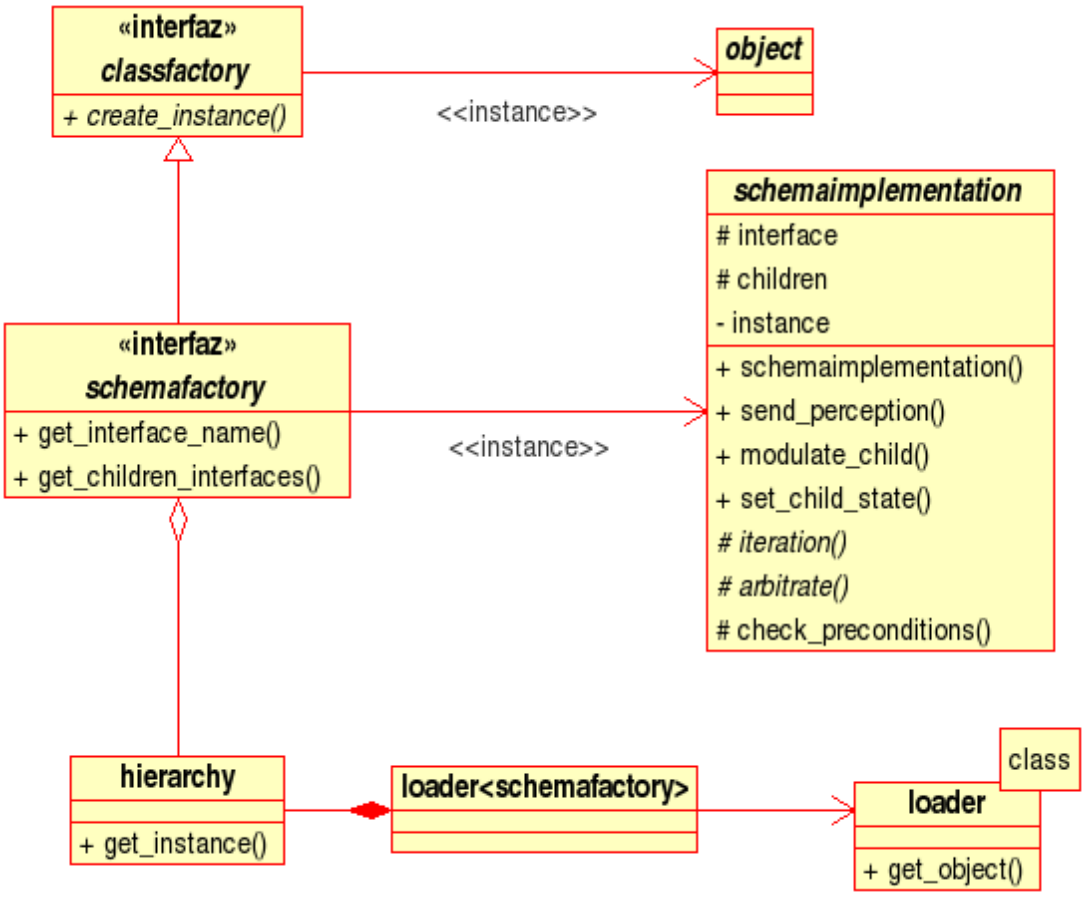


Figura 4.5: *Clases relacionadas con la carga de un esquema*

cómo. Ahora lo que nos interesa es ver *el cuándo* en el proceso de instancia un esquema. Para dirigir la descripción nos planteamos dos preguntas, ¿es necesario instanciar toda la jerarquía que desarrolla un comportamiento?, y si no, ¿cuándo hay que crear una instancia de esquema?.

JDE plantea que sólo aquellos esquemas que han sido activados por su padre consumen recursos del sistema, por lo que la respuesta a la primera pregunta es no. No es necesario crear un esquema antes de que sea activado, lo que implica que no se ha requerido su funcionalidad aun. Por otro lado ya mencionamos la posibilidad de bucles en las dependencias entre esquemas, por lo que todas las posibles jerarquías desarrolladas para un comportamiento, pueden explotar combinatoriamente, bloqueando al sistema en una instanciación infinita de esquemas. Teniendo esto en cuenta, la instanciación de un esquema debe retrasarse hasta que se requiera su funcionalidad. La solución de retrasar la instanciación no resuelve el problema de bucles en jerarquías con dependencias mal creadas, pero si permite jerarquías con bucles que no crezcan infinitamente.

En cuanto a la segunda pregunta que planteamos, según el párrafo anterior la respuesta sería que una instancia de esquema se crea cuando es activada. Aunque en algunas ocasiones puede llegar a crearse antes, por ejemplo si un padre quiere configurar a un hijo antes de activarlo mediante modulación. Así podemos decir que *en jde+ la creación de un esquema se retarda hasta su activación o modulación.*

Con estos planteamientos, el sistema *jde+* ira creando las instancias a medida que se requieran por sus antecesores (sus padres) y no en un proceso inicial de instanciación que intente desarrollar todas las posibles dependencias de forma recursiva. Además de evitar la explosión combinatoria mencionada, el sistema estará menos cargado, puesto que sólo se instanciarán los esquemas que por las circunstancias hayan sido requeridos. Este comportamiento es muy similar al usado por el cargador de librerías dinámicas de GNU/Linux cuando funciona en modo perezoso. Cuando se ejecuta un programa enlazado dinámicamente el cargador registra las dependencias de librerías sin cargar ninguna. En el momento en que se accede a algún símbolo enlazado dinámicamente, el cargador carga la librería necesaria. De esta forma sólo se cargan aquellas librerías usadas en la ejecución concreta del programa.

Como ya hemos dicho varias veces antes, una instancia en *jde+* consta de dos partes, *lo general* y *lo particular*. *Lo general* reside en `schemainstance`, que contiene el identificador de una instancia, su estado y el flujo propio de un esquema. *Lo particular* se encuentra en `schemaimplementation`, o mejor dicho, en las clases derivadas de `schemaimplementation` que el desarrollador de esquemas implementa con la funcionalidad especifica de cada esquema. La creación de una instancia de esquema sigue de dos pasos. En el primero `hierarchy` crea el objeto `schemainstance` que representará a la instancia de esquema en el sistema cuando alguien lo activa o modula por primera vez. Este es ligado a la jerarquía indicándole quién es su padre y quiénes sus hijos (en términos de identificadores). Además se le indica el modo de construir su parte especifica, pasándole la factoría que debe usar, para que en el segundo paso, y cuando `schemainstance` crea oportuno instancie su `schemaimplementation`. Este es el motivo por el cual indicamos que la clase `schemainstance` pertenecía a las dos zonas marcadas en la figura 4.3, ya que incorpora parte de la funcionalidad que en análisis colocamos en `hierarchy` (instanciación) y parte de la que colocamos en `schema` (identificador, estado,...). Una vez creada la instancia del esquema, ésta comienza a actuar en busca de su objetivo relacionándose con el resto de la jerarquía.

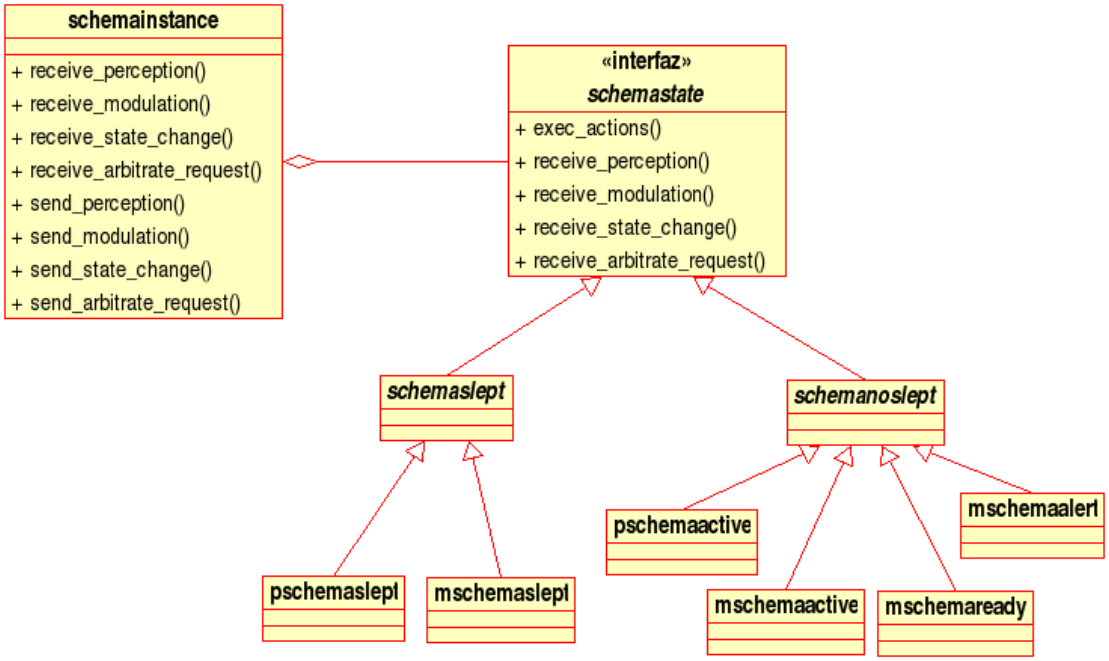


Figura 4.6: *Clases relacionadas con el estado de un esquema*

4.2.5 Estados de un esquema

Las acciones realizadas por un esquema, como vimos en la teoría de JDE (ver sección 2.1), dependen del estado en que se encuentre. Por ejemplo, si un esquema motor se encuentra en estado ACTIVO, éste comprueba sus precondiciones y ejecuta su iteración. Si por el contrario esta en estado ALERTA comprueba sus precondiciones. El estado también influye en como los esquemas reciben los mensajes, por ejemplo ignorando la llegada de nuevas percepciones si se ha pasado al estado DORMIDO. Esto introduce gran cantidad de partes condicionales, que en el momento de la implementación se traduce en un código difícil de entender y depurar.

Para aliviar esto, se aplica el patrón de diseño *state*[14] descrito en el apartado 3.5.2. En nuestro diseño el patrón encaja perfectamente, ya que un esquema realiza diferentes acciones condicionadas por su estado. En la figura 4.6 podemos ver como hemos aplicado el patrón de diseño. La clase `schemastate` es la clase abstracta de la que se derivan todos los posibles estados de un esquema. Tiene cinco métodos, el primero `exec_actions` ejecuta las acciones de un esquema en su iteración y el resto es usado para recibir los diferentes mensajes. En nuestro escenario hemos derivado dos clases que representan los estados dormidos y los despiertos, y bajo estas derivamos los estados concretos. Esto atiende a que algunas acciones son comunes y de esta manera algunos estados comparten dichas acciones. El cambio de estado lo marca el estado actual, de manera que un esquema no debe preocuparse más que de ejecutar el método adecuado en cada instante.

Por último falta indicar que un objeto estado no guarda un estado interno propio, lo que implica que un objeto estado no sabe a que instancia pertenece, debiendo incluir un puntero o referencia a la instancia que lo usa en cada uno de sus métodos. Esto nos permite utilizar un único objeto de cada estado para todas las instancias, evitando el uso de recursos innecesarios. Esto responde a otro patrón de diseño, el patrón *singleton*[14] descrito en el apartado 3.5.3.

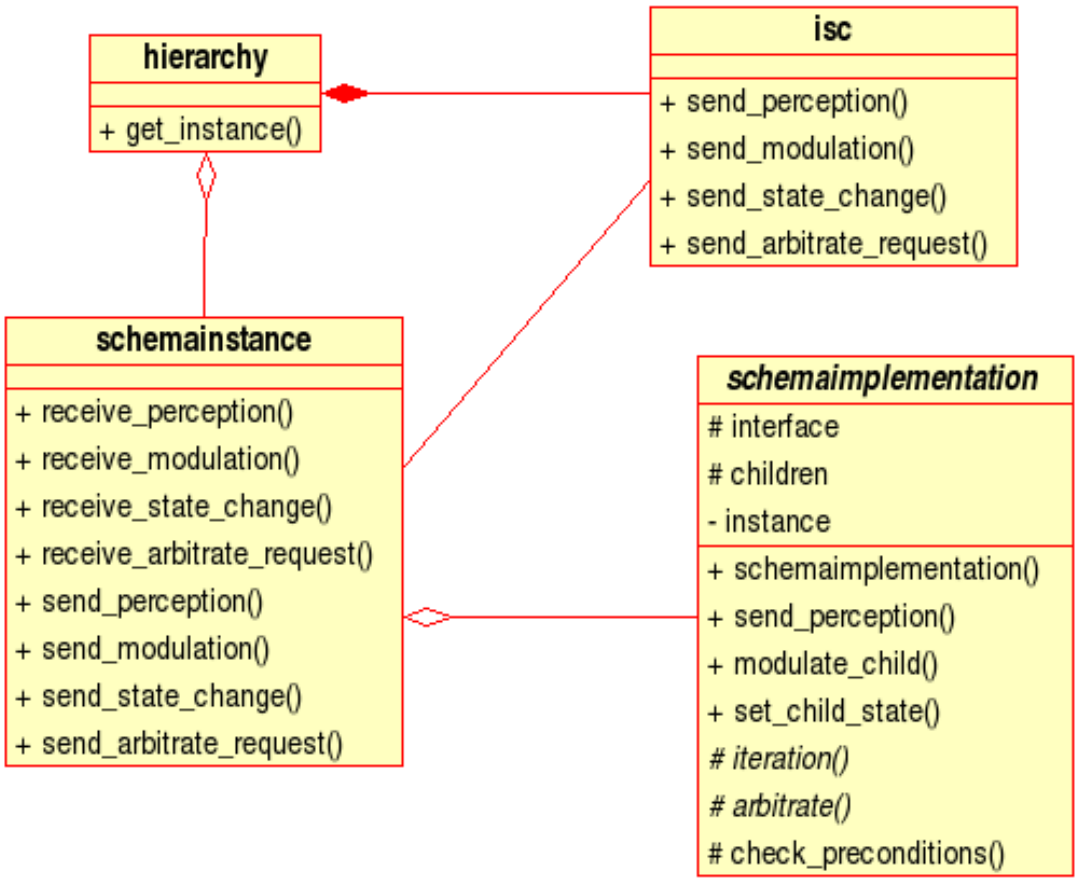


Figura 4.7: *Clases relacionadas con la comunicación entre esquemas*

4.2.6 Comunicación entre esquemas

De la comunicación entre esquemas ya hemos mencionado prácticamente todos los aspectos. En este apartado mostraremos las clases que actúan en este mecanismo y reordenaremos todos los conceptos que han aparecido.

