UNIVERSIDAD POLITÉCNICA DE VALENCIA

DEPARTAMENTO DE SISTEMAS INFORMÁTICOS Y COMPUTACIÓN

TESIS DOCTORAL

Meta-razonamiento en Agentes con Restricciones Temporales Críticas

Realizada por:

D. Carlos Carrascosa Casamayor

Dirigida por:

Dr. D. Vicente J. Botti Navaro

Dr. D. Andrés M. Terrasa Barrena

A mis padres,
por su cariño, interés y apoyo constante

"Sólo el soñador emponzoña todos sus días,
Soportando más pesar del que merecen sus pecados"
– "Caída de Hyperion", John F. Keats, (1795 - 1821)

Resumen

El paradigma de agentes/sistemas multi-agente es uno de los modelos computacionales de mayor relevancia de los últimos tiempos, habiendo dado lugar a múltiples investigaciones y aplicaciones concretas. Este modelo computacional tiene como objetivo la construcción de sistemas que se enfrenten a situaciones mostrando ciertas características propias de un ser humano, tales como *inteligencia*, *reactividad*, *pro-actividad*, ...

De entre todas las variedades de tipos de agente que se pueden definir, el trabajo realizado se centra en aquellos agentes que deben trabajar en un entorno con restricciones temporales críticas, es decir, donde existen ciertos problemas a los que el agente debe dar respuesta antes de que pase un determinado tiempo o las consecuencias serán catastróficas. En un agente de este tipo es fundamental tratar de conseguir un uso óptimo del tiempo de procesador, recurso más importante en esta clase de sistemas.

Es por esto que resulta relevante conseguir que dicho agente sea capaz de dedicar su tiempo de procesador a aquello que sea necesario de acuerdo a la situación en la que se encuentre. Para conseguir esta adaptación es fundamental que el agente sea capaz de razonar sobre su propio proceso de razonamiento, es decir, *meta-razonar*, siempre teniendo en cuenta que este proceso de meta-razonamiento va a consumir también tiempo de procesador.

De esta manera, el objetivo de este trabajo es el estudio de las capacidades necesarias para poder incorporar la habilidad de meta-razonar a un agente con restricciones temporales críticas, así como la incorporación a una arquitectura de agente concreta, la de agente ARTIS. Después del estudio comentado, se llegó a la conclusión de que para poder incorporar la habilidad de meta-razonamiento a un agente con restricciones temporales críticas era necesario incluir al agente las siguientes capacidades: detección de situaciones significativas, adaptar su comportamiento, adaptar el proceso de razona-

miento del agente teniendo en cuenta las restricciones temporales del mismo y aprendizaje.

Si se quiere que el agente sea capaz de adaptarse, debe ser capaz de detectar en qué momento, en qué situación debe de realizar esa adaptación, ese cambio. Es por ello que la capacidad para detectar situaciones significativas es fundamental, pues sin ella no tiene sentido la adaptación.

En cuanto a la capacidad de adaptar su comportamiento, el agente dispondrá de diversos comportamientos, pudiendo activar en cada momento aquel que sea el adecuado para la situación en la que se encuentra.

Para poder adaptar el proceso de razonamiento del agente teniendo en cuenta las restricciones temporales del mismo se definen los siguientes tres conceptos como variables, permitiendo con su variación este tipo de adaptación:

- *Reactividad*:

  Este concepto es una característica básica de un agente. La novedad consiste, tal y como se ha comentado, en considerar este concepto como variable, definiendo un *Grado de Reactividad*, que permita que el agente se comporte como más reactivo o más deliberativo de acuerdo a su valor.

- *Introspección*:

  La psicología general distingue entre *extrospección* e *introspección* como métodos de observación que en el primer caso se basa en el análisis de los contenidos que se derivan de los sentidos, mientras que en el segundo se basa en los contenidos de la conciencia del propio individuo.

  De forma similar, el presente trabajo plantea esta distinción al estudiar el proceso de deliberación de un agente, distinguiendo entre la deliberación motivada como respuesta a cambios en el entorno y la deliberación sobre los objetivos propios del agente (en base a la pro-actividad del mismo). De esta manera, se introduce el *Grado de Introspección* como un valor entre 0 y 1 que indica la división en el tiempo de proceso que se realiza para indicar el tiempo que dedica el agente para atender a cambios en el entorno frente al tiempo que dedica a deliberar, a la *introspección*. Conforme el grado de introspección tiene un valor más cercano a 0, el agente es más sensible a los cambios en el entorno, mientras que

si el grado de introspección está más cercano a 1, está más centrado en sí mismo, es más introspectivo.

- *Concentración*:

  Este concepto se define, desde el punto de vista de la psicología general, como la capacidad de centrarse en unos estímulos determinados, desechando aquellos que no están relacionados con ellos, con el fin de focalizar la atención y optimizar la reflexión. De igual manera, la psicología general considera el concepto de atención como la concentración selectiva de la actividad mental que implica un aumento de eficiencia sobre una tarea determinada, además de una inhibición perceptiva y cognitiva de las demás actividades.

El objetivo de esta capacidad de adaptación del proceso de razonamiento es, por lo tanto, el de aplicar los tres conceptos anteriores a la arquitectura de agente. Así, una forma de aumentar la eficiencia en el uso del tiempo de procesamiento en un agente consiste en poder focalizar dicho proceso, es decir, reducir el ámbito sobre el cual va a versar su proceso de razonamiento, para así emplear el tiempo de procesamiento disponible en aquello que sea de relevancia a la situación actual (variar la concentración del mismo). De esta manera, se define un *foco de atención* como un subconjunto de creencias del agente que resultan significativas en alguna situación concreta. Tal y como presenta el concepto de concentración, estos subconjuntos de creencias se utilizarán para optimizar el proceso de razonamiento (y meta-razonamiento) priorizando el razonamiento acerca del foco o focos de atención activos. El mecanismo de focalización no tiene porque centrarse a un único foco, sino que pueden existir diversos focos de atención simultáneos, cada uno de ellos con un grado de atención distinto (de acuerdo a la importancia relativa de ese foco con relación a los demás focos activos).

En cuanto a la última capacidad, la de aprendizaje, le debe permitir al agente refinar, utilizando su propia experiencia, aquellos parámetros que controlan su comportamiento, hasta el extremo de ser capaz de aprender nuevas formas de comportarse, nuevos comportamientos a añadir a aquellos especificados por el diseñador.

Tal y como se ha comentado previamente, estas capacidades que conforman la habilidad de meta-razonar se han llevado a cabo en la arquitectura de agente ARTIS. Para ello, se han extendido los diferentes modelos de di-

cha arquitectura que permiten la aproximación a la misma desde diferentes niveles de abstracción: modelo formal (definición desde un nivel abstracto), modelo de usuario (definición desde el nivel del diseñador de agentes ARTIS) y modelo de sistema (definición desde el nivel directamente ejecutable).