Como se indica en el capítulo de objetivos, la comunicación entre esquemas mediante paso de mensajes es una de las principales ideas incorporadas en el sistema *jde+*. Su introducción hace posible tratar a un esquema como un ente compacto que los utiliza para comunicarse con el resto de esquemas del sistema, permitiendo la incorporación opcional de políticas de comunicación, ejecución de eventos y otros mecanismos.

La base de este sistema de comunicación es la clase `isc`, que recordando toma su nombre de "**i**nter-**s**chema **c**ommunicator". Su misión principal es recoger mensajes y entregarlos. Para ello ofrece una serie de métodos mediante los cuales los esquemas entregan los mensajes que se almacenarán en una cola interna. Posteriormente `isc` los entregará al esquema destino. Es importante remarcar que la recogida y la entrega están totalmente desacopladas, por lo que los flujos del emisor y el receptor nunca se bloquean. La manera de indicar el origen y el destino de un mensaje es a través del identificador del esquema (el SID). El `isc` registra todas las comunicaciones para posteriormente poder analizar las interacciones entre esquemas, herramienta muy útil para la depuración de un comportamiento en desarrollo.

La figura 4.7 muestra las clases que intervienen en el mecanismo de comunicación. El envío de percepciones, modulaciones y cambios de estado se realiza siempre desde la funcionalidad especifica de un esquema, usando para ello los métodos de la clase `schemaimplementation`. Estos a su vez usan a `schemainstance` como intermediario

(*proxy*), que por fin hace la entrega a `isc`. En cuanto a las peticiones de arbitraje, es la propia clase `schemainstance` la que las genera y envía a `isc`, ante un conflicto de control. Todos los mensajes son encolados a la llegada al `isc`, y junto con los datos del mensaje se almacena el origen, el destino y la hora de llegada.

La entrega, por su parte, la realiza `isc`, que incorpora un flujo independiente para esta tarea. Así dicho flujo, desencola los mensajes y los entrega a su destinatario, registrando la transacción. Para la entrega es preciso localizar la instancia, pues lo único que se conoce es el identificador de destino. Para ello se utiliza el método `get_instance` de la clase `hierarchy`, que dado un identificador de esquema, devuelve la instancia asociada. Una vez conocida la instancia se hace uso de uno de sus métodos de entrega (los que empiezan por "receive"), dependiendo del tipo de mensaje a entregar. Estos, como vimos en el apartado 4.2.5, realizarán las tareas apropiadas al estado actual del esquema.

4.2.7 Reutilización de esquemas

Cuando hablamos de la instanciación de un esquema, detallamos que esta sucedía cuando el esquema era activado por su padre. También comentamos que se llevaba a cabo en dos pasos, primero se creaba el objeto `schemainstance` y despues el derivado de `schemaimplementatión`. Pero en ningún momento hablamos de que sucedía cuando un esquema era dormido por su padre. En este apartado comentamos las decisiones tomadas al respecto.

Según JDE, cuando un esquema pasa al estado DORMIDO deja de consumir recursos, y de algún modo se "olvida" su estado. Esta solución es la más sencilla de implementar, pues basta destruir los objetos que representan a un esquema cuando este pasa a estado DORMIDO. Además, esta solución permite mantener un rendimiento temporal óptimo, pues se libera el espacio consumido por un esquema que no esta actuando. Pero esto tiene implicaciones negativas para el rendimiento del sistema, que estaría creando y destruyendo objetos constantemente, sobre todo en jerarquías muy dinámicas.

Para evitar esto *jde+* toma una solución intermedia, que busca minimizar los recursos ocupados manteniendo un buen rendimiento. Para conseguir esto, lo que hemos denominado *lo general* no se destruye hasta la finalización del sistema, es decir, un esquema mantiene su objeto `schemainstance` siempre. Con el se mantiene el identificador del esquema, su flujo independiente, que en estado DORMIDO se bloquea, y su estado (recordemos que el objeto estado es un *singleton*[14] por lo que no ocupa espacio extra). Y por otra parte, lo que hemos denominado `lo particular` se reutiliza dentro de la jerarquía, pudiendo compartir varios esquemas un mismo objeto `schemaimplementation`. Para la reutilización es necesario que un esquema haya sido dormido, momento en que libera su objeto `schemaimplementation` que ingresa en una *lista de reutilizables*. Si se instancia otro esquema del mismo tipo en la jerarquía puede reutilizarlo. Así, si la activación de ambos esquemas no se solapa en el tiempo, un solo objeto `schemaimplementation` es suficiente para las dos instancias.

Esta solución tiene varias implicaciones. La primera es que el uso de memoria puede dispararse en jerarquías muy grandes, ya que ni se libera la memoria de un objeto `schemainstance`, ni la de los `schemaimplementation` (aunque se puedan reutilizar). La solución más drástica sería liberar toda la memoria siempre que dormimos un esquema, pero ya comentamos que esto también podría conllevar problemas. Quizá podríamos plantear otras políticas de reutilización más inteligentes, que inevitablemente consumirían más recursos de computo. Por ello, aunque la solución planteada no es óptima para todos los casos, si es muy buena para la mayoría de casos que trataremos (por el momento la mayor jerarquía construida con jde.c tiene unos 20 esquemas).

La segunda implicación tiene que ver con la reutilización de los objetos `schema-implementation`. JDE indica que un esquema no almacena su estado despues de dormido. El sistema *jde+* tiene el control de un esquema hasta `schemainstance`, a partir de aquí, es el desarrollador el que decide como implementar la funcionalidad particular sobre una clase que deriva de `schemaimplementation`. Por ello, el sistema no puede controlar el estado interno de esta funcionalidad particular, que es dependiente del contexto actual (el padre y la modulación que aplicó). Reutilizar esta funcionalidad particular, y reconectarla en otros contextos puede conllevar funcionamientos erroneos (al menos hasta que sea reconfigurada). Un ejemplo simple sería un esquema perceptivo con un parametro de modulación que indica el número de medidas por mensaje que debe enviar. Si este esquema se reconecta y activa antes de ser reconfigurado, enviará a su nuevo padre un número (quizás) erroneo de medidas, que podría causar un mal funcionamiento en el sistema. Puesto que es un problema que esta fuera del alcance del sistema, la solución se deja en manos del desarrollador del esquema. Esta consiste en añadir el método `reset_handler` a la clase `schemaimplementation`, que el desarrollador puede redefinir con código para reiniciar el estado del esquema. Dicho método es invocado por el sistema cuando realiza una reutilización.

Una consecuencia directa de esta solución, es que un esquema en *jde+* puede mantener sus estado, incluso despues de DORMIDO. Esto como hemos visto no se contempla en JDE, pero puede resultar útil en ciertas circunstancias. Por ejemplo si suponemos que un esquema obtiene parte de su funcionalidad accediendo a un dispositivo del sistema (fichero, puerto serie, ...) puede resultar necesario que mantenga el estado de dicho dispositivo entre activaciones para el correcto funcionamiento del esquema.

En la figura 4.8 vemos las clase implicadas en la reutilización de esquemas. La clase `hierarchy` crea y mantiene todos los objetos `schemainstance`. Esto se realiza siempre en el método `get_instance` que se encarga de crear las instancias cuando son utilizadas por primera vez. La clase `schemainstance` es la que más funcionalidad incorpora para este respecto. Dicha funcionalidad es similar a la que vimos cuando hablamos del patrón de diseño *singleton*[14] en el apartado 3.5.3. Para ello se incorporan a `schamainstance` tres miembros de clase, `reusable_implementations` es un contenedor para objetos `schemaimplementation` reusables, `adquire_implementation` reutiliza un objeto `schemaimplementation` o lo crea si no es posible reutilizar, y `release_implementation` hace un objeto `schemaimplementation` reusable insertandolo en `reusable_implementations`. Así, cuando un esquema se activa se usa `adquire_implementation` para obtener su parte específica (la adecuada para este esquema según su objetivo). En el momento de desactivarlo con `release_implementation` se libera dicha parte específica, que pasa a estar disponible para otras instancias que lo puedan reutilizar. Por último, la clase abstracta `schemaimplementation` incorpora el método `reset_handler`, que se ejecuta automáticamente para reiniciar el estado interno de la parte específica reutilizada.

4.2.8 Concurrencia en el sistema

Un último punto que no debemos pasar por alto en el diseño del sistema es su naturaleza concurrente. Es de vital importancia tener en cuenta en el diseño que algunas de las clases mencionadas serán accedidas de manera concurrente desde varios flujos de ejecución. En este apartado comentaremos las soluciones aplicadas y sus implicaciones.

El sistema descrito hasta ahora tiene numerosos flujos de ejecución independientes. Por un lado esta el flujo principal, que inicia el sistema creando la instancia de `hierarchy`, que a su vez carga las factorías y crea los esquemas. Por otro, tenemos el flujo del `isc`, encargado de enviar mensajes que traduce los identificadore usando a `hierarchy`. Y por último, tenemos el flujo de ejecución de cada uno de los esquemas

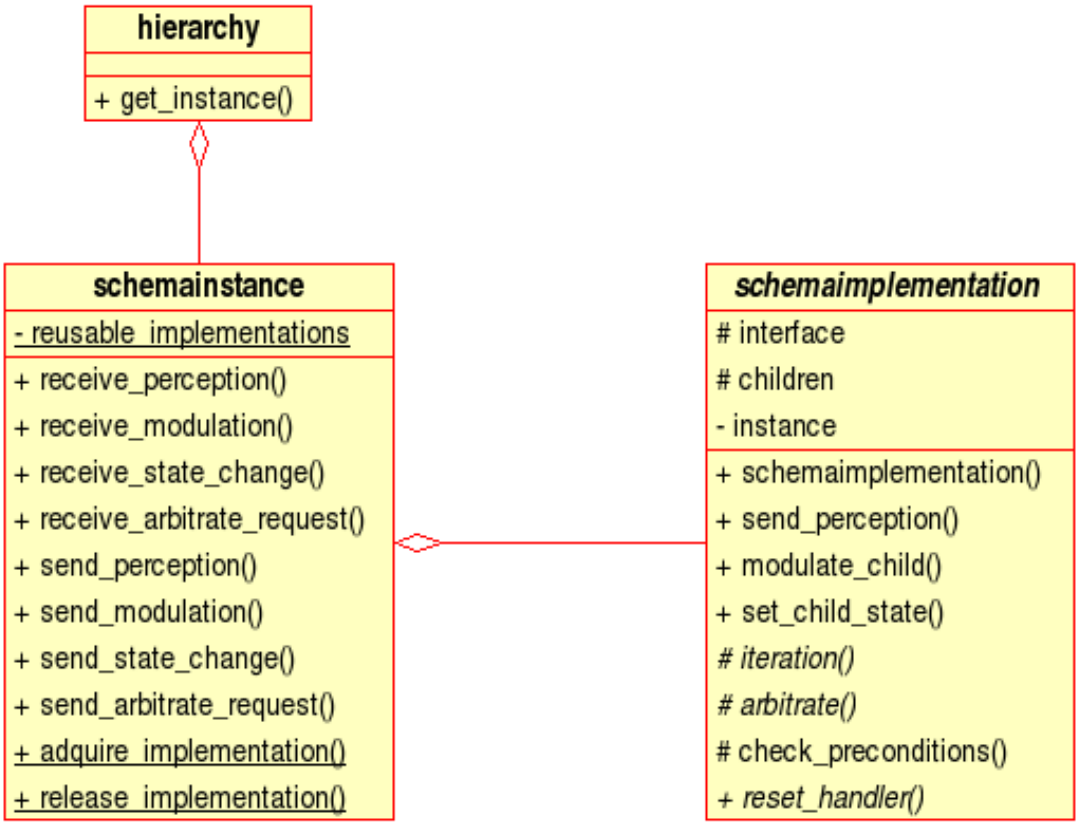


Figura 4.8: *Clases relacionadas con la reutilización de esquemas*

de la jerarquía que creemos, que enviarán mensajes a `isc` y requerirán información a `hierarchy`.

Así, la clase `hierarchy` es accedida desde múltiples flujos, el flujo del `isc` y el de cada esquema creado. Y el `isc` es accedido desde cada esquema. Por esto, las clases `hierarchy` e `isc` deben incorporar mecanismos de sincronización para evitar condiciones de carrera.

Para este caso aplicaremos el patrón de diseño *monitor*[25] descrito en la sección 3.4 a las clases `hierarchy` e `isc`. De este modo, el acceso estará secuenciado a las partes accedidas concurrentemente, eliminando los problemas de condición de carrera.

Evidentemente, esto implica que las clases que se comportan como monitores son "cuellos de botella" del sistema. Por ello, la implementación de los métodos accedidos de manera secuenciada deberá ser muy eficiente para evitar colapsos. En el supuesto de que el sistema se colapse se podría aplicar el otro patrón comentado en la sección 3.4, el *objeto activo*[25] que no secuencia el acceso. Este evitaría el bloqueo en la llamada a sus métodos, pero internamente los secuencia al igual que hace un *monitor*[25]. Esta decisión deberá tomarse tras realizar pruebas de rendimiento en el sistema, una vez implementado.

Existen otros puntos del sistema en los que se deben poner medidas de sincronización por su acceso concurrente. Uno de estos puntos son los objetos derivados de `schemastate`. Como mencionamos en el apartado 4.2.5, aplicamos el patrón de diseño *singleton*[14] para que sólo existiese una instancia de cada objeto en todo el sistema. Dado que el mecanismo del *singleton*[14] es accedido desde múltiples esquemas es necesario secuenciar el acceso al método de clase `instance` que tiene cada una de las clases que representan a un estado. Para este caso basta un simple cerrojo `mutex` que permita el acceso de uno en uno. De la misma manera en el mecanismo

de reutilización de esquemas (como vimos con funcionalidad similar a la de un *singleton*[14]) es necesario secuenciar el acceso a los métodos de la clase `schemainstance` `adquire/release_implementation`. Para este caso también basta un cerrojo `mutex`.

Otro motivo que requiere mecanismos de sincronización, es el bloqueo de flujos de ejecución cuando no tienen trabajo que realizar. Es el caso del flujo del `isc` cuando no tiene mensajes para enviar, o el de un esquema cuando esta DORMIDO. La solución más simple consiste en chequear constantemente la condición que indique la vuelta al trabajo. Esto se denomina "espera activa" y desperdicia muchos ciclos de procesador. La solución óptima es usar las variables condición que presentamos en la sección 3.4, con las que podemos conseguir que un flujo de ejecución se bloquee hasta la llegada de una señal.

4.2.9 Ejecución remota de un esquema

El subobjetivo O-4 pide que se diseño del sistema que de alguna manera permita ubicar esquemas en nodos de ejecución distribuidos. Por ello, el diseño de las partes de un esquema a recibido esta influencia. En este apartado indicamos las decisiones tomadas a este respecto. Esta funcionalidad, como ya se ha comentado, no se implementará aunque se deja la puerta abierta para su introducción.

Ya hemos descrito las partes principales de un esquema, `schemainstance` y `schemaimplementation`. Y también mencionamos que `schemainstance` es un *proxy* para `schemaimplementation`, puesto que esta última se relaciona con el resto del sistema sólo a través suya. Esto queda claro en el diagrama de clases de la figura 4.3.

Pues bien, para poder distribuir los esquemas bastaría intercalar entre ambas clases un intermediario que gestionase la comunicación entre ellas. De esta manera, la parte que incorpora la funcionalidad particular de un esquema (típicamente su parte computacionalmente más costosa) podría colocarse en un nodo remoto y seguir relacionándose con el sistema a través de dicho intermediario.

4.3 Implementación

Hasta el momento, todo lo contado en este capítulo nos describe como el sistema *jde+* realizará tal o cual tarea o como reaccionará ante tal situación. Ahora mostraremos la implementación realizada de *jde+* que materializa todo el diseño desarrollado. Esta sección no pretende ser un listado del código fuente, si no una descripción de las partes más destacadas de la implementación, más los detalles de construcción del sistema y de los esquemas, y el modo de uso del sistema.

4.3.1 Descripción de la implementación

La implementación del sistema *jde+*, como ya se ha comentado, se ha realizado en el lenguaje de programación C++. Siguiendo la metodología del diseño, el paradigma utilizado es la orientación a objetos. En este apartado trazaremos las clases desarrolladas en el diseño con las desarrolladas en la implementación, mostrando las unidades de compilación resultantes. También hablaremos de otros aspectos relevantes relacionados con la implementación realizada.

La unidad de compilación habitual en el lenguaje C++ se compone de un fichero de encabezados, que incluye las declaraciones, y un fichero de código con las definiciones. Diferenciaremos ambos tipos de fichero por su extensión. Un fichero de encabezado usará ".h" y uno de código usará ".cpp". Cada unidad de compilación contendrá una o más clases y nombraremos sus ficheros con el nombre de la clase principal que incluya.

Unidad de compilación	Clases contenidas
classfactory	classfactory
critical_section	mutex, scoped_lock, condition
data	data
hierarchy	hierarchy
isc	isc, isc_msg, isc_perception_msg, isc_modulation_msg, isc_arbitrate_req_msg, isc_state_change_msg
loader	loader, loaderexception
schemafactory	schemafactory
schemaimplementation	schemaimplementation
schemainstance	schemainstance
schemainterface	schemainterface, pschema, mschema
schemastate	schemastate, schemaslept, schemanoslept, pschemaslept, pschemaactive, mschemaslept, mschemaalert, mschemaready, mschemaactive
schematypes	No incluye clases, sólo tipos y funciones compartidas
smart_ptr	counted_ptr
util	No incluye clases, sólo tipos y funciones compartidas

Cuadro 4.1: *Unidades de compilación de* jde+

El sistema *jde+* se compone de 14 unidades de compilación que incluyen un total de 30 clases. En la tabla 4.1 se listan todas la unidades de compilación con las clases que contienen. Las unidades de compilación `critical_section`, `schematypes`, `smart_ptr` y `util` contienen clases auxiliares, tipos de datos y utilidades usadas en la implementación del resto de clases. Las demás unidades de compilación contienen las clases que describimos en el diseño, a excepción de `loader` e `isc` que implementan clases de uso interno.

En C++ la manera de incluir la funcionalidad de una unidad de compilación en otra es a través de la inclusión (usando la macro `include`) de su fichero de encabezados. Cuando el preprocesador de C++ encuentra una inclusión, automáticamente la sustituye por el fichero de encabezados incluido. En ciertas situaciones puede suceder que un fichero de encabezado se incluya más de una vez (dependencias circulares, dependencias indirectas, ...), produciendo declaraciones repetidas que terminan con un error de compilación. Para evitar estas situaciones podemos hacer uso de otras macros del preprocesador que nos garanticen que un fichero sólo se incluirá una vez. Esto se aplica a todos los ficheros de *jde+*, que incluyen lo siguiente:

```
#ifndef NOMBRE_FICHERO_H
#define NOMBRE_FICHERO_H
...
DECLARACIONES
...
#endif
```

Esto hace que al incluir este encabezado por primera vez, se acceda al `ifndef`, ya que aun no esta definido el nombre `NOMBRE_FICHERO_H`. En las siguientes inclusiones no se podrá acceder al `ifndef` y no se incluirá nada del fichero.

Las dependencias entre clases deben tratarse con cuidado, pues a la hora de compilar el sistema las dependencias circulares producen errores difíciles de solucionar. Es este aspecto se ha puesto especial cuidado. El principal mecanismo para evitar las dependencias es el avance de declaración. En C++ podemos avanzar la declaración

```
#ifndef SCHEMA_INSTANCE_H
#define SCHEMA_INSTANCE_H
#include "schemaimplementation.h"

class schemainstance{
private:
   ...
   schemaimplementation* impl;
   ...
};
#endif
```

```
#ifndef SCHEMA_IMPLEMENTATION_H
#define SCHEMA_IMPLEMENTATION_H
#include "schemainstance.h"

class schemaimplementation{
private:
   ...
   schemainstance* inst;
   ...
};
#endif
```

⇓⇓

```
class schemaimplementation{
   ...
   schemainstance* inst;
   ...
};

class schemainstance{
   ...
   schemaimplementation* impl;
   ...
};
```

Cuadro 4.2: *Declaraciones con dependencias circulares*

de una clase con `class nombre_clase;`, también se pueden avanzar las declaraciones de tipos. Además en C++ se permite el uso de punteros a clases en declaración de las que sólo se conoce su avance de declaración. Esta es la clave para resolver las dependencias circulares. Un ejemplo de esto lo vemos entre las clases `schemainstance` y `schemaimplementation`. Una usa a la otra y viceversa. En parte superior de la tabla 4.2 vemos como sería su declaración normal. Al preprocesar `schemainstance` se incluiría el fichero `schemaimplementation.h`, que a su vez incluye `schemainstance.h`, como el nombre `SCHEMA_INSTANCE_H` ya se ha definido esta última inclusión no tiene ningún efecto. Así, el resultado despues del preprocesado se puede ver en la parte inferior de la tabla 4.2. Si ahora lo compilamos, se produce un error que nos indica que `schemainstance` no se ha declarado a la altura de la clase `schemaimplementation` y por tanto no puede usarse en su miembro `impl`. Para solucionar esto usamos el avance de declaración. En la tabla 4.3 vemos las mismas declaraciones pero con avances. Ahora no es necesario incluir ningún fichero, estos se incluirán posteriormente en los ficheros de código donde ya no provocarán dependencias circulares.

Todo el código del sistema sigue estas simples reglas para evitar errores de compilación por dependencias circulares. Además, se incluye un fichero de encabezados `jde+.h` que simplifica el uso del sistema para los desarrolladores de esquemas. Este incluye todas las declaraciones que un esquema puede necesitar, y basta incluirlo en el encabezado de nuestro esquema para poder acceder a todas las clases que `schemaimplementation` tiene a su alcance.

En cuanto a los esquemas ya se ha descrito que requieren de una clase factoría que sepa cargar el derivado de `schemaimplmentation` que implementan, todo ello agrupado en una librería dinámica que el sistema cargará. Pero además, se requiere una función cargadora para poder obtener la factoría de la librería dinámica, ya que los mecanismos de carga dinámica utilizados no saben de objetos o clases. Esta función cargadora

#ifndef SCHEMA_INSTANCE_H
#define SCHEMA_INSTANCE_H
class schemaimplementation;
 	
class schemainstance{
private:
...
schemaimplementation* impl;
...
};
#endif
	#ifndef SCHEMA_IMPLEMENTATION_H
	#define SCHEMA_IMPLEMENTATION_H
	class shemainstance;
	
	class schemaimplementation{
	private:
	...
	schemainstance* inst;
	...
	};
	#endif

Cuadro 4.3: *Avance de declaraciones*

extern "C"{
extern schemafactory* get_factory();
}
	extern "C"{
	extern schemafactory* get_factory(){
	return new schema_A_factory();
	}
	}

Cuadro 4.4: *Función cargadora de* `schemaA_factory`

simplementa devolverá una instancia de la factoría del esquema en cuestión. También debemos tener en cuenta que el mecanismo de carga dinámica esta desarrollado en y para C, mientras nuestro sistema esta desarrollado en C++. El problema que esto conlleva, es que internamente C y C++ manejan de diferente manera los nombres de sus símbolos. Para solucionarlo el compilador `g++` incluye la estructura `extern ''C''` para indicar que use nombres estilo C. Así, la función cargadora de la que hablábamos deberá usar esta estructura, para que el cargador dinámico la pueda localizar. La tabla 4.4 muestra la declaración y definición de una función cargadora ejemplo.

```
extern "C"{
  extern schemafactory* get_factory();
}
```

Todo el código de este proyecto junto con los ejemplos desarrollados en el capítulo 5 se ofrecen con licencia GPL, y se entregan en un cdrom que acompaña a esta memoria. Además se podrá encontrar en la web del grupo de robótica, en la sección de proyectos.

4.3.2 Construcción del sistema

La construcción o compilación del sistema es el proceso por el cual obtenemos el ejecutable del sistema. La construcción del sistema *jde+* consiste en obtener el código máquina de cada unidad de compilación, enlazarlas y ensamblarlas en un archivo ejecutable.

El proceso de construcción se realiza de manera "casi" automática gracias a las herramientas descritas en la sección 3.6, las `Autotools`. Con estas herramientas basta definir los ficheros fuente que requiere un ejecutable, las opciones de compilación y las librerías externas que se usarán. Con esta información, se obtienen una serie de `scripts` que al ejecutarse llaman al compilador y enlazador, obteniendo si no se producen errores el ejecutable del sistema.

El compilador utilizado para construir el sistema es `gcc`, un compilador para C/C++ de libre distribución con gran soporte. Lo usaremos a través de `g++` que es un recubrimiento que selecciona el lenguaje C++ de `gcc`. De entre las opciones utilizadas para

la compilación la única destacable es la que esta directamente relacionada con la carga dinámica de código. Por defecto, el ejecutable construido no exporta la tabla de símbolos a las librerías cargadas de manera dinámica. Esto hace que el código de los esquemas, cargado dinámicamente, no tenga visibilidad sobre los símbolos del sistema (variables, objetos, métodos, ...). El compilador incluye una opción especifica para forzar el exportado de dicha tabla de símbolos. La opción es `--export-dynamic` y se debe utilizar en tiempo de enlazado.

El resto del proceso es bastante típico y no requiere de más detalles. Al final, obtenemos el ejecutable del sistema nombrado `jde+`. Su uso se detalla en el apartado 4.3.4.

4.3.3 Construcción de los esquemas

La construcción de un esquema es el proceso por el cual obtenemos una librería dinámica que el sistema *jde+* es capaz de cargar. Este proceso no esta automatizado, por lo que lo describimos a continuación.

El primer paso es obtener el código objeto del esquema , sus tipos de datos, sus interfaces y en general todo el código fuente que implementa un esquema. Para ello usaremos el compilador de la siguiente manera:

```
$> g++ -c <otras opciones> -o esquemaA.o esquemaA.cpp
$> g++ -c <otras opciones> -o tipoX.o tipoX.cpp
$> g++ -c <otras opciones> -o interfazZ.o interfazZ.cpp
```

Con esto obtendremos (si no se producen errores en la compilación) ficheros con extensión ".o" que incluyen el código objeto del fichero fuente compilado. A continuación los agruparemos para que formen una librería dinámica o `shared object`. Para ello, usaremos de nuevo el compilador (aunque esta vez actuara como enlazador) como sigue:

```
$> g++ -shared -o esquemaA.so esquemaA.o tipoX.o interfazZ.o
```

Y con esto obtenemos el fichero `esquemaA.so` listo para cargarse en el sistema `jde+`.

4.3.4 Uso del sistema

El uso del sistema consisten en construir una jerarquía que desarrolle un comportamiento determinado. Para ello bastará indicar al sistema donde encontrar los esquemas y cual es el que debe colocar en la raíz. A partir de este momento, el sistema cargará el esquema raíz y lo activará. Desde ese momento, el sistema se encargará de cargar los esquemas necesarios, de gestionar los esquemas instanciados y de manejar las comunicaciones. La situación en cada instante realizará el resto, configurando la jerarquía que acabará activándose en cada momento.