Resum

El paradigma d'agents/sistemes multi-agent és un dels models computacionals de major rellevància dels últims temps, havent-hi donat lloc a múltiples investigacions i aplicacions concretes. Este model computacional té com objectiu la construcció de sistemes que s'enfronten a situacions mostrant certes característiques propies d'un esser humà, tals com *intel·ligència*, *reactivitat*, *proactivitat*, ...

D'entre totes les varietats de tipus d'agent que es poden definir, el treball realitzat es centra en aquells agents que han de treballar en un entorn amb restriccions temporals crítiques. És a dir, on hi ha certs problemes als que l'agent ha de donar resposta abans que passe un determinat temps o les conseqüències seran catastròfiques. En un agent d'este tipus és fonamental tractar d'aconseguir un ús òptim del temps de processador, el qual és el recurs més important en esta classe de sistemes.

És per açò que resulta rellevant aconseguir que eixe agent siga capaç de dedicar el seu temps de processador a allò que siga necessari d'acord a la situació en què es trobe. Per a aconseguir esta adaptació és fonamental que l'agent siga capaç de raonar sobre el seu propi procés de raonament, és a dir, *meta-raonar*, sempre tenint en compte que este procés de meta-raonament consumirà també temps de processador.

D'esta manera, l'objectiu d'este treball és l'estudi de les capacitats necessàries per a poder incorporar l'habilitat de meta-raonar a un agent amb restriccions temporals crítiques, així com la incorporació a una arquitectura d'agent concreta, la de l'agent ARTIS. Després de l'estudi comentat, es va arribar a la conclusió que per a poder incorporar l'habilitat de meta-raonament a un agent amb restriccions temporals crítiques era necessari incloure a l'agent les següents capacitats: detecció de situacions significatives, adaptar el seu comportament, adaptar el procés de raonament de l'agent tenint en compte

les restriccions temporals d'este i, finalment, aprenentatge.

Si es vol que l'agent siga capaç d'adaptar-se, ha de ser capaç de detectar en quin moment, en quina situació deu realitzar eixa adaptació, eixe canvi. És per això que la capacitat per a detectar situacions significatives és fonamental, doncs sense ella no té sentit l'adaptació.

En quant a la capacitat d'adaptar el seu comportament, l'agent disposarà de diversos comportaments, podent activar en cada moment aquell que siga l'adequat per a la situació en què se troba.

Per a poder adaptar el procés de raonament de l'agent tenint en compte les restriccions temporals d'este es definixen els següents tres conceptes com a variables, permetent amb la seua variació este tipus d'adaptació:

- *Reactivitat*:

  Este concepte és una característica bàsica en un agent. La novetat consisteix, tal com s'ha comentat, a considerar este concepte com a una variable, definint un *Grau de Reactivitat*, que permeta que l'agent es comporte com més reactiu o més deliberatiu d'acord al seu valor.

- *Introspecció*:

  La psicologia general distingeix entre *extrospecció* i *introspecció* com a mètodes d'observació que en el primer cas es basa en l'anàlisi dels continguts que es deriven dels sentits, mentre que en el segon es basa en els continguts de la consciència del propi individu.

  De forma semblant, el present treball planteja esta distinció al estudiar el procés de deliberació de un agent, distingint entre la deliberació motivada com a resposta a canvis en l'entorn i la deliberació sobre els objectius propis de l'agent (en base a la seua proactivitat). D'esta manera, s'introdueix el *Grau d'Introspecció* com un valor entre 0 i 1 que indica la divisió en el temps de procés que se realitza per a indicar el temps que dedica l'agent per a atindré a canvis en l'entorn enfront del temps que dedica a deliberar a la *introspecció*. Conforme el grau d'introspecció té un valor mes pròxim a 0, l'agent és mes sensible als canvis en l'entorn, mentre que si el grau d'introspecció està mes pròxim a 1, està més centrat en si mateix, és més introspectiu.

- *Concentració*:

> Este concepte es defineix, des del punt de vista de la psicologia general, com la capacitat de centrar-se en uns estímuls determinats, rebutjant aquells que no estan relacionats amb ells, a fi de focalitzar l'atenció i optimar la reflexió. D'igual manera, la psicologia general considera el concepte d'atenció com la concentració selectiva de l'activitat mental que implica un augment d'eficiència sobre una tasca determinada, a més d'una inhibició perceptiva i cognitiva de les altres activitats.

L'objectiu d'esta capacitat d'adaptació del procés de raonament és, per tant, el d'aplicar els tres conceptes anteriors a l'arquitectura d'agent. Així, una forma d'augmentar l'eficiència en l'ús del temps de processament en un agent consisteix a poder focalitzar eixe procés, és a dir, reduir l'àmbit sobre el qual versarà el seu procés de raonament, per a així utilitzar el temps de processament disponible en allò que siga de rellevància a la situació actual (variar la concentració d'este). D'esta manera, es defineix un *focus d'atenció* com un subconjunt de creences de l'agent que resulten significatives en alguna situació concreta. Tal com presenta el concepte de concentració, aquests subconjunts de creences s'utilitzaran per a millorar el procés de raonament (i meta-raonament) prioritzant el raonament sobre el focus o focus d'atenció actius. El mecanisme de focalització no té perquè centrar-se a un únic focus, sinó que poden existir diversos focus d'atenció simultanis, cadasqú d'ells amb un grau de atenció distint (d'acord amb la importància relativa d'eixe focus amb relació als altres focus actius).

En quant a l'última capacitat, la d'aprenentatge, li permet a l'agent refinar, utilitzant la seua pròpia experiència, aquells paràmetres que controlen el seu comportament, fins a l'extrem de ser capaç d'aprendre noves formes de comportar-se, nous comportaments a afegir a aquells especificats pel dissenyador.

Tal com s'ha comentat prèviament, estes capacitats que conformen l'habilitat de meta-raonar s'han dut a terme en l'arquitectura d'agent ARTIS. Per eixe motiu, s'han estés els diferents models d'eixa arquitectura que permeten l'aproximació a esta des de diferents nivells d'abstracció: model formal (definició des d'un nivell abstracte), model d'usuari (definició des del nivell del dissenyador d'agents ARTIS) i model de sistema (definició des del nivell directament executable).

Abstract

Over the last few years, the agent/multi-agent system paradigm is one of the most relevant computing models. It has produced a lot of specific research and applications. The objective of this computational model is to build systems that will face situations showing some human being features, such as *intelligence*, *reactivity*, *pro-activity*, ...

There are a lot of different kinds of agents. The work that has been done is focused in agents working in an environment with critical temporal restrictions. In these environments, there are some problems to be solved by the agent before some time has passed, otherwise the consequences may be catastrophic. It is fundamental for an agent like this to get an optimal usage of the processor time. In fact, this is the most important resource in these systems.