Como ejemplo se incluye un programa principal que inicia el sistema, aunque es posible que las necesidades particulares requieran adaptarlo. Lo mostramos a continuación:

```
#include <iostream>
#include <vector>
#include <stdexcept>
#include "hierarchy.h"

using namespace std;
```

```
int main(int argc, char* argv[]){
  vector<string> path;
  char c;

  if (argc < 3){
    cerr << "Usage: " << argv[0] << " root_interface path1 path2 ...\n";
    return 1;
  }

  for (int i=2; i < argc; i++)
    path.push_back(argv[i]);

  cout << "JDE+\n";
  try{
    hierarchy h(path);

    h.active(argv[1]);
    cerr << "Push any key to exit....\n"
    cin >> c;
  }
  catch(std::exception& e){
    cerr << "Jde+: exception raised: " << e.what() << "\n";
  }
  catch(...){
    cerr << "Jde+: unknown exception raised\n";
  }
  return 0;
}
```

El programa acepta varios parámetros. Uno inicial que indica el nombre del interfaz del esquema que estará en la raíz de la jerarquía. Y a continuación las rutas donde el sistema buscará esquemas para cargar. Con estos datos, primero se crea el objeto `hierarchy`, al que se le pasa un vector con las rutas, y despues se le indica que active una jerarquía con la raíz indicada. El programa espera hasta que el usuario pulsa cualquier tecla para terminar, momento en que se paran todos los esquemas y se finaliza la ejecución. Una ejemplo de ejecución sería así:

```
$> jde+ "dummy_interface" ./schemas
JDE+
Hierarchy: Loading schemas from ./schemas
Hierarchy: loading ./schemas/schema_A.so for interface 'dummy_interface'
Hierarchy: loading ./schemas/schema_B.so for interface 'message_interface'
Hierarchy: 2 loaded successfully
Hierarchy: creating instance(ROOT) of 'dummy_interface' interface
...
...
Push any key to exit....
```

Este programa será el utilizado para realizar las pruebas que veremos ya en el siguiente capítulo.

Capítulo 5

Pruebas

En este capítulo mostraremos una serie de pruebas realizadas con el fin de demostrar la validez del sistema, frente a los objetivos que marcamos. Para ello, se han elegido cuatro ejemplos, que cubren las características más importantes del sistema. Para cada uno se han implementado una serie de esquemas de prueba con los que se construye una jerarquía. Una vez en ejecución el sistema, se observan los resultados, y se contrastan con lo esperado. Se incluye parte del código de los ejemplos realizados con los objetivos, de erradicar las posibles dudas, y demostrar ejemplos completos y reales del sistema en funcionamiento. Todo el código se puede encontrar en el cdrom anexo a la memoria con licencia GPL.

Los ejemplos elegidos muestran los siguientes aspectos del sistema:

- Mecanismos de comunicación: percepción, modulación y cambio de estado.
- Mecanismos de selección de acción.
- Instanciación múltiple de un esquema sin reutilización.
- Instanciación múltiple de un esquema con reutilización.

Los esquemas que presentaremos en los ejemplos realizan tareas simples para no ensombrecer lo que realmente queremos mostrar, el funcionamiento y uso del sistema. Por ello, las tareas elegidas son un tanto extravagantes, y alejadas de la robótica, como por ejemplo la generación de números aleatorios o el chequeo de probabilidades.

Como indicamos en el apartado 4.2.2, resulta interesante implementar *wrappers* para la clase `data`, capaces de manejar los tipos de datos que usen nuestros esquemas. Para demostrar esto, en nuestros ejemplos incluiremos *wrappers* típicos, para los tipos de datos que usemos.

5.1 Ejemplo 1: percepción, modulación y cambio de estado

Este primer ejemplo trata de mostrar la funcionalidad de los mecanismos de comunicación que los esquemas tienen a su disposición. En él veremos una jerarquía formada por tres esquemas, que se envían percepciones, se modulan y se cambian de estado. El cuarto tipo de comunicación, la petición de arbitraje, se verá más adelante cuando hablemos del mecanismo de selección de acción.

Concretando, la jerarquía esta formada por un esquema motor y dos perceptivos, a los que denominaremos EM1, EP1 y EP2 respectivamente. EM1 es el esquema raíz, y tiene por hijos a EP1 y EP2. El objetivo de EM1, es mostrar por pantalla una clave aleatoria. Para ello, usa la funcionalidad de EP1, que es capaz de generar números

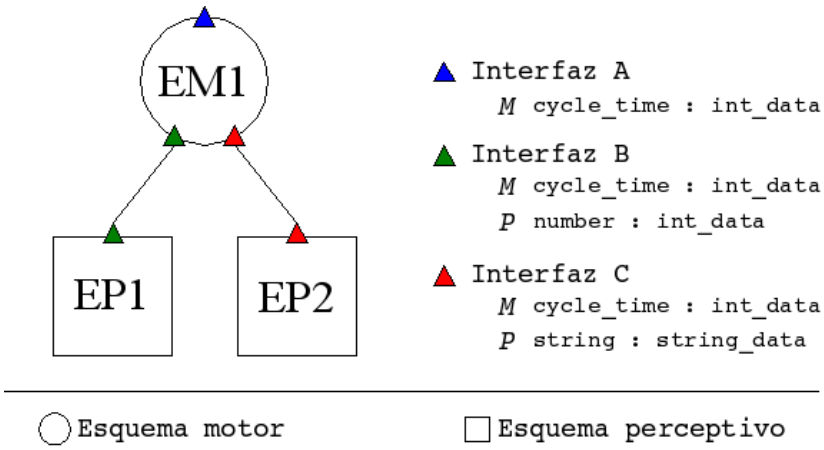


Figura 5.1: *Jerarquía del ejemplo 1*

aleatorios, y la de EP2 capaz de generar cadenas de caracteres aleatorias. En el ejemplo veremos como EM1, activará, modulará a EP1 y EP2, con el fin de alcanzar su objetivo.

En cuanto a los interfaces con los que se relacionan, tenemos que EM1 implementa el interfaz A, que únicamente tiene el parámetro de modulación `cycle_time` con el que se ajusta el tiempo de ciclo del esquema. EP1 por su parte, implementa el interfaz B que tiene el parámetro de modulación `cycle_time`, y produce la percepción `number`. Y por último, EP2 implementa el interfaz C que también tiene el parámetro de modulación `cycle_time`, y produce la percepción `string`. Por simplicidad se han escogido los nombres A,B y C para los interfaces, aunque en una aplicación real sería más conveniente escoger nombres que describan la función para la que sirven, para identificar mejor la capacidad del esquema que los implemente. Los tipos de datos usados son enteros y cadenas de caracteres, que recubriremos con los *wrappers* `int_data` y `string_data`. Gráficamente lo podemos describir como se ve en la figura 5.1.

Lo primero que vamos a describir son los *wrappers* para los tipos de datos usados, mostrando la implementación de alguno de ellos. Luego hablaremos de los interfaces, y de nuevo mostraremos alguna implementación. Seguiremos con los esquemas, viendo cómo se une la funcionalidad propia de cada esquema a todo lo comentado anteriormente. Y para finalizar, ejecutaremos la jerarquía y comentaremos los datos obtenidos.

5.1.1 *Wrappers* para los tipos de datos

En este ejemplo, los tipos de datos usados son muy simples, enteros y cadenas de caracteres, ya que los esquemas sólo intercambian números y palabras. En una jerarquía más compleja, podríamos tratar tipos de datos más interesantes y complejos, como imágenes, o lecturas de un láser. Aun así para ilustrar su uso recubriremos la clase `data`, con dos clases *wrapper* `int_data` y `string_data`, que albergaran un entero y una cadena de caracteres respectivamente. Además ofrecerán operaciones para facilitar su uso y trasformación de y hacia un objeto `data`.

Para su implementación usaremos la clase `data` ofrecida por el sistema, como contenedor genérico de datos, descrita en la sección 4.2. Esta clase nos permite transportar cualquier dato, sin más que indicarle el tamaño del mismo y un puntero a su localización en memoria. Ambos *wrappers* tienen implementaciones similares, por lo que sólo entraremos en los detalles de `int_data`, para `string_data` se aplicarán los mismos razonamientos salvo que el tipo manejado es diferente.

La declaración de la clase `int_data` es la siguiente:

```
class int_data{
public:
```

```
  int_data();/*constructor*/
  int_data(const int v);/*constructor with initialization*/
  int_data(const int_data& i);/*copy constructor*/
  int_data(const data& d);/*copy constructor from data object reference*/
  int_data(data* d);/*constructor from data object pointer*/
  ~int_data();/*destructor*/

  /*cast operators*/
  operator data&() const;/*converts int_data to data reference*/
  operator int() const;/*converts int_data to int*/

  /*assign operator*/
  int_data& operator=(const int v);

private:
  int get_value() const;/*return int value*/
  void set_value(const int v);/*assign int value*/
  data* int_d;
  bool data_owned;
};
```

Como podemos apreciar, se incluyen todo tipo de constructores, operadores de conversión y de asignación. Con esta clase, manejar el tipo `int` sobre un objeto `data` es trivial. Podemos construirlo, asignarlo e incluso convertirlo (data⇔int_data⇔int). Cuando expliquemos el código de los esquemas veremos ejemplos de su uso.

La clase `string_data` sigue la misma idea, constructores y operadores de todo tipo para manejar el tipo `char[]` o cadena de caracteres. También veremos ejemplos de su uso más adelante.

5.1.2 Interfaces

Como vimos en la sección 4.2 un esquema se presentaba al sistema mediante un *interfaz*, y definía sus dependencias usando *interfaces* que describiesen las funcionalidades requeridas. Estos *interfaces* se implementan con clases derivadas de `mschema` y `pschema`, según describan esquemas motores o perceptivos, que redefinen sus funciones abstractas. Por ello debemos implementar los *interfaces* que representarán a los esquemas de este ejemplo.

Como hemos presentado antes, los *interfaces* de nuestros esquemas llevarán los nombre A,B y C. A es un interfaz para un esquema motor, y tiene un solo parámetro de modulación, `cycle_time` para ajustar el tiempo de ciclo del esquema. B es un interfaz para esquemas perceptivos, y tiene también otro parámetro de modulación para el tiempo de ciclo más un parámetro de percepción denominado `number` con el que se envían las percepciones generadas por este esquema. C también es un interfaz para esquemas perceptivos, incluye un parámetro de modulación para ajustar el tiempo de ciclo y uno de percepción denominado `string` para el envío sus percepciones.

La declaración del interfaz A es la siguiente:

```
class A: public mschema{
public:
  A(string nick = "A");
  ~A();
  virtual bool has_data_name(const string& data_name) const;
  virtual vector<string> get_perceptions_names() const;
```

```
  virtual vector<string> get_modulations_names() const;
private:
  virtual data& get_data(const string& data_name);
  /*data*/
  int_data* cycle_time;
  /*data names*/
  static vector<string>* const perceptions;
  static vector<string>* const modulations;
};
```

Como se puede apreciar, la clase `A` se deriva de la clase abstracta `mschema` (a su vez derivada de `schemainterface`), por lo que `A` es un interfaz de esquema motor. En la parte pública podemos ver que se declaran los métodos abstractos destinados a la introspección del interfaz, y en la parte privada tenemos los datos.

Los interfaces B y C tienen declaraciones similares.

5.1.3 Esquemas

Los esquemas que se utilizan en este ejemplo son EM1, EP1 y EP2, como vimos al inicio de este apartado. EM1 es un esquema motor representado por el interfaz A. EP1 y EP2, por su parte, son esquemas perceptivos representados por los interfaces B y C respectivamente.

EM1 tiene por objetivo mostrar por pantalla una clave aleatoria, puesto que es un esquema motor, diremos que ésta es su actuación. Para generar su actuación usa dos esquemas perceptivos que implementen los *interfaces* B y C. Usando la notación que introdujimos en el punto 4.2.1 diríamos que EM1 implementa "A(cycle_time,)", y que usa "B(cycle_time,number)" y "C(cycle_time,string)". Por su parte B y C están implementados por el esquema EP1 que genera números aleatorios, y EP2 que genera cadenas de caracteres aleatorias.

Cada esquema se implementa como una clase derivada de `schemaimplementation`, que redefine los métodos que incorporan la funcionalidad de un esquema: `iteration`, `arbitrate` y `check_preconditions`. En este ejemplo no se producirán conflictos de control ya que los dos esquemas hijos de EM1 son perceptivos y no compiten entre sí. Además siempre se cumplirán las precondiciones de EM1. Por esto el único método con funcionalidad interesante será `iteration`. A continuación mostramos la implementación de cada uno de los esquemas. En el siguiente apartado veremos a estos esquemas en acción.

El método `iteration` de EM1 es:

```
void EM1::iteration(){
  int_data cycle_time(&get_modulation("cycle_time"));
  double t = gettimeofdaysec() - get_system_start_time();
  static int i = 0;

  set_cycle_time(cycle_time);

  if (i==0){
    /*wakeup children*/
    cout << "EM1(" << t << "): Waking Up children 'B & C'\n";
    set_child_state("B",ACTIVE);
    set_child_state("C",ACTIVE);
    /*modulate cycle_time*/
    cout << "EM1(" << t << "): modulating children 'B & C' cycle_time\n";
```

```
    modulate_child("B","cycle_time",cycle_time);
    modulate_child("C","cycle_time",cycle_time);
    i++;
  }else{
    int_data number(&get_child_perception("B","number"));
    string_data _string(&get_child_perception("C","string"));

    cout << "EM1(" << t << "): I'm alive with sid: "
         << sid_to_str(my_sid()) << "\n";
    cout << "EM1(" << t << "): number perception from B: "
         << (int)number << "\n";
    cout << "EM1(" << t << "): string perception from C: "
         << (char*)_string << "\n";
    cout << "EM1(" << t << "): generating key: " << (int)number
         << (char*)_string << "\n";
  }
}
```

En la primera iteración se ejecutará el código del `if`, entonces despertaremos (poniendolos en estado ACTIVO) a los dos hijos. Hay que resaltar la manera de referirnos a los hijos, que como se puede observar en el código se hace usando el nombre de su interfaz. En ningún caso se usa el nombre de los esquemas, que de hecho EM1 no conoce. A continuación modularemos su tiempo de ciclo, y aunque en este ejemplo se realiza sólo en la primera iteración, típicamente un esquema modula a sus hijos de manera continua, variando los parámetros como mejor le convenga. Las funciones `modulate_child` y `set_child_perception` son facilidades que aporta el sistema para que un esquema se comunique con sus hijos. Estas funciones generan los mensajes adecuados y los entregan al `isc` que se encarga del resto. `get_modulation` también esta implementada por el sistema, y permite que un esquema obtenga la última modulación recibida.

En el resto de iteraciones se ejecutará el código del `else`, donde obtendremos el valor de las percepciones que nuestros hijos ya habrán enviado. Con estos datos construimos la clave (simplemente concatenandolos) y la mostraremos por pantalla.

Las funciones `get_system_start_time` y `set_cycle_time` son también facilidades del sistema. La primera devuelve la hora a la que se inicio el sistema, utilizada como referencia temporal para pintar marcas de tiempo que nos ayuden en la depuración, las veremos en el apartado ejecución al lado de algunos mensajes. La segunda sirve para que un esquema pueda fijar su tiempo de ciclo.

El método `iteration` de EP1 es:

```
void EP1::iteration(){
  int_data cycle_time(&get_modulation("cycle_time"));
  int_data number;
  double t = gettimeofdaysec() - get_system_start_time();
  static unsigned int seed;

  set_cycle_time(cycle_time);

  cout << "EP1(" << t << "): I'm alive with sid: "
       << sid_to_str(my_sid()) << "\n";
  number = rand_r(&seed);
  cout << "EP1(" << t << "): Sending perception to father: "
       << (int)number << "\n";
```

```
  send_perception("number",number);
}
```

En cada iteración se genera un número aleatorio y se envía al padre, indicando que el nombre de esta percepción en `number`. Otra vez `send_perception` es parte de la funcionalidad ofrecida por el sistema. Esta se encarga de localizar al padre actual del esquema (EP1 no sabe que su padre es EM1 ni lo necesita), construir el mensaje adecuado y entregarlo al `isc`.

El método `iteration` de EP2 es:

```
void EP2::iteration(){
  int_data cycle_time(&get_modulation("cycle_time"));
  string_data _string(rand_string(8));
  double t = gettimeofdaysec() - get_system_start_time();

  set_cycle_time(cycle_time);

  cout << "EP2(" << t << "): I'm alive with sid: "
       << sid_to_str(my_sid()) << "\n";
  cout << "EP2(" << t << "): Sending perception to father: "
       << (char*)_string << "\n";
  send_perception("string",_string);
}
```

En este caso en cada iteración se genera una cadena de caracteres aleatoria y se envía al padre con el nombre `string`.

5.1.4 Ejecución

Para ejecutar la jerarquía, basta que compilemos los esquemas tal y como indicamos en el apartado 4.3.3. Una vez compilados los esquemas ejecutaremos el sistema, como también indicamos en el apartado 4.3.4, indicando el interfaz que representa al esquema raíz (A en nuestro ejemplo) y la ruta donde se almacenan los esquemas en disco.

En nuestro sistema de pruebas se haría así:

```
$> jde+ A ./schemas
```

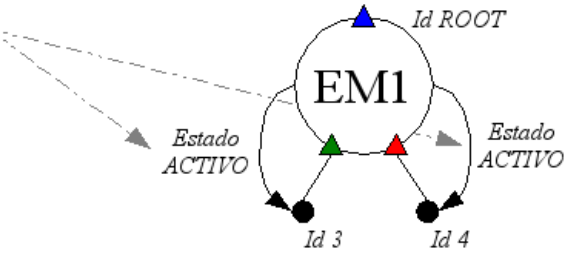
Esto iniciaría el sistema, que cargaría los esquemas y comenzaría creando la instancia raíz EM1. Como se puede apreciar en el código de los esquemas, parte de las instrucciones generan trazas que nos permiten analizar el funcionamiento de la jerarquía. En la figura 5.2 vemos la salida de las dos primeras iteraciones junto a gráficos que detallan lo que esta sucediendo. En el primero podemos ver la instancia raíz a la que aun no se le han enlazado las instancias de sus hijos. Tras la activación dichas instancias se crean y comienzan a ejecutar su iteración. En el segundo gráfico vemos como EM1 modula el tiempo de ciclo de sus hijos. En el último gráfico vemos como EP1 y EP2 envían sus percepciones a su padre EM1.

Además de las trazas generadas por cada instancia, el sistema genera un registro de las comunicaciones. Dicho registro detalla el tipo de mensaje[1], el origen, el destino, información adicional sobre el mensaje (por ejemplo si es un cambio de estado el nuevo estado), la hora de envío y la hora de entrega (con respecto al inicio del sistema). Un ejemplo de este registro sería así:

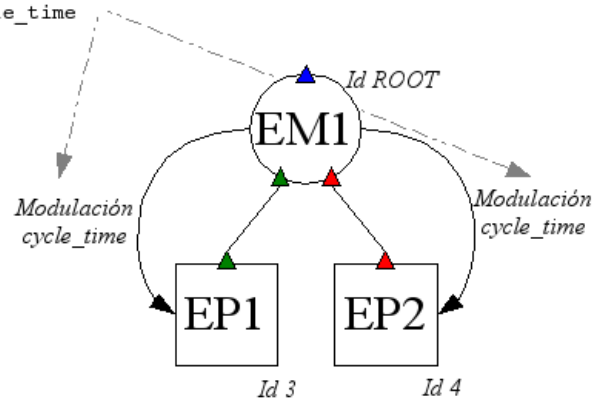
[1]S: cambio de estado, M: modulación, P: percepción y A: petición de arbitraje

```
S,2,2,ALERT,0.00673604,0.00746202
S,2,3,ACTIVE,0.111884,0.112654
S,2,4,ACTIVE,0.111897,0.113338
M,2,3,cycle_time,0.111952,0.113622
M,2,4,cycle_time,0.111962,0.113911
P,3,2,number,0.217879,0.218288
P,4,2,string,0.218024,0.218543
P,3,2,number,5.22011,5.22054
P,4,2,string,5.22025,5.2208
P,3,2,number,10.2224,10.2228
```

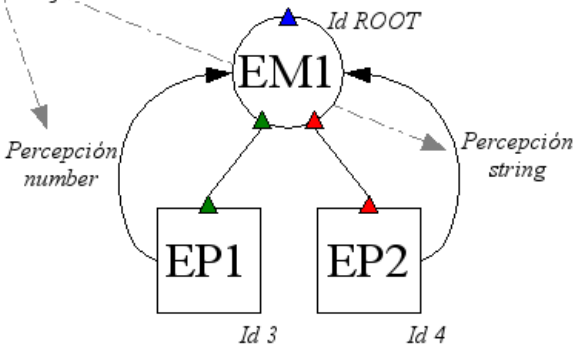
```
JDE+
EM1: Creating schema of interface 'A'
EM1(0.270687): Waking Up children 'B & C'
EP1: Creating schema of interface 'B'
EP2: Creating schema of interface 'C'
```




```
EM1(0.270687): modulating children 'B & C' cycle_time
```




```
EP1(0.374654): I'm alive with sid: 3
EP1(0.374654): Sending perception to father: 1012484
EP2(0.374743): I'm alive with sid: 4
EP2(0.374743): Sending perception to father: ZyFKDbwj
EM1(5.27493): I'm alive with sid: ROOT
EM1(5.27493): number perception from B: 1012484
EM1(5.27493): string perception from C: ZyFKDbwj
EM1(5.27493): generating key: 1012484ZyFKDbwj
```



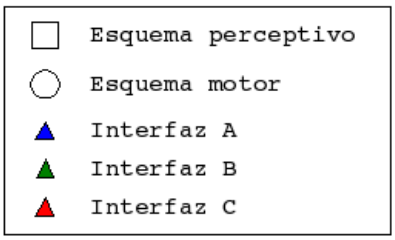


Figura 5.2: *Ejecución del ejemplo 1*

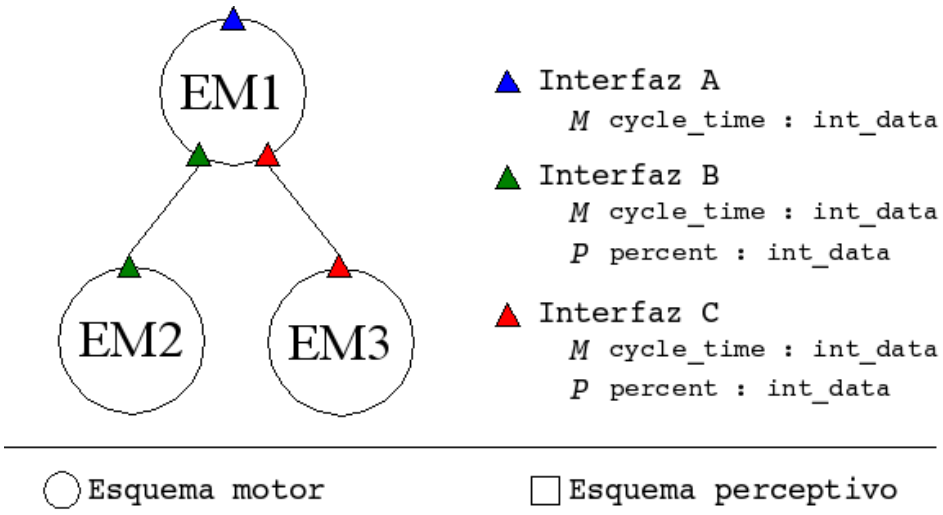


Figura 5.3: *Jerarquía del ejemplo 2*

Con estos datos podemos trazar los mensajes que envió cada esquema en cada instante, para poder verificar que funcionan correctamente. Además podemos medir la eficiencia del comunicador, calculando la media de tiempo que un mensaje permanece encolado.

5.2 Ejemplo 2: selección de acción

En este ejemplo mostraremos los mecanismos de selección de acción que implementa el sistema *jde+*. Al igual que en el ejemplo anterior desarrollaremos una serie de esquemas que nos permitan hacer hincapie en dichos mecanismos, sin perseguir objetivos complejos que distraigan la atención de lo que se intenta mostrar.

Para este ejemplo construiremos una jerarquía con tres esquemas motores, EM1 que se colocara como raíz, y EM2 y EM3 que serán sus hijos. Los esquemas hijos incluirán precondiciones, que el padre podrá controlar, mostrando el arbitraje de grano grueso. Y al despertar simultáneamente a los dos hijos obtendremos situaciones en las que se produzcan conflictos de control, que el padre deberá atender, mostrando el arbitraje de grano fino.

En este ejemplo la funcionalidad de los tres esquemas será simplemente mostrar un mensaje por pantalla, con la que podremos ver que esquema ejecuta su iteración en cada instante. Los tres esquemas aceptan la modulación de su tiempo de ciclo. Las precondiciones de los hijos se han implementado con un mecanismo estadístico. El funcionamiento es similar a tirar un dado con 100 caras en cada chequeo, si no se supera cierto umbral se cumplen las precondiciones, en caso contrario no se satisfacen. Dicho umbral es modulable y lo ajustará el padre del esquema.

Los interfaces usados se denominan igual que en el ejemplo anterior A,B y C. Todos tienen el parámetro de modulación `cycle_time` para ajustar el tiempo de ciclo. B y C, además tienen el parámetro de modulación `threshold` con el que se modula el umbral de las precondiciones. Aunque B y C son iguales, para este ejemplo los trataremos como si fuesen interfaces que representan a esquemas diferentes, para no entrar aún en la instanciación múltiple, que veremos con el siguiente ejemplo.

En la figura 5.3 podemos ver como es la jerarquía de este ejemplo. Para los datos se ha reutilizado el recubrimiento `int_data` del ejemplo anterior, por lo que no detallaremos este aspecto. En cuanto a los interfaces, aunque incluyen campos diferentes, siguen los mismos principios que los mostrados en el ejemplo 1. Así, pasamos a ver directamente la implementación de cada esquema.

5.2.1 Esquemas

En este ejemplo, los esquemas requieren funcionalidad especifica para tratar los mecanismos de selección de acción. Por ello, en este caso además del método `iteration`, redefiniremos `arbitrate` para EM1, y `check_preconditions` para EM2 y EM3.

EM1 es un esquema motor que requiere dos hijos motores que implementen los *interfaces* B y C. En el supuesto de que despierte a sus dos hijos simultáneamente, cabe la posibilidad de que surjan conflictos de control, vacíos de control si ninguno supera sus precondiciones, o solapes de control si los dos quieren activarse a la vez. Por ello, este esquema debe en primer lugar intentar que sus esquemas hijos se activen en situaciones complementarias, o lo que hemos denominado arbitraje de grano grueso. Para ello ajustará el umbral de sus precondiciones, que aunque no resulta un mecanismo demasiado potente, si es muy simple y nos permite calcular el porcentaje de solapes y vacíos de control que queremos causar. Por ejemplo con umbrales 30 y 70, se producirán $0{,}3x0{,}7 = 0{,}21 \Rightarrow 21\,\%$ de solapes de control y $0{,}7x0{,}3 = 0{,}21 \Rightarrow 21\,\%$ de vacíos de control.

Pero en el caso de que surjan conflictos, el esquema debe responder arbitrando un ganador. Un arbitraje correcto debería tener en cuenta varios aspectos para poder elegir a un ganador justo, pero de nuevo para simplificar el ejemplo siempre se elegirá al mismo ganador. Esta funcionalidad se incorpora en el método `arbitrate`.