That is the reason why it is relevant to get the agent to dedicate its processor time to what is really necessary in each situation. The agent being able to reason about its own reasoning process is fundamental to get this adaptation. This is called *meta-reasoning*, and when used, it has to be aware that this process also consumes processor time.

So, the purpose of this work is the study of the necessary capabilities to incorporate the meta-reasoning ability to an agent with critical temporal restrictions. Moreover, to incorporate it to an specific agent architecture, the ARTIS agent architecture. The conclusion of this study was that the capabilities needed to incorporate the ability of meta-reasoning to an agent with critical temporal restrictions are: to detect significant situations, to adapt its behaviour, to adapt its reasoning process taking into account its temporal restrictions, and learning.

To get an agent able to adapt, it must be able to detect the moment, the situation, when this adaption must be carried out. This is the reason why the

detection of significant situations is an essential capability, because without it the adaption is meaningless.

Regarding the capability to adapt its behaviour, the agent will have several behaviours, being able to activate, at each moment, the proper one to its situation.

The following three concepts are defined as variable ones to allow the agent to adapt its reasoning process taking into account its temporal restrictions:

- *Reactivity*:

  This concept is a basic feature of an agent. As it has been commented, the new approach presented here is to consider this concept as variable, defining a *Reactivity Degree*. This degree allows the agent to behave as more reactive or more deliberative according to it.

- *Introspection*:

  Generic Psychology distinguishes between *extrospection* and *introspection* as observation methods. The first one is based on the sense-derived contents analysis. The second one is based on the contents of the awareness of the individual itself.

  In a similar way, the present work raises this distinction when studying the agent's deliberation process. This work distinguishes between the deliberation motivated as an answer to environment changes and the deliberation about the agent's own objectives (according to its proactivity). In this way, the *Introspection Degree* is introduced as a value between 0 and 1 indicating the split between the process time dedicated to attend to environment changes and the process time dedicated to deliberate, to the *introspection*. As the introspection degree's value gets closer to 0, the agent becomes more sensitive to environment changes, whilst as the introspection degree's value gets closer to 1, the agent becomes more self-centered (more introspective).

- *Concentration*:

  This concept is defined, from Generic Psychology point of view, as the capacity of focusing on some specific stimuli, avoiding the ones not related to them, so that the attention is focalized and the reflection is optimized. On a similar way, Generic Psychology considers the attention

concept as the selective concentration of the mental activity that implies an increase of the efficiency over an specific task, as well as a perception and cognition inhibition of the rest of activities.

So, the purpose of this capability to adapt the reasoning process is to apply the preceding three concepts to the agent architecture. Thus, one way to increase the efficiency of the agent in the use of the process time is to be able to focus such process; that is, to reduce the field of its reasoning process to use the available process time to what is relevant to each situation (to modify the agent's concentration). In this way, an *attention focus* is defined as a subset of the agent's believes that are significant in an specific situation. As the concept of concentration presents, these believes subsets will be used to optimize the reasoning (and meta-reasoning) process by means of prioritizing the reasoning about the active attention focus or focuses. The focalization mechanism does not have to be reduced to only one focus, but there can exist several attention focus at the same time, each one with a different attention degree (according to the relative importance of this focus with the other active focuses).

Regarding to the last capability, learning, it must allow the agent to refine, using its own experience, its behaviour parameters. The agent should be even able to learn new ways of behaving to add to the ones specified by the designer.

As it has been commented before, these capabilities forming the meta-reasoning ability have been carried out in the ARTIS agent architecture. To make it so, all the different models of such architecture (corresponding to different abstraction levels) have been extended: formal model (abstract level definition), user model (definition from the ARTIS agent designer's level) and system model (definition from the directly executable level).

Agradecimientos

Ha sido un largo camino para llegar hasta aquí. Un camino que comenzó con seis proyectos finales de carrera, seis proyectos de los que dos (que más bien eran uno) fueron la semilla de lo que ahora es esta tesis. Mi primer agradecimiento por tanto debe ir para aquellos que comenzaron conmigo el camino, pero en especial para Toni, aquel que compartió conmigo aquel proyecto final de carrera y se apartó del camino (de Vera). Hay, sin embargo, dos compañeros (Miguel y Vicente) que sí que siguieron conmigo todo el camino, desde aquellos días, y sin los cuales tampoco se habría llegado a este punto, pues parte de lo que aquí hay presentado se germinó en esas reuniones entre los tres en las que, divirtiéndonos bastante siempre, podíamos acabar *rompiendolo todo* por enésima vez.

A Nacho, por mantener el fuerte durante el tiempo en el que yo perdí el camino.

Al grupo original de *Testeo de Red del C-18*, y a su versión actual del 205, a los que quedan, a los que se fueron y a los nuevos.

A mis directores, Vicente y Andrés, al primero porque siempre ha estado delante de mí en el camino, y al segundo porque se incorporó al camino un poco después que yo, pero ha acabado guiándome para llegar al final.

Y en general, al resto del $\mathcal{GTI} - \mathcal{IA}$ por construir entre todos un entorno en el que resulta agradable trabajar

A Pili y a Jose, por su *concentración, atención, introspección y reflexión*.

A mis padres, porque sin ellos no habría camino.

Por último, a todos aquellos que me han acompañado en mis diferentes estados de ánimo, cuando esperaba a que me visitara la inspiración o la invitaba a quedarse y tomar algo, entre los que están Bruce Springsteen, Van

Morrison, Aaron Neville, Melissa Etheridge, y tantos y tantos otros.

Índice general

Índice de figuras

Parte I

Introducción

1 Introducción

"Lo que importa verdaderamente en la vida
no son los objetivos que nos marcamos,
sino los caminos que seguimos para lograrlo."
– Peter Bamm (1897-1975)

1.1. Motivación

El trabajo aquí presentado se enmarca dentro de una de las líneas de interés del grupo de investigación $\mathcal{GTI} - \mathcal{IA}$ (Grupo de Tecnología Informática – Inteligencia Artificial) del Departamento de Sistemas Informáticos y Computación ($\mathcal{DSIC}$) de la Universidad Politécnica de Valencia. En esta línea se trata de desarrollar una arquitectura de agente junto con una metodología que permita diseñar e implementar agentes que pueden llegar a tener restricciones temporales críticas.

El paradigma de agentes/sistemas multi-agente es considerado como el nuevo paradigma del desarrollo del software, habiendo proliferado enormemente en los últimos años los desarrollos e investigaciones alrededor de este tema. Aunque no hay una definición comúnmente aceptada, se puede considerar un agente software como un sistema de computación situado en un

entorno que es capaz de actuar de forma autónoma y flexible dentro de dicho entorno.

Uno de los principales objetivos que se tienen cuando se trabaja con agentes, es que estos sean capaces de adaptarse a cambios en su entorno. Un agente no se debe comportar igual en cualquier situación, sino que dependiendo de la situación (de su estado interno) y de su entorno (si es o no muy estresante, cambios que hacen que el comportamiento actual del agente no tenga ya sentido, ...) el agente debe modificar su forma de actuar.