El código de los métodos relevantes del esquema EM1 es el siguiente:

```
void EM1::iteration(){
  int_data cycle_time(&get_modulation("cycle_time"));
  double t = gettimeofdaysec() - get_system_start_time();
  static int i = 0;

  set_cycle_time(cycle_time);

  if (i==0){
    /*wakeup children*/
    cout << "EM1(" << t << "): Waking Up children 'B & C'\n";
    set_child_state("B",ALERT);
    set_child_state("C",ALERT);
    /*modulate cycle_time*/
    cout << "EM1(" << t << "): modulating children cycle_time\n";
    modulate_child("B","cycle_time",cycle_time);
    modulate_child("C","cycle_time",cycle_time);
    /*adjusting activation regions*/
    cout << "EM1(" << t << "): modulating children preconditions\n";
    modulate_child("B","threshold",int_data(30));/*30% to get READY state*/
    modulate_child("C","threshold",int_data(70));/*70% to get READY state*/
    i++;
  }else{
    cout << "EM1(" << t << "): I'm alive with sid: "
         << sid_to_str(my_sid()) << "\n";
    cout << "EM1(" << t << "): executing iteration\n";
  }
}

void EM1::arbitrate(const string& request_nick){
  double t = gettimeofdaysec() - get_system_start_time();
```

```
  cout << "EM1(" << t << "): arbitration request from: "
       << request_nick << "\n";
  if (request_nick == "B"){
    cout << "EM1(" << t << "): B go to ALERT\n";
    set_child_state("B",ALERT);
  }else{
    cout << "EM1(" << t << "): C go to ACTIVE(winner!)\n";
    set_child_state("C",ACTIVE);
  }
}
```

En `iteration` vemos que en la primera iteración se pone en estado ALERTA a los hijos, y se modula su tiempo de ciclo y su umbral para las precondiciones. De nuevo, solo aplicamos la modulación en una iteración, aunque como se comento típicamente en aplicaciones reales se modula continuamente variando la influencia en sus hijos según varíe la situación del entorno. En el resto de iteraciones simplemente se pinta un mensaje por pantalla.

En `arbitrate` simplemente comprobamos de quien viene la petición, en caso de que provenga de B le enviamos al estado ALERTA y en caso de que provenga de C le indicamos que se ACTIVE. Es importante destacar que las competiciones están totalmente distribuidas, y que cada hijo detectará los conflictos de manera independiente, contactando cada uno por separado al padre que les contesta en exclusiva qué hacer (cambiar de estado generalmente).

Los esquemas hijos por su parte deben comprobar sus precondiciones y en caso de obtener el estado ACTIVO ejecutan su iteración, que simplemente consiste en pintar un mensaje por pantalla.

El código del esquema EM2 es el siguiente:

```
void EM2::iteration(){
  int_data cycle_time(&get_modulation("cycle_time"));
  double t = gettimeofdaysec() - get_system_start_time();

  set_cycle_time(cycle_time);

  cout << "\tEM2(" << t << "): I'm alive with sid: "
       << sid_to_str(my_sid()) << "\n";
  cout << "\tEM2(" << t << "): executing iteration\n";
}

bool EM2::check_preconditions(){
  int_data threshold(&get_modulation("threshold"));
  double t = gettimeofdaysec() - get_system_start_time();
  static unsigned int seed;
  int r = rand_r(&seed);
  float p = (float)r/(float)RAND_MAX;
  float perc_p = (float)((int)threshold)/100.0;

  cerr << "\tEM2(" << t << "): p/perc " << p << "/" << perc_p << "\n";
  if (p < perc_p)
    cout << "\tEM2(" << t << "): preconditions reached\n";
  else
    cout << "\tEM2(" << t << "): preconditions not reached\n";
  return p < perc_p;
```

```
}
```

En `iteration` simplemente se imprime un mensaje. En `check_preconditions` simulamos el dado con un número aleatorio, y comprobamos si supera o no el umbral fijado en cada instante. El código de EM3 es idéntico por lo que no lo repetiremos.

5.2.2 Ejecución

La ejecución de este ejemplo, con los umbrales 30/70 para los esquemas EM2 y EM3, produce muchos conflictos de control. Este ejemplo se centra en ver este tipo de situaciones, para ello mostramos las situaciones más destacadas junto con las trazas que producen los esquemas.

La figura 5.4 muestra un caso normal, en el que no se ha producido ningún conflicto de control. EP2 no supera sus precondiciones, pero EP3 si, por lo que pasa a ejecutar su iteración de manera exclusiva.

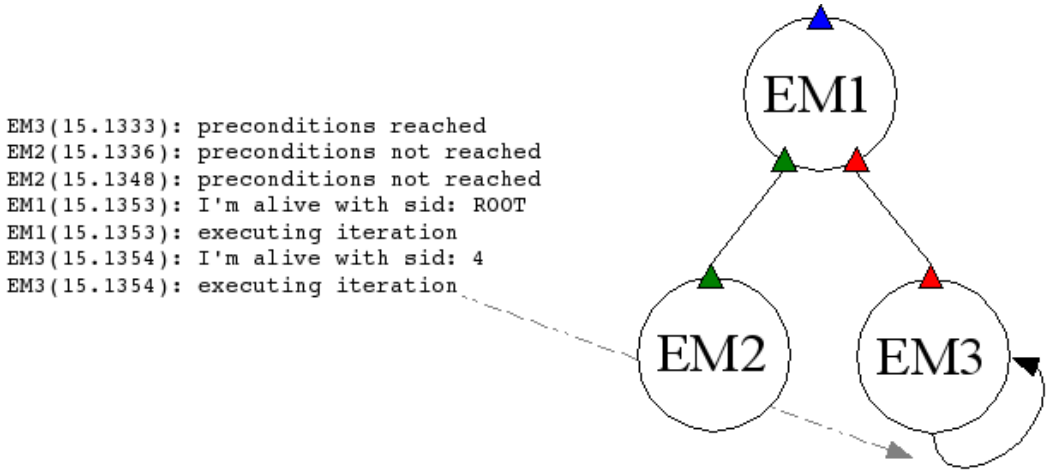


Figura 5.4: *Vacío de control*

En la figura 5.5 se produce un solape de control, cuando EM2 consigue superar sus precondiciones, pasando a estado PREPARADO, mientras EM3 ejecuta su iteración que dura tres segundos. En ese momento, EM2 envía una petición de arbitraje a EM1, que le responde diciéndole que vaya a ALERTA, donde sigue comprobando sus precondiciones.

La figura 5.6 muestra un vacío de control, producido por que ninguno de los esquemas cumple sus precondiciones, quedando el nivel sin ningún esquema motor ACTIVO.

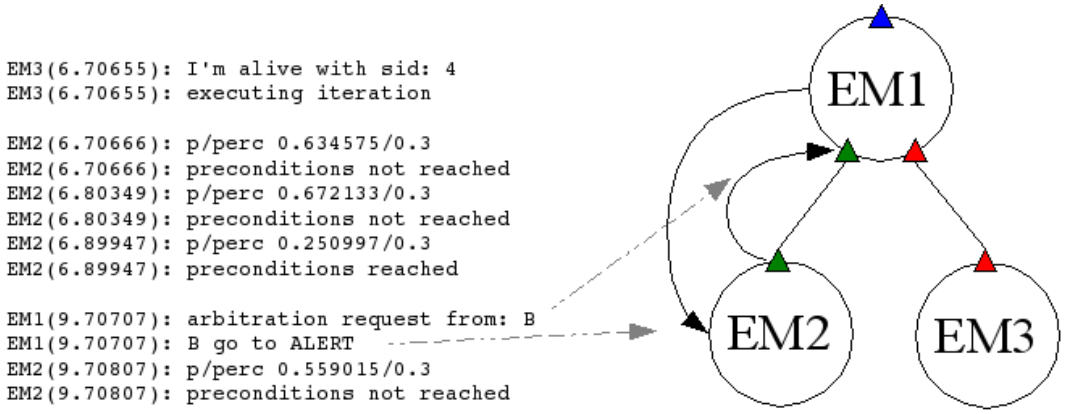


Figura 5.5: *Solape de control*

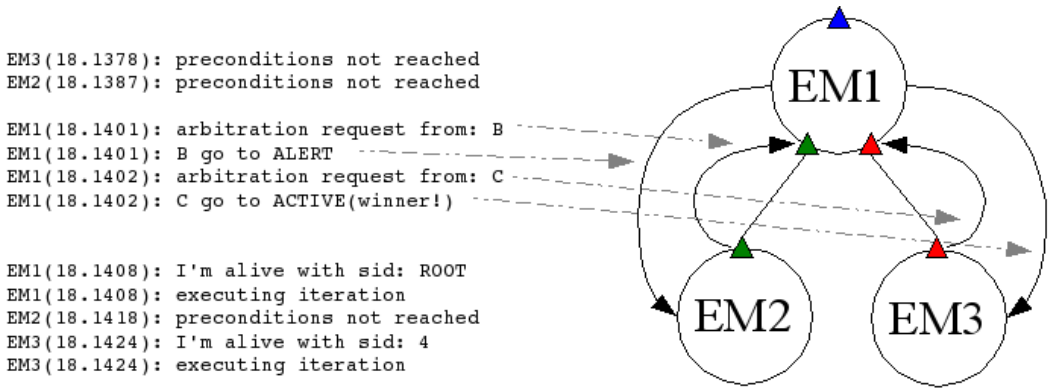


Figura 5.6: *Vacío de control*

Como se puede ver en las trazas, ambos esquemas solicitan el arbitraje, e instantes más tarde el padre les responde, con sendos mensajes, indicando a EP2 que pase a ALERTA comprobando sus precondiciones de nuevo, e indicando a EP3 que se ACTIVE pasando a ejecutar su iteración.

Otra situación que podría suceder es que estando EM2 ejecutando su iteración EM3 cumpla sus precondiciones y pase a PREPARADO. Sucedería un solape de control, y EM3 requeriría arbitraje. Pero en este caso EM1 le indicaría que pasara a ACTIVO, produciéndose una situación anomala, dos esquemas motores en ACTIVO en el mismo nivel. Esto es posible siempre y cuando el arbitro lo permita, y en nuestro caso el arbitraje es demasiado simple para evitar esta situación.

5.3 Ejemplo 3: instanciación múltiple de un esquema sin reutilización

En este tercer ejemplo vamos a mostrar una de las características estrella de *jde+*, la instanciación múltiple de un esquema. Esta consiste en obtener más de una copia de un mismo esquema dentro del sistema. Para ello usaremos un ejemplo similar al primero, salvo que ahora usaremos dos instancias de cada esquema perceptivo. En total tendremos una jerarquía con tres esquemas diferentes, pero con cinco instancias. Todas las instancias se activarán, por lo que no se mostrará la reutilización de esquemas. Este tema se verá en el último ejemplo.

Al igual que en el ejemplo 1, la jerarquía tendrá un esquema motor EM1 con el objetivo de pintar claves aleatorias, salvo que esta vez queremos que sean más grandes. Para ello usaremos cuatro esquemas perceptivos, dos generadores de números aleatorios y dos generadores de cadenas de caracteres aleatorias. Con ello obtendremos una jerarquía como la mostrada en la figura

5.3.1 Esquemas

La implementación de los esquemas que componen esta jerarquía ya la vimos en el ejemplo 1. La única peculiaridad que tiene este ejemplo es la manera en que nos referimos a dos esquemas del mismo tipo. En los ejemplos anteriores, hemos visto que para referimos a un hijo usamos el nombre del interfaz que le representa. Pero ahora resulta que EM1 tiene dos hijos B y dos hijos C. ¿Como hace para referirse a cada uno en particular?.

La solución la comentamos en el apartado 4.2.1 y consiste en asignar “apodos” a los interfaces que los hacen únicos como hijos de un esquema. Con esto, basta que al declarar los interfaces que usará un esquema indiquemos los apodos por los que se

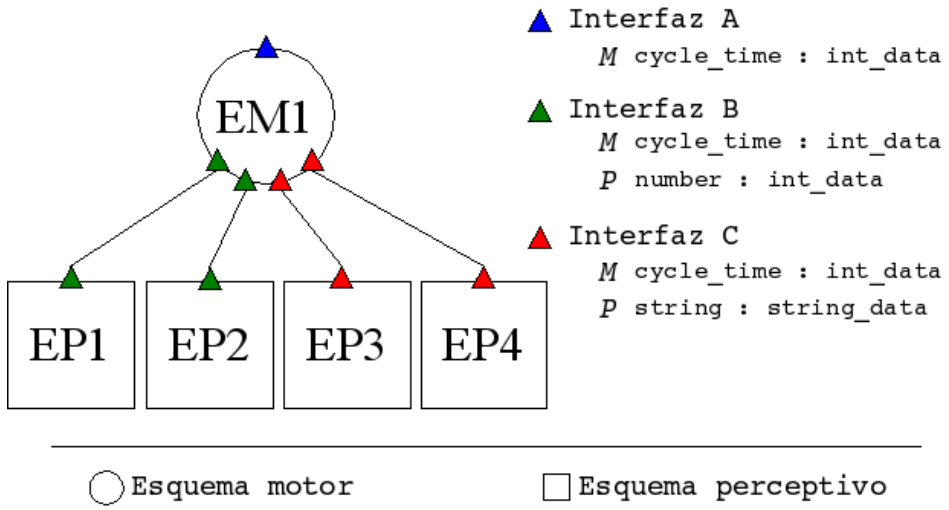


Figura 5.7: *Jerarquía del ejemplo 3*

referirá a ellos. En el siguiente trozo de código se muestra un esquema completo con cuatro hijos, dos de interfaz B y otros dos de interfaz C. Para que esto sea posible, es necesario usar los "apodos".

```
#include <jde+.h>
#include <A.h>
#include <B.h>
#include <C.h>
#include <string_data.h>
#include <int_data.h>
#include <iostream>
#include "util.h"


using std::cerr;
using std::cout;

DECLARE_SCHEMA(EM1,                    /*schema name*/
               A,                      /*schema interface*/
               ADD_INTERFACE(B,"B1"); /*children interfaces*/
               ADD_INTERFACE(B,"B2");
               ADD_INTERFACE(C,"C1");
               ADD_INTERFACE(C,"C1");,
               /*no private data*/)

EM1::EM1()
   : INIT_SCHEMAIMPLEMENTATION{
  cout << "EM1: Creating schema of interface '"
       << interface->interface_name << "'\n";
  /*configure cycle_time default*/
  set_modulation("cycle_time",int_data(5000));/*5s*/
}

EM1::~EM1(){
  cout << "EM1: Deleting schema\n";
}
```

```
void EM1::iteration(){
  int_data cycle_time(&get_modulation("cycle_time"));
  int_data number1(&get_child_perception("B1","number"));
  int_data number2(&get_child_perception("B2","number"));
  string_data _string1(&get_child_perception("C1","string"));
  string_data _string2(&get_child_perception("C2","string"));
  double t = gettimeofdaysec() - get_system_start_time();
  static int i = 0;

  set_cycle_time(cycle_time);

  if (i==0){
    /*wakeup children*/
    cout << "EM1(" << t << "): Waking Up children 'B1/2 & C1/2'\n";
    set_child_state("B1",ALERT);
    set_child_state("B2",ALERT);
    set_child_state("C1",ALERT);
    set_child_state("C2",ALERT);
    /*modulate cycle_time*/
    cout << "EM1(" << t << "): modulating children cycle_time\n";
    modulate_child("B1","cycle_time",cycle_time);
    modulate_child("B2","cycle_time",cycle_time);
    modulate_child("C1","cycle_time",cycle_time);
    modulate_child("C2","cycle_time",cycle_time);
    i++;
  }else{
    cout << "EM1(" << t << "): I'm alive with sid: "
         << sid_to_str(my_sid()) << "\n";
    cout << "EM1(" << t << "): number perception from B1: "
         << (int)number1 << "\n";
    cout << "EM1(" << t << "): number perception from B2: "
         << (int)number2 << "\n";
    cout << "EM1(" << t << "): string perception from C1: "
         << (char*)_string1 << "\n";
    cout << "EM1(" << t << "): string perception from C2: "
         << (char*)_string2 << "\n";
    cout << "EM1(" << t << "): generating key: " << (int)number1
         << (int)number2 << (char*)_string1 << (char*)_string2 << "\n";
  }
}
```

En este ejemplo completo de un esquema podemos ver el uso de algunas macros que ofrece el sistema. La principal es `DECLARE_SCHEMA` que genera la factoría y la declaración de un esquema simplemente indicando el nombre, el interfaz y los interfaces de los hijos requeridos. La macro `ADD_INTERFACE` se usa para insertar hijos, y nos permite declarar los apodos de un interfaz en su segundo parámetro.

En la modulación de los hijos hay que resaltar el uso de los apodos para referirnos a ellos, y no el nombre del interfaz que no nos permitiría modular a dos hijos idénticos.

5.3.2 Ejecución

La única peculiaridad que muestra la ejecución de este ejemplo, es la creación de varias instancias de un mismo esquema. En la traza obtenida de una ejecución del sistema, podemos observar como el sistema carga tres esquemas y crea cinco instancias que se enlazan a su padre con sus apodos correspondientes:

```
JDE+
Hierarchy: Loading schemas from ./schemas
Hierarchy: loading ./schemas/EM1.so for interface 'A'
Hierarchy: loading ./schemas/EP1.so for interface 'B'
Hierarchy: loading ./schemas/EP2.so for interface 'C'
Hierarchy: 3 loaded successfully
Hierarchy: creating instance(ROOT) of 'A' interface
Hierarchy: linking child instance(3) with nick 'B1' to instance(ROOT)
Hierarchy: linking child instance(4) with nick 'B2' to instance(ROOT)
Hierarchy: linking child instance(5) with nick 'C1' to instance(ROOT)
Hierarchy: linking child instance(6) with nick 'C2' to instance(ROOT)
Push any key to exit....
Isc: delivering S,2,2,ALERT
Schemainstance: Creating new implementation
EM1: Creating schema of interface 'A'
Mschemaready(ROOT): ready_active 0
EM1(0.473324): Waking Up children 'B1/2 & C1/2'
EM1(0.473324): modulating children 'B1/2 & C1/2' cycle_time
Isc: delivering S,2,3,ALERT
Hierarchy: creating instance(3) of 'B' interface
Isc: delivering S,2,4,ALERT
Hierarchy: creating instance(4) of 'B' interface
Isc: delivering S,2,5,ALERT
Hierarchy: creating instance(5) of 'C' interface
Isc: delivering S,2,6,ALERT
Hierarchy: creating instance(6) of 'C' interface
```

5.4 Ejemplo 4: instanciación múltiple de un esquema con reutilización

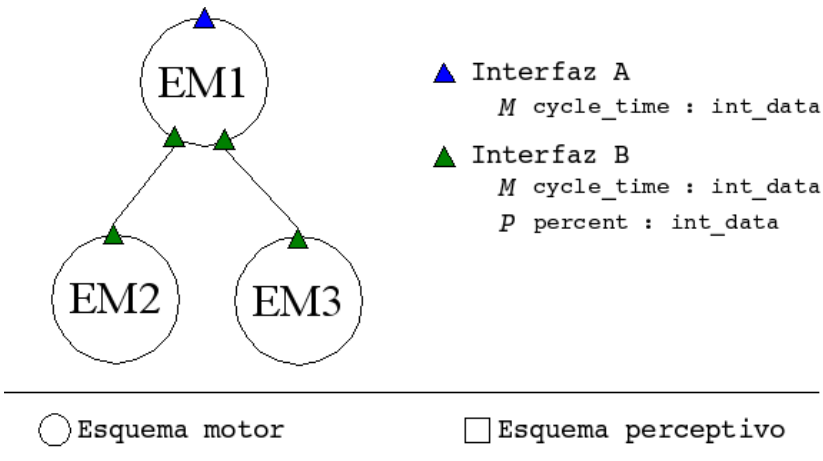


Figura 5.8: *Jerarquía del ejemplo 4*

Ya hemos visto prácticamente todos los mecanismos que *jde+* ofrece a un desarrollador de aplicaciones robóticas. Si recapitulamos, veremos que son apenas media docena de métodos, debido en parte al objetivo principal de *jde+*, la simplicidad. Pero aun nos falta presentar un mecanismo, la reutilización de esquemas. Este mecanismo es automático, y tan solo requiere opcionalmente la implementación de un método de reinicialización, como vimos en el apartado 4.2.7.

El ejemplo que usaremos para demostrar la reutilización es similar al usado en el ejemplo 2. EM1 será el esquema raíz, que tendrá dos hijos motores, aunque esta vez serán idénticos y no se solapará su activación. Es decir, tendremos una jerarquía formada con raíz EM1, y dos hijos EM2, que distinguiremos con los apodos B1 y B2. La figura 5.8 presenta esta jerarquía.

5.4.1 Esquemas

El esquema EM1 tiene una iteración que activa o desactiva a cada hijo en función de un contador. Su código no hace nada especial para usar la reutilización, es el sistema el que lo hace. Dicho código lo podemos ver a continuación:

```
void EM1::iteration(){
  int_data cycle_time(&get_modulation("cycle_time"));
  double t = gettimeofdaysec() - get_system_start_time();
  static int i;

  set_cycle_time(cycle_time);

  switch(i){
   case 0:
    /*wakeup child B1*/
    cout << "EM1(" << t << "): Waking Up child B1\n";
    set_child_state("B1",ALERT);
    /*modulate cycle_time*/
    cout << "EM1(" << t << "): modulating child B1 cycle_time\n";
    modulate_child("B1","cycle_time",
   cycle_time);/*set cycle_time same as mine*/
    break;
   case 5:
    /*sleep child B1*/
    cout << "EM1(" << t << "): Sleeping child B1\n";
    set_child_state("B1",SLEPT);
    break;
   case 6:
    /*wakeup child B2*/
    cout << "EM1(" << t << "): Waking Up child B2\n";
    set_child_state("B2",ALERT);
    /*modulate cycle_time*/
    cout << "EM1(" << t << "): modulating child B2 cycle_time\n";
    modulate_child("B2","cycle_time",
   cycle_time);/*set cycle_time same as mine*/
    break;
   case 11:
    /*sleep child B2*/
    cout << "EM1(" << t << "): Sleeping child B2\n";
```

```
    set_child_state("B2",SLEPT);
    break;
  }
  cout << "EM1(" << t << "): I'm alive with sid: "
       << sid_to_str(my_sid()) << "\n";
  cout << "EM1(" << t << "): executing iteration\n";
  if (i<11)
    i++;
  else
    i = 0;
}
```

Como se puede apreciar, el contador `i` marca el ritmo de las actuaciones. Cuando vale 0 despertamos al hijo B1, cuando vale 5 lo dormimos, cuando vale 6 despertamos al hijo B2, momento en el que se produce la reutilización, y por último cuando vale 11 dormimos a B2 y volvemos a 0. Cada esquema hijo permanece despierto cinco iteraciones.

La reutilización de esquemas surge cuando se quiere activar un esquema, y previamente uno del mismo tipo se ha dejado de usar (se ha dormido). Esto es debido a que el sistema guarda las instancias despues de desactivarse, y las reutiliza cuando se requiere una de ese tipo en cualquier punto de la jerarquía.

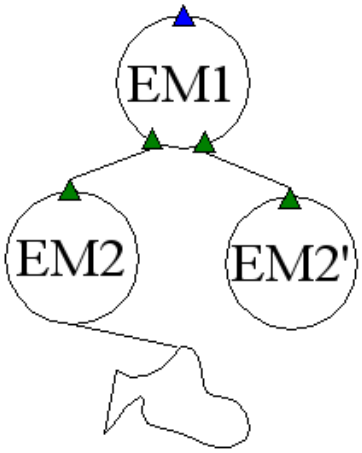
Por su parte, el código del esquema EM2 implementa una iteración que pinta un mensaje por pantalla,en el que se muestra el valor de una variable que representa el estado interno del esquema. Además implementa una función de reinicio del estado interno, que pone a cero el estado interno, que el sistema ejecutará automáticamente cuando crea conveniente. Su código es el siguiente:

```
void EM2::iteration(){
  int_data cycle_time(&get_modulation("cycle_time"));
  double t = gettimeofdaysec() - get_system_start_time();

  set_cycle_time(cycle_time);

  cout << "EM2(" << t << "): I'm alive with sid: "
       << sid_to_str(my_sid()) << "\n";
  cout << "EM2(" << t << "): executing iteration. Internal state(i="
       << i++ << ")\n";
}

void EM2::reset_handler(){
  double t = gettimeofdaysec() - get_system_start_time();

  cout << "EM2(" << t << "): reseting internal state. Old value(i="
       << i << ")\n";
  i = 0;
}
```

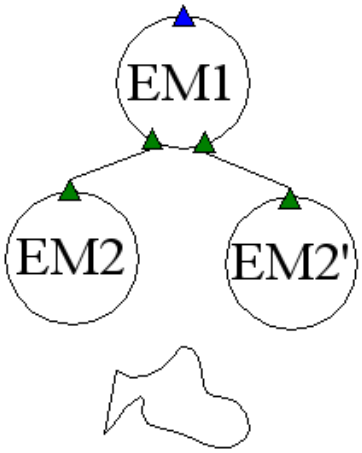
5.4.2 Ejecución

La ejecución de este ejemplo produce dos instancias de esquema, una que es la raíz y otra que se comparte entre los dos hijos. Siendo precisos, lo que realmente se reutiliza es lo particular, es decir, el objeto `schemaimplementation` que contiene la funcionalidad

```
EM1: Creating schema of interface 'A'
EM1(0.308857): Waking Up child B1
EM1(0.308857): modulating child B1 cycle_time
EM1(0.308857): I'm alive with sid: ROOT
EM1(0.308857): executing iteration
EM2: Creating schema of interface 'B'
EM2(0.416755): reseting internal state. Old value(i=0)
EM2(0.483611): I'm alive with sid: 3
EM2(0.483611): executing iteration. Internal state(i=0)
EM1(1.31248): I'm alive with sid: ROOT
EM1(1.31248): executing iteration
EM2(1.48746): I'm alive with sid: 3
EM2(1.48746): executing iteration. Internal state(i=1)
EM1(2.31633): I'm alive with sid: ROOT
EM1(2.31633): executing iteration
EM2(2.49131): I'm alive with sid: 3
...
```




```
...
EM1(5.32787): Sleeping child B1
EM1(5.32787): I'm alive with sid: ROOT
EM1(5.32787): executing iteration
```




```
EM1(6.33172): Waking Up child B2
EM1(6.33172): modulating child B2 cycle_time
EM1(6.33172): I'm alive with sid: ROOT
EM1(6.33172): executing iteration
EM2(6.33245): reseting internal state. Old value(i=5)
EM2(6.40377): I'm alive with sid: 4
EM2(6.40377): executing iteration. Internal state(i=0)
EM1(7.33558): I'm alive with sid: ROOT
EM1(7.33558): executing iteration
EM2(7.40756): I'm alive with sid: 4
EM2(7.40756): executing iteration. Internal state(i=1)
EM1(8.33942): I'm alive with sid: ROOT
```

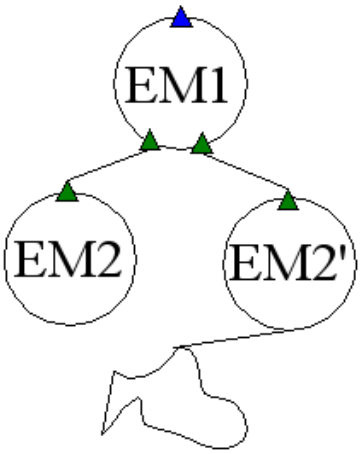


Figura 5.9: *Ejecución del ejemplo 4*

del esquema. Para verlo, en la figura se muestan las trazas generadas por los esquemas junto a gráficos aclaratorios.