El área de las arquitecturas de agente considera los aspectos relacionados con la construcción de sistemas computacionales (describiendo la interconexión de los módulos software/hardware) que satisfagan las propiedades de la teoría de agentes. De esta manera existen toda una miríada de arquitecturas de agente diferentes, cada una de ellas indicando la composición de un agente con unas características particulares. Una de estas arquitecturas, cuyo desarrollo es el objetivo de la línea de investigación en la que se enmarca este trabajo, es la arquitectura de agente ARTIS. En este caso, la característica diferenciadora de este tipo de agente es que puede trabajar en entornos con restricciones temporales críticas. Para ello, la arquitectura de agente ARTIS utiliza técnicas propias de los sistemas de tiempo real.

Se entiende por sistema de tiempo real un sistema computacional en el que interesa no sólo dar una respuesta al problema que resuelve sino el momento en el que es proporcionada dicha respuesta.

En un sistema de tiempo real crítico tradicional se asegura el cumplimiento de las restricciones temporales críticas definidas, pero no se tiene en cuenta consideraciones sobre lo bien que se aprovecha el tiempo de procesador del sistema una vez ejecutadas las tareas con dichas restricciones críticas.

Otro de los problemas que se pueden encontrar en los sistemas de tiempo real tradicionales es su falta de flexibilidad, debido fundamentalmente a la simplicidad de las tareas que incorporan, y a que no son capaces de razonar acerca de sí mismos (ya que las técnicas no están orientadas a eso).

Uno de los ejemplos de aplicación típicos de sistemas basados en agentes de tiempo real sería el del control de un avión. Así, si se tiene un agente encargado del control de un avión, el comportamiento de ese agente (las tareas que debe realizar, los sistemas físicos que debe controlar,...) no es el mismo en una situación de despegue o aterrizaje que en una situación de navegación.

Obviamente, el sistema tiene que cumplir en todo momento las restricciones temporales establecidas en la fase de diseño, incluyendo los momentos en los que el sistema cambie de comportamiento.

En la situación previa a este trabajo, la arquitectura de agente ARTIS no permite su adaptación a cambios en el entorno, y éste es el objetivo de este trabajo: desarrollar un mecanismo que permita incorporar el conocimiento adecuado para modificar el comportamiento del agente haciendo que se adapte tanto a las necesidades cambiantes de su propósito como a los cambios significativos de su entorno a los que no puede responder en base a su estado actual. Por tanto, esta tesis aborda el problema de introducir adaptabilidad en agentes con restricciones temporales críticas, y, en concreto, en la arquitectura de agente ARTIS. Dentro de los sistemas con este tipo de restricciones, no existe una solución satisfactoria a dicho problema.

Por último, resaltar que la línea de investigación del $\mathcal{GTI} - \mathcal{IA}$ relacionada con el desarrollo de la arquitectura de agente ARTIS ha disfrutado y disfruta del apoyo de diversos proyectos de subvención de diversa índole, en los que ha participado el autor de este trabajo, y que en orden cronológico son los siguientes:

- **Sistemas de Tiempo Real basados en el Conocimiento para el Control de Procesos Industriales. Entornos de Desarrollo y Ejecución:** Entidad Financiadora: Comisión Interministerial de Ciencia y Tecnología (*CICyT*) (TAP94-0511-C02-01). Duración, desde Abril 1994 hasta Abril 1997. Investigador Responsable: Vicente J. Botti Navarro.

- **ARTIS: Herramienta para el Desarrollo de Sistemas Inteligentes en Tiempo Real Aplicados al Control de Robots Móviles:** Entidad Financiadora: *CICyT* (TAP97-1164-C03-01). Duración, desde Octubre 1997 hasta Octubre 1998. Investigador Responsable: Vicente J. Botti Navarro.

- **ARTIS: Herramienta para el Desarrollo de Sistemas Inteligentes en Tiempo Real Estricto:** Entidad Financiadora: *CICyT* (TAP98-0333-C03-01). Duración, desde Octubre 1998 hasta Octubre 2001. Investigador Responsable: Vicente J. Botti Navarro.

- **SIMBA: Arquitectura de Sisteam Multiagente Basada en ARTIS:** Entidad Financiadora: Ministerio de Ciencia y Tecnología, D. G. de Investigación (CICYT DPI2002-04434-C04-02). Duración, desde Diciem-

bre 2002 hasta Diciembre 2005. Investigador Responsable: Ana María García Fornes.

- **Desarrollo de Sisteam Multiagente de Tiempo Real:** Entidad Financiadora: Ministerio de Ciencia y Tecnología (TIC2003-07369-C02-01). Duración, desde Diciembre 2003 hasta Noviembre 2006. Investigador Responsable: Vicente J. Botti Navarro.

Fruto de la participación en esta línea de investigación, y por tanto en los proyectos anteriores, surgen las siguientes publicaciones:

- *OLA: Una Herramienta para Análisis Off-line en Sistemas de Tiempo-Real Inteligentes* (C. Carrascosa, A. García, V. Julián, A. Terrrasa, V. Tomás, A. García-Fornes y V. Botti), publicado en las Actas de la TTIA'95, p. 30-41. ISBN 84-92098-1-X (1995).
- *Un lenguaje para el desarrollo y prototipado rápido de sistemas de tiempo real inteligentes* (C. Carrascosa, V. Julián, A. García-Fornes y A. Espinosa), publicado en las Actas de la CAEPIA'97, p. 685-694. ISBN 84-8498-765-5 (1997).
- *Modelling agents in hard real-time environments* (V. Botti, C. Carrascosa, V. Julián y J. Soler), publicado en los *Proceedings of MAAMAW'99*. LNCS / LNAI Vol. 1647, p. 63-76. ISSN: 0302-9743. (1999).
- *Formalización y traducción a un modelo ejecutable de las entidades de un agente ARTIS* (V.J. Julián, C. Carrascosa y V. Botti), publicado en las Actas de CAEPIA-TTIA'99. ISBN 931170-0-5, ISBN 84-699-0289-X. (1999).
- *InSiDE: una herramienta para el desarrollo de Agentes ARTIS* (V. Julián, M. Gonzalez, M. Rebollo, C. Carrascosa, V. Botti), publicado en las Actas de SEID'2000, p. 79-87. ISBN 84-8158-163-1. (2000).
- *Applying the ARTIS Agent Architecture to Mobile Robot Control* (J. Soler, V. Julián, C. Carrascosa y V. Botti), publicado en los *Proceedings of IBERAMIA 2000*. LNCS/LNAI Vol. 1952, p. 359-368. ISSN: 0302-9743. (2000).
- *SIMBA: An Approach for Real-Time Multi-Agent Systems* (J. Soler, V. Julián, M. Rebollo, C. Carrascosa y V. Botti), publicado tanto en el Boletín ACIA (Nº 28, p. 297-304. ISSN: 1577-1989) como en los *Proceedings of CCIA 2002* (LNCS/LNAI Vol. 2504, p. 282-293. ISSN: 0302-9743). (2002).

- *Towards a Real-Time MAS Architecture* (J. Soler, V. Julián, M. Rebollo, C. Carrascosa y V. Botti), publicado en los *Proceedings of Challenges in Open Agent Systems 2002*. LNCS/LNAI. ISSN: 0302-9743. (2002).
- *SIMBA architecture for social real-time domains* (C. Carrascosa, M. Rebollo, J. Soler, V. Julián y V. Botti), publicado en los *Proceedings of EUMAS 2003*, p. 1-13. (2003).
- *Deliberative Server for Real-Time Agents* (C. Carrascosa, M. Rebollo, V. Julián y V. Botti), publicado en los *Proceedings of 3rd International / Central and Eastern European Conference on Multi-Agent Systems*. LNCS/LNAI Vol. 2691, p. 485-496. ISSN: 0302-9743. (2003).
- *Real-Time Agents: Reaction vs. Deliberation* (C. Carrascosa, J. Fabregat, A. Terrasa, V. Botti), que será publicado en los *Proceedings of Agents in dynamic and real-time environments* (2004).

1.2. Objetivo

El objeto general del presente trabajo es incorporar un proceso de meta-razonamiento en un agente con restricciones temporales críticas (en concreto el agente ARTIS), que le permita al agente incrementar su capacidad de adaptación a cambios en las diferentes situaciones a las que se enfrente, entendiendo por situación la conjunción del estado del entorno con el estado interno del propio agente. Estas capacidades de meta-razonamiento se aplicarán al uso del tiempo de procesador disponible así como al comportamiento que presenta el agente.

Este objeto último que tiene la presente tesis se concreta en los siguientes objetivos fundamentales:

1. Revisión del estado del arte.
2. Estudio de las deficiencias de la arquitectura de agente ARTIS, y en particular, de las relacionadas con la falta de adaptación ante cambios en el entorno.
3. Diseño de las capacidades de meta-razonamiento que aporten una mejora a la adaptatividad del agente.

4. Extensión de la arquitectura ARTIS para incorporar las nuevas capacidades de meta-razonamiento. Como se verá a lo largo de este documento, para poder llevar esto a cabo, ha sido necesario replantear la arquitectura de agente ARTIS a todos sus niveles de abstracción: su definición formal, el modelo mediante el cual el diseñador especifica el agente (modelo de usuario) y el modelo de bajo nivel que es directamente ejecutable sobre el computador (modelo de sistema).

5. Implementación de las extensiones anteriores sobre los modelos de usuario y de sistema.

6. Validación del funcionamiento de dichas extensiones mediante la definición de un conjunto de aplicaciones de prueba.

1.3. Estructura del Documento

El presente documento está estructurado en seis partes:

1. La primera parte es esta misma, compuesta por la introducción del documento incluyendo la motivación y objetivos del trabajo desarrollado así como esta estructuración del documento.

2. La segunda parte recoge un estudio del estado del arte, incluyendo el capítulo 2 donde se revisan los conceptos necesarios tanto sobre inteligencia artificial, o más concretamente sobre agentes, como sobre sistemas de tiempo real, comentando algunas aproximaciones dentro de estas áreas. Dentro del estudio del estado del arte se recoge también el capítulo 3 donde se presenta el estado de la arquitectura de agente ARTIS previo al trabajo aquí presentado.

3. La tercera parte recoge la descripción de las diferentes aportaciones de esta tesis. Para ello, se divide en varios capítulos, el primero de los cuales (el capítulo 4) muestra una visión global de las diferentes aportaciones realizadas centrándose en lo que supone uno de los principales puntos del presente trabajo, el método de gestión de diferentes comportamientos dentro de la arquitectura de agente ARTIS. Los tres siguientes capítulos (del 5 al 7) presentan respectivamente las extensiones realizadas a los diferentes modelos de abstracción de la arquitectura de agente ARTIS, modelo formal, modelo de usuario y modelo de sistema.

El último capítulo de esta parte resalta una de las extensiones realizadas al agente ARTIS, como es la del procedimiento de gestión de las respuestas del agente ante eventos significativos mediante el concepto de *Meta-Reglas*, desarrollando dicho concepto en los tres niveles de abstracción propios de la arquitectura de agente ARTIS (abstracto o formal, de usuario y de sistema).

4. La siguiente parte, que comprende el capítulo 9, presenta una serie de pruebas que permiten validar las nuevas funcionalidades de la arquitectura de agente ARTIS, describiendo previamente el sistema de ejecución junto con sus restricciones.

5. La penúltima parte del documento, formada por el capítulo 10, incluye las conclusiones del trabajo realizado así como una serie de propuestas de líneas de desarrollo futuras.

6. La última parte del documento está formada por dos anexos. El primero presenta el nuevo modelo de gestión de eventos incorporado al modelo de sistema de la arquitectura de agente ARTIS, mientras que el segundo presenta la sintaxis completa del lenguaje de control desarrollado para especificar *Meta-Reglas*.

Parte II

Estado del Arte

2 Estado del Arte

"The aesthetics of knowledge are at the heart of several key polarities in the history of Chinese thought. The kind of knowledge that anyone can gain by studying is called ***hsüeh*** *("book learning"would be the closest equivalent)."*
"They Have a Word for It"
– Howard Rheingold

2.1. Introducción

Este capítulo presenta un estudio del estado del arte del trabajo que se recoge en este documento. Para ello, y teniendo en cuenta que este trabajo aúna conceptos de diversos ámbitos, se ha estructurado el capítulo en cuatro secciones. La primera de estas secciones presenta una panorámica del paradigma de agentes, que va desde la problemática de la definición de agente hasta una revisión del concepto y taxonomía de arquitecturas de agente junto con algunos ejemplos. La segunda sección está dedicada a introducir el concepto de meta-razonamiento. La tercera sección muestra una visión global del ámbito de los sistemas de tiempo real, prestando especial importancia a la problemática de la planificabilidad de este tipo de sistemas, así como a

los cambios de modo. La última sección de este capítulo presenta la disciplina de inteligencia artificial en tiempo real, en la que se enmarcaría el trabajo desarrollado, junto con algunas aproximaciones previas dentro de esta disciplina al problema de un agente con restricciones temporales.

2.2. Agente

El próximo avance más significativo en el desarrollo de software

En los últimos años el paradigma de agente y sistemas multi-agente se ha convertido en una de las principales técnicas aplicadas en la resolución de problemas complejos propios de la disciplina de la Inteligencia Artificial. Tanto es así que es considerado como *el próximo avance más significativo en el desarrollo de software* [Sargent 1992] y *la nueva revolución en software* [Ovum 1994]. Numerosas aplicaciones basadas en este nuevo paradigma vienen ya siendo empleadas en infinidad de áreas [Jennings 1998], tales como:

- Dentro del marco de las *aplicaciones industriales* podríamos destacar aquellas que se encargan de:
  - *Control de procesos*: aplicado a la gestión del transporte de electricidad (en el norte de España), control de un acelerador de partículas, monitorización y diagnóstico de fallos en plantas nucleares y control en el proceso de bobinado del acero.
  - *Producción*: se ha aplicado con éxito por ejemplo a sistemas encargados de las fases de ensamblaje, pintado, almacenamiento de productos, etc...
  - *Control de tráfico aéreo*: se han desarrollado aplicaciones para el control del tráfico aéreo en aeropuertos como el de Sidney en Australia.
- También está siendo empleado en *aplicaciones comerciales* para:
  - *Gestión de información*: como por ejemplo el filtrado inteligente de correo electrónico, de grupos de noticias o la recopilación automática de información disponible en la red.
  - *Comercio electrónico*: se emplea para proporcionar el entorno virtual donde realizar las operaciones comerciales (compra-venta de

productos) o también para realizar tareas de búsqueda de productos (comparando precios, consultando disponibilidad) todo ello de manera automatizada.

- Otra área de interés son las *aplicaciones médicas* como por ejemplo:
  - *Monitorización de pacientes en cuidados intensivos*: empleado para monitorizar y controlar a pacientes ingresados en unidades de cuidados intensivos.
  - *Atención al paciente*: estos sistemas se encargarían de seguir el tratamiento de un paciente controlando todos los aspectos relativos a la enfermedad que tenga el mismo.
- Por último, también se viene empleando en áreas de *entretenimiento* como pueden ser:
  - *Juegos*: la aplicación de esta tecnología en juegos permite disponer de juegos más sofisticados, con características inteligentes donde se pueden incorporar personajes virtuales que pueden funcionar de forma casi autónoma.
  - *Teatro interactivo y cine*: se permite a un usuario interpretar el papel de un personaje en una obra donde el resto de los personajes pueden ser virtuales.

2.2.1. Definición de Agente

Se pueden encontrar propuestas en la literatura un gran número de definiciones del concepto de agente, sin que ninguna de ellas haya sido plenamente aceptada por la comunidad científica, siendo quizás la más simple la de Russell [Russell 1995], que considera un agente como una entidad que percibe y actúa sobre un entorno.

Basándose en esta definición, se pueden caracterizar distintos agentes de acuerdo a los atributos que posean (y que van a definir su comportamiento) [Botti 1999] para resolver un determinado problema. Así, se puede hablar de agente social, agente adaptativo,... de acuerdo a los atributos que posea.

Por otra parte, se pueden encontrar otras definiciones de agente que restringen la amplitud de esa primera definición, exigiendo a un agente que cumpla algunos de esos atributos en su definición básica.

De esta forma, Franklin [Franklin 1996] incluye el atributo de autonomía dentro de su definición básica de agente, que formaliza un agente como un sistema situado dentro de y formando parte de un entorno que percibe y actúa sobre ese entorno, a través del tiempo, persiguiendo su propia agenda para así actuar sobre lo que percibirá en el futuro.

Otra definición similar es la mencionada por Huhns [Huhns 1998], donde los agentes son componentes activos y persistentes que perciben, razonan, actúan y comunican.

Una definición que trata de conciliar todas estas diferencias en base a una serie de atributos es la de H. Van Parunak [Van Parunak 1999], que define un agente como una evolución, un incremento, de un objeto activo que puede o no tener toda una serie de atributos adicionales (tantos y del tipo como sea necesario). Así, compara un agente con una navaja del ejército suizo en el que se puede ver la definición básica como sólo la navaja, y en el que si se necesita algún accesorio más se le añade, y si no se necesita, no hay necesidad de acarrear todo (figura 2.1).

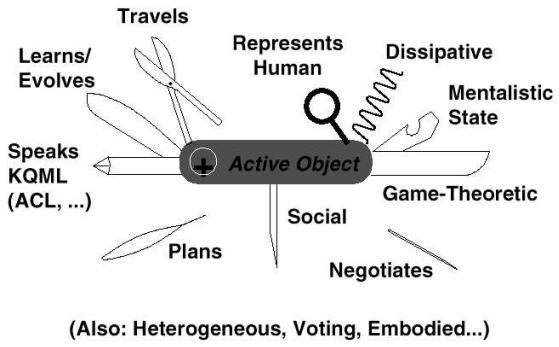

Figura 2.1: Accesorios de un Agente (Parunak)

Finalmente, la definición que recibe una mayor aceptación hoy en día es la propuesta en [Wooldridge 1995], según la cual, un agente es un sistema de computación capaz de acción autónoma flexible en un entorno, entendiendo por flexible que sea:

- *Reactivo*, que responda al entorno en que se encuentra.
- *Pro-activo*, que sea capaz de intentar cumplir sus propios planes u obje-

tivos.

- *Social*, que sea capaz de comunicarse con otros agentes mediante algún tipo de lenguaje.

Atendiendo a esta idea, para poder acuñar a una herramienta el término *agente*, ésta debe ser capaz de cumplir los requerimientos anteriormente expuestos. Actualmente, tan sólo un pequeño porcentaje del software existente se adapta a dicha definición.

2.2.1.1. Atributos de los Agentes

Algunos de los atributos o propiedades, que se suelen atribuir a los agentes en mayor o menor grado para resolver problemas particulares y que han sido descritos por autores tales como Franklin y Graesser [Franklin 1996], y Nwana [Nwana 1996], son:

- *Continuidad Temporal*: se considera un agente un proceso sin fin, ejecutándose continuamente y desarrollando su función.
- *Autonomía*: según [Castelfranchi 1995] un agente es autónomo si puede operar sin la intervención directa de humanos o de otros, y tiene alguna clase de control sobre sus acciones y estado interno
- *Sociabilidad*: este atributo permite a un agente comunicar con otros agentes o incluso con otras entidades.
- *Racionalidad*: el agente es capaz de razonar sobre los datos que percibe con tal de calcular la solución más correcta.
- *Reactividad*: un agente actúa como resultado de cambios en su entorno. En este caso, un agente percibe el entorno y esos cambios dirigen el comportamiento del agente.
- *Pro-actividad*: un agente es pro-activo cuando es capaz de perseguir sus propios objetivos a pesar de cambios en el entorno. Esta definición no contradice la de reactividad. El comportamiento del agente es resultado de dos tipos de comportamientos, el comportamiento receptivo y

el comportamiento de descubrimiento. En un comportamiento receptivo, el agente es guiado por el entorno. El comportamiento de descubrimiento aúna procesos internos del agente para obtener sus propios objetivos. El comportamiento del agente debe ser cercano a la mitad de los dos comportamientos. El agente debe tener un grado de comportamiento receptivo (atributo de reactividad) y un grado de comportamiento de descubrimiento (atributo de pro-actividad).

- *Adaptatividad*: está relacionado con el aprendizaje que un agente es capaz de realizar y si puede cambiar su comportamiento basándose en ese aprendizaje.
- *Movilidad*: capacidad de un agente de trasladarse a través de una red telemática.

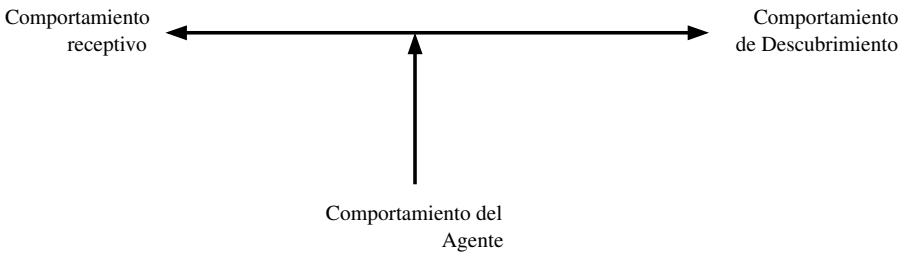

Figura 2.2: Situación del comportamiento del agente

- *Veracidad*: asunción de que un agente no comunica información falsa a propósito.
- *Benevolencia*: asunción de que un agente está dispuesto a ayudar a otros agentes si esto no entra en conflicto con sus propios objetivos.

En este momento no se ha alcanzado un consenso sobre el grado de importancia de cada una de estas propiedades para un agente. Sin embargo, se puede afirmar que estas propiedades son las que distinguen a los agentes de meros programas.

2.2.2. Arquitecturas de Agente

Una arquitectura de agente es una metodología particular para construir agentes

El área de las arquitecturas de agente considera los aspectos relacionados con la construcción de sistemas computacionales (describiendo la intercone-

xión de los módulos software/hardware) que satisfagan las propiedades de la teoría de agentes. Se puede considerar que una arquitectura de agente es una metodología particular para construir agentes [Maes 1991] [Iglesias 1998].

Arquitecturas Horizontales frente a Verticales

Existen varias clasificaciones paralelas de las arquitecturas de agente atendiendo a diversas características. Una primera clasificación se basa en el concepto de *capa* . De esta manera, se divide el procesamiento del agente en *trozos* cada uno de los cuales está encargado de parte de este procesamiento. Estos *trozos* son denominados *capas*. Así esta primera clasificación se realiza en base a si todas las capas tienen acceso a los sensores y actuadores (horizontales) o sólo la capa más baja tiene acceso a ellos (verticales) .

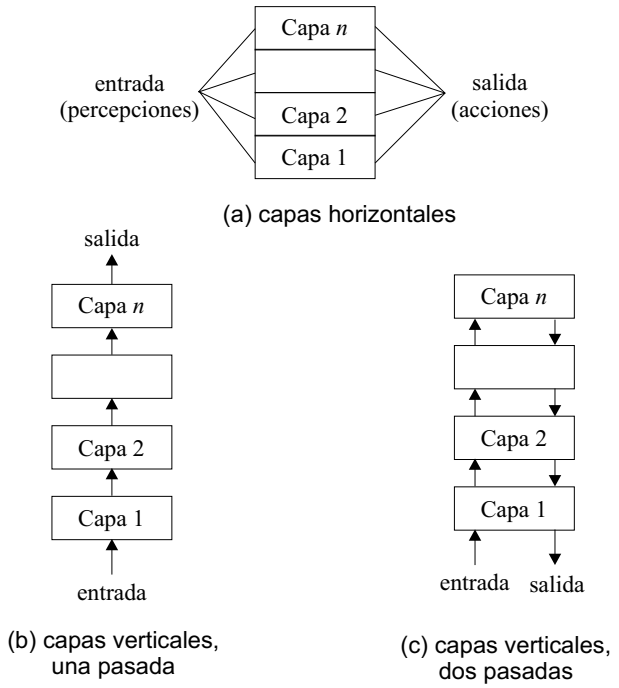

Figura 2.3: Arquitectura horizontal (superior) y vertical (inferior)

Las arquitecturas horizontales ofrecerán la ventaja del paralelismo entre capas a costa de un conocimiento de control más complejo para coordinar las acciones de las capas, mientras que las verticales reducen este control a costa de una mayor complejidad en la interacción entre capas.

Otra posible clasificación es atendiendo al tipo de procesamiento empleado, pudiendo distinguir así tres tipos de arquitecturas: deliberativas, reacti-

vas e híbridas.

2.2.2.1. Arquitecturas Deliberativas

Basado en representación simbólica

El agente es construido de acuerdo al paradigma de la IA simbólica, que se basa en la hipótesis de los sistemas de símbolos físicos enunciada por Newell y Simons, según la cual un sistema de símbolos físicos capaz de manipular estructuras simbólicas puede exhibir una conducta inteligente. Para poder trabajar en el nivel de Conocimiento de Newell, se plantea el problema de cómo describir los objetivos y medios de satisfacerlos, y cómo realizar la traducción del nivel de conocimiento al nivel simbólico [Iglesias 1998].

IRMA [Bratman 1987] y GRATE* [Jennings 1993] son algunos ejemplos de este tipo de arquitecturas.

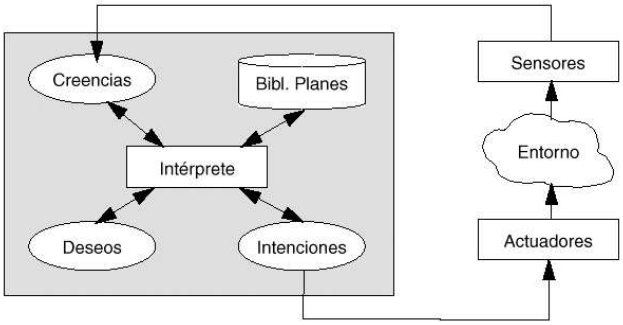

Figura 2.4: Arquitectura deliberativa genérica

Las arquitecturas de agentes deliberativos suelen basarse en la teoría clásica de resolución de problemas de inteligencia artificial que se puede encontrar explicada en [Hendler 1990, Rich 1994, Minton 1994]: dado un estado inicial, un conjunto de operadores/planes y un estado objetivo, la deliberación del agente consiste en determinar qué pasos debe encadenar para lograr su objetivo. Como ejemplo de arquitectura cuyo componente principal es un planificador podemos citar los *Softbots* [Etzioni 1991], cuya misión consiste en ayudar a los usuarios a realizar las tareas típicas de *Unix*.

Podemos distinguir los siguientes tipos principales de arquitecturas deliberativas o simbólicas [Moulin 1996] [Wooldridge 1995]: arquitecturas inten-

cionales y arquitecturas sociales.

- Las agentes intencionales se distinguen por ser capaces de razonar sobre sus creencias e intenciones. Se pueden considerar como sistemas de planificación que incluyen creencias e intenciones en sus planes.
- Los agentes sociales [Moulin 1996] se pueden definir como agentes intencionales que mantienen además un modelo explícito de otros agentes y son capaces de razonar sobre estos modelos.

Dentro de las arquitecturas intencionales, cabe destacar, según se comenta en [Haddadi 1996], aquellas que han tomado como punto de partida la teoría de agentes BDI en su implementación, representando explícitamente las actitudes intencionales de los agentes. Estos sistemas también suelen utilizar planificación para determinar qué acciones deben llevar a cabo pero, a diferencia de los sistemas de planificación, emplean planes en que se comprueban creencias, deseos e intenciones. Las creencias son el conocimiento que el agente tiene sobre sí mismo y su entorno. Los deseos son objetivos del agente a largo plazo. Como normalmente no puede cumplir todos los objetivos a la vez, ya que tiene unos recursos limitados, se introducen las intenciones, que son los objetivos que en cada momento intenta cumplir el agente. Normalmente también se introduce el concepto de planes, que permiten definir las intenciones como los planes instanciados. Hay muchas arquitecturas de agentes que siguen el modelo BDI, como *IRMA* [Bratman 1988] y *PRS* [Georgeff 1989].

Los agentes sociales pueden clasificarse en dos grandes grupos: agentes intencionales cuya arquitectura ha sido aumentada para abordar el razonamiento sobre otros agentes, tales como *COSY* [Haddadi 1996], *GRATE** [Jennings 1992a, Jennings 1992b] y *DA-Soc* [Hägg 1994]; y arquitecturas que siguiendo la Inteligencia Artificial Distribuida clásica han prestado más atención a los aspectos cooperativos (cuándo, cómo y con quién cooperar), sin modelar necesariamente las intenciones de los agentes, tales como *Archon* [Cockburn 1996, Wittig 1992], *Imagine* [Consortium 1994, Steiner 1996], *Coopera* [Sommaruga 1996] y *MAST* [Domínguez 1992, Garijo 1996, Iglesias 1996].

Las arquitecturas deliberativas son verticales

Las arquitecturas deliberativas pueden clasificarse como verticales porque los estímulos recibidos del exterior son procesados en varias capas de diferente nivel de abstracción y al final el nivel superior decide qué acciones

hay que llevar a cabo (y las realiza directamente o se lo indica a las capas inferiores).

2.2.2.2. Arquitecturas Reactivas

No se incluye un modelo del mundo simbólico central y no usa razonamiento simbólico complejo

La principal característica de este tipo de arquitecturas es que el agente no incluye un modelo del mundo simbólico central y no usa razonamiento simbólico complejo. La *Arquitectura de Subsunción* [Brooks 1986], *PENGI* [Agre 1987] y *ANA* [Maes 1991] son algunos ejemplos de esta arquitectura.

Patrones activados bajo ciertas condiciones de los sensores con un efecto directo en los actuadores

De forma más concreta, las arquitecturas reactivas, tal y como se comenta en [Ferber 1996, Wooldridge 1995], cuestionan la viabilidad del paradigma simbólico y proponen una arquitectura que actúa siguiendo un enfoque conductista, con un modelo estímulo-respuesta. Las arquitecturas reactivas no tienen un modelo del mundo simbólico como elemento central de razonamiento y no utilizan razonamiento simbólico complejo, sino que mantienen una serie de patrones que se activan bajo ciertas condiciones de los sensores y tienen un efecto directo en los actuadores. Esta discusión entre mantener una representación explícita del modelo o no, no es una discusión específica del campo de agente sino de la inteligencia artificial en general [Brooks 1991], de hecho las primeras arquitecturas de agentes reactivos se basan en los planificadores reactivos.

Las principales arquitecturas reactivas son [Ferber 1996]:

- Reglas situadas: la implementación más sencilla de reactividad consiste en definir el comportamiento con reglas del tipo "si situación percibida entonces acciones específicas".

- Arquitecturas de subsunción (*subsumption*) y autómatas de estado finito: permiten gestionar problemas de mayor complejidad que las reglas. Las arquitecturas de subsunción [Brooks 1991] están compuestas por capas que ejecutan una determinada conducta (por ejemplo explorar, evitar un obstáculo, etc.). La estructura de cada capa es la de una red de topología fija de máquinas de estado finitas. Las capas mantienen una relación de inhibición sobre las capas inferiores (inhibir entradas de los sensores y acciones en los actuadores). El control no es central, sino dirigido por los datos en cada capa.

- Tareas competitivas: un agente debe decidir qué tarea debe realizar de entre varias posibles [Ferber 1992], seleccionando la que proporciona un nivel de activación mayor. Se basa en una aproximación ecológica del problema de resolución distribuida de problemas, simulando, como por ejemplo en el sistema *MANTA –Modelling an Anthill Activity–* [Drogoul 1994], que cada agente es una hormiga y decide qué acción debe hacer para cumplir sus objetivos. El problema se resuelve sin comunicación entre los individuos, estableciendo un criterio de terminación del problema. Por ejemplo, los problemas clásicos de búsqueda (misioneros y caníbales, mundo de los bloques, etc.) se interpretan como agentes (cada misionero, cada bloque, etc.) que pueden realizar movimientos y una condición global de terminación.

- Redes neuronales: la capacidad de aprendizaje de las redes neuronales también ha sido propuesta en algunas arquitecturas formadas por redes capaces de realizar una función (evitar colisiones, etc.).

  Las arquitecturas reactivas pueden clasificarse como verticales porque los estímulos recibidos del exterior son procesados por capas especializadas que directamente responden con acciones a dichos estímulos y pueden inhibir las capas inferiores.

Las arquitecturas reactivas son verticales

2.2.2.3. Arquitecturas Híbridas

En esta aproximación, el agente está formado por dos o más subsistemas. Uno de ellos es deliberativo, conteniendo un modelo del mundo simbólico y encargándose de determinar qué acciones deben realizarse para satisfacer los objetivos locales y cooperativos de los agentes. Otro es reactivo y se encarga de procesar los estímulos que no necesitan deliberación. De entre los agentes que poseen arquitectura híbrida podemos destacar *TouringMachines* [Ferguson 1992], *PRS* [Georgeff 1989], *COSY* [Burmeister 1992], *AIS* [Hayes-Roth 1995] e *InteRRaP* [Muller 1994].

Las arquitecturas híbridas pueden ser horizontales (como las *TouringMachines*) o verticales (como *PRS*, *AIS* e *InteRRaP*). Es decir, los módulos o bien se ejecutan en paralelo (como en las *TouringMachines*) o bien se ejecutan de forma mútuamente exclusivas (como *COSY