En las primeras iteraciones EM1 activa a su hijo B1. Este se materializa en el esquema EM2, que instancia lo particular, representado en el dibujo con la forma irregular. Mientras este ejecuta sus iteraciones vemos como va evolucionando su estado interno (0,1,2,...). En la quinta iteración EM1 decide dormir a su hijo B1. En este momento el sistema lo detecta, y desvincula a EM2 de lo particular, almacenandolo para una futura reutilización. En la sexta iteración EM1 despierta a su hijo B2, que corresponde a EM2'. Cuando este intenta obtener su parte propia, el sistema en vez de construir una nueva instancia reusa la que guardó y reinicia su estado interno. Como podemos ver en la traza del final, el estado interno antiguo es el que dejo EM2 tras cinco iteraciones, y despues del reinicio vuelve a 0. Esto se muestra en la figura 5.9.

Capítulo 6

Conclusiones y trabajos futuros

En este último capitulo se muestran las conclusiones extraidas del desarrollo de este proyecto, comentando los puntos más relevantes que han permitido el logro del trabajo, y las principales dificultades encontradas. También hablamos, de las lineas de trabajos futuros que pueden derivarse de este proyecto para su mejora y ampliación.

6.1 Conclusiones

Como conclusión principal extraemos que se ha logrado la implementación del sistema jde+ con los objetivos y requisitos mostrados en el capítulo 2. Capaz de cargar una colección de esquemas de manera dinámica, e iniciar el comportamiento pedido, instanciando los esquemas requeridos para ello. Capaz de gestionar todas las necesidades de cada esquema, hasta la desactivación del comportamiento, momento en el cual el sistema detiene a cada esquema. El diseño del sistema se ha construido minuciosamente, con el resultado de un diseño muy cuidado, claro y coherente. Su aplicación al pie de la letra en la implementación, ha permitido la construcción del sistema *jde+*, un sistema complejo con un núcleo de aproximadamente 4000 líneas de código. Todo el código fuente del sistema jde+ se encuentra disponible en el cdrom adjunto y en la web del grupo *http://gsyc.escet.urjc.es/robotica* bajo licencia GPL.

El sistema implementado permite la inserción de esquemas de manera muy simple, tal y como se pedía en el subobjetivo O-1. Para ello el sistema aporta una clase base (`schemaimplementation`) que incluye todos los mecanismos que necesita un esquema, permitiendo que el desarrollador de aplicaciones robóticas se centre en la lógica de su aplicación. Además, se ha implementado con éxito la carga dinámica de código, por lo que una vez construidos los nuevos esquema, estos pueden cargarse sin requerir ni una modificación en el sistema.

Los mecanismos de comunicación implementados permiten ceñirnos a las reglas expuestas en JDE, de manera que un esquema sólo puede comunicarse con su padre y con sus hijos. Además, se abre la posibilidad de ejecutar determinadas acciones en función de los eventos producidos, como la llegada de datos, o el cambio de estado. Esto cumple el subobjetivo O-2 que pretendía un sistema con mecanismos potentes.

La creación de instancias múltiples requerida en O-3, se ha solventado de manera eficaz con la idea "*esquema = componente*". Esta idea nos permite ver a un esquema como un ente compacto y autocontenido que usa los servicios del sistema para interactuar con el resto de esquemas. El principal servicio usado es la comunicación, que por ser por paso de mensajes, refuerza más la faceta de componente de un esquema. Así, la materialización de un esquema en el sistema consiste en la creación de una o múltiples instancias de estos componentes, que aun siendo del mismo tipo tienen un contexto propio. Esta característica es una de las más importantes del sistema, pues

abre la puerta a nuevas jerarquías que anteriormente no era posible materializar en jde.c.

Las ideas introducidas en el diseño en la linea del subobjetivo O-4, concluyen en un diseño preparado para distribuir esquemas en diferentes nodos de computo. Si bien, es cierto que no se ha implementado solución alguna a este respecto, ya que se escapaban del contexto de este proyecto.

El rendimiento del sistema se ha mostrado excelente en los ejemplos desarrollados en el capítulo 5, tanto computacionalmente, como temporalmente. Es de esperar que en comportamientos más complejos (5-20 esquemas) el consumo aumente, debido en parte a la gran cantidad de mensajes que se intercambiarán. Aun así se espera que los consumos se mantengan muy por debajo de los usados por la lógica de los comportamientos implementados. En cuanto al consumo en comportamientos muy complejos (+20 esquemas), es difícil preverlo, aunque es posible que el sistema requiera optimizaciones (estructuras de datos más eficientes, evitar copia de datos, ...) para mejorarlo.

Las principales dificultades encontradas en el desarrollo del proyecto han sido técnicas, solucionándose en todos los casos con el uso de manuales e información obtenida de Internet. En concreto, las dificultades encontradas fueron la compilación del sistema con `gcc`, y la depuración del sistema con `gdb`.

En cuanto a la compilación del sistema, resultó complicada debido a la gran cantidad de clases y a las ínter-dependencias entre ellas. También se tuvo que investigar el modo de compilar las partes, para que las cargadas dinámicamente (los esquemas) tuviesen visibilidad de todos los símbolos[1] del sistema y viceversa. Las soluciones se extrajeron del manual del compilador (gcc(1)), y de varios foros de debate hayados en Internet.

La otra dificultad encontrada fue la depuración del sistema en la fase de pruebas. El sistema jde+ es un sistema concurrente, lo que complica enormemente la detección de errores y la depuración, dado que existen múltiples flujos de ejecución concurrentes. A los errores típicos de cualquier desarrollo, se unen los errores por condiciones de carrera, generalmente impredecibles e irreproducibles. Para la depuración se uso el potente depurador `gdb`, que permite explorar cada flujo de manera independiente, su manual (gdb(1)) resultó de gran ayuda y una vez más la información obtenida en Internet.

6.2 Trabajos futuros

El desarrollo de este proyecto ha desembocado en el sistema jde+ completamente funcional, pero que es susceptible de muchas mejoras y ampliaciones. En esta sección enumeramos algunos de los posibles trabajos futuros que se podrían realizar al sistema, que en algunos casos podrían ser objeto de proyectos completos.

Una de las lineas de futuro podría ser el soporte para ejecución remota de esquemas, permitiendo así obtener mayor capacidad de cómputo. El diseño de jde+ lo ha tenido en cuenta (apartado 4.2.9), separando la parte general de la particular de un esquema. La implementación de este soporte en *jde+* abriría las puertas a nuevos comportamientos, demasiado complejos computacionalmente como para poder ejecutar en un solo nodo.

La construcción de un comportamiento requiere una serie de *esquemas de servicio*, que interactúan con la plataforma robótica para manejar sus dispositivos (sonar, láser, motores, ...). Haciendo una equivalencia, podríamos ver a *jde+* como un sistema operativo (SO) y a los esquemas de servicio como los programas básicos de tal SO. Una linea de trabajo futuro, es la implementación de estos esquemas de servicio, para que el usuario pueda usarlos como una facilidad más del sistema, preocupándose de los esquemas de más alto nivel. Esto implica desarrollar una batería de esquemas, que

[1] En el sentido de variables, objetos, funciones, ...

sirvan de interfaz con las plataformas y dispositivos, que posee el grupo de robótica de la URJC.

Otro trabajo interesante que podría realizarse, es una herramienta para la depuración de alto nivel de una jerarquía de esquemas. La idea, sería conectar una especie .ºsciloscopio virtual"que nos permitiese ver en tiempo real la interacción de unos esquemas con otros, registrando sus comunicaciones para posteriormente poder analizar las causas de un fallo. El sistema actual, centraliza toda la comunicación en el ya comentado `isc`, por lo que bastaría extraer la información de allí.

Y por último, otro tema que podría explorarse es el auto-ajuste del tiempo de ciclo de un esquema. Como se comentó en su momento, uno de los parámetros de modulación principales de un esquema es su tiempo de ciclo, que controla lo que debería tardar una iteración. Este parámetro, lo ajusta el usuario generalmente de manera experimental. El efecto de un tiempo de ciclo demasiado grande, es que el esquema pierde datos del nivel inferior, o no los suministra con la suficiente celeridad al nivel superior. El efecto de un tiempo de ciclo demasiado pequeño, es que se leen datos repetidos del nivel inferior, o se suministran demasiados datos al nivel superior. Así, resulta crucial ajustar este tiempo de ciclo lo mejor posible. En el sistema jde.c, no hay ninguna solución simple, puesto que el sistema no gestiona directamente la comunicación ínter-esquema. Sin embargo, en jde+, podríamos lograr un ajuste automático en algunos casos. En el caso de esquemas base generadores de percepciones, como un esquema sonar o uno láser, nos interesa poder ajustar el tiempo ciclo para controlar el flujo de datos (por ejemplo, 10 medidas por segundo implican un tiempo de ciclo de 100ms). Evidentemente en estos casos, el ajuste automático no es deseable. Pero en el caso de esquemas con iteraciones deterministas, es decir, que con los mismos datos producen los mismos resultados, podríamos hacer que se ejecutase la iteración sólo cuando hubiese datos nuevos (modulaciones o percepciones), evitando iteraciones innecesarias con datos repetidos. El efecto conseguido, sería el ajuste de tiempo de ciclo óptimo. El resto de posibles casos se deberían estudiar cuidadosamente, para no producir ínter-bloqueos entre los esquemas.

Bibliografía

[1] The computer language shootout benchmarks. http://shootout.alioth.debian.org/.

[2] Posix threads programming. http://www.llnl.gov/computing/tutorials/workshops/workshop/pthreads/MAIN.html.

[3] Philip E. Agre and David Chapman. Pengi: an implementation of a theory of activity. In *Proceedings of 6th AAAI National Conference on Artificial Intelligence*, pages 268–272, Seattle, WA, 1987. Morgan Kaufmann.

[4] Philip E. Agre and David Chapman. What are plans for? In Pattie Maes, editor, *Designing Autonomous Agents: theory and practice from Biology to Engineering and Back*, pages 17–34. MIT Press, 1990.

[5] Ronald C. Arkin. Motor schema-based mobile robot navigation. *International Journal of Robotics Research*, 8(4):92–112, August 1989.

[6] Ronald C. Arkin. Reactive robotic systems. In M. Arbib, editor, *Handbook of the brain theory and neural networks*, pages 793–796. MIT Press, 1995.

[7] Ronald C. Arkin. *Behavior based robotics*. MIT Press, 1998.

[8] Rodney A. Brooks. A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, 2(1):14–23, March 1986.

[9] Rodney A. Brooks. A hardware retargetable distributed layered architecture for mobile robot control. In *Proceedings of the 1987 International Conference on Robotics and Automation*, pages 106–110, Raleigh-NC, March 1987. Computer Society Press.

[10] Rodney A. Brooks. Intelligence without reason. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 569–595, 1991.

[11] Karel Capek. *RUR: Rossum's universal robots*. 1923.

[12] Ève Coste-Maniére and Reid G. Simmons. Architecture, the backbone of robotic systems. In *Proceedings of the 2000 IEEE International Conference on Robotics and Automation*, pages 67–72, San Francisco, CA (USA), April 2000.

[13] R. James Firby. An investigation into reactive planning in complex domains. In *Proceedings of the 6th AAAI National Conference on Artificial Intelligence*, pages 202–206, Seattle, WA, 1987.

[14] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design patterns cd: Elements of reusable object-oriented software, 2003.

[15] Lía García Pérez. *Navegación autónoma de robots en agricultura: un modelo de agentes*. PhD thesis, Universidad Complutense de Madrid, 2004.

[16] Kurt Konolige and Karen L. Myers. The Saphira architecture for autonomous mobile robots. In David Kortenkamp, R. Peter Bonasso, and Robin Murphy, editors, *Artificial Intelligence and Mobile Robots: case studies of successful robot systems*, pages 211–242. MIT Press, AAAI Press, 1998. ISBN: 0-262-61137-6.

[17] John E. Laird, Allen Newell, and Paul S. Rosenbloom. SOAR: an architecture for general intelligence. *Artificial Intelligence*, 33:1–64, 1987.

[18] Maja J. Mataric. Situated robotics. In Lynn Nadel, editor, *Encyclopedia of Cognitive Science*. Nature Publishers Group, London UK, 2002.

[19] Esther García Morata. Construcción de un teleoperador para el robot eyebot. Proyecto fin de carrera, Universidad Carlos III, 2002.

[20] Robin R. Murphy. Dempster-shafer theory for sensor fusion in autonomous mobile robots. *IEEE Transactions on Robotics and Automation*, 14(2):197–206, April 1998.

[21] Robin R. Murphy. *Introduction to AI robotics*. MIT Press, 2000.

[22] N.J. Nilsson. A mobile automaton: an application of artificial intelligence techniques. In *Proceedings of the 1st International Joint Conference on Artificial Intelligence IJCAI*, pages 509–520, Washington, (USA), 1969.

[23] David W. Payton. Internalized plans: a representation for action resources. *Robotics and Autonomous Systems*, 6:89–103, 1990.

[24] José M. Cañas Plaza. *Jerarquía Dinámica de Esquemas para la generación de comportamiento autónomo*. PhD thesis, Universidad Politécnica de Madrid, 2003.

[25] Douglas Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann. *Pattern-Oriented Software Architecture: Volume 2*. 2001.

[26] Miguel Schneider Fontán. *Planificación basada en percepción activa para la navegación de un robot móvil*. PhD thesis, Universidad Complutense de Madrid, 1996.

[27] Francisco Serradilla. *Arquitectura cognitiva basada en el gradiente sensorial y su aplicación a la robótica móvil*. PhD thesis, Universidad Politécnica Madrid, 1997.

[28] Reid Simmons, Richard Goodwin, Karen Zita Haigh, Sven Koening, and Joseph O'Sullivan. A layered architecture for office delivery robots. In *Proceedings of the ACM International Conference on Autonomous Agents*, pages 245–252, Marina del Rey (USA), feb 1997.

[29] Reid G. Simmons. Structured control for autonomous robots. *IEEE Journal of Robotics and Automation*, 10(1):34–43, February 1994.

[30] Anthony Stentz, Cristian Dima, Carl Wellington, Herman Herman, and David Stager. A system for semi-autonomous tractor operations. *Autonomous Robots*, 13:87–104, 2002.

[31] Alan M. Turing. Computer machinery and intelligence. *Mind*, 59(236):433–460, October 1950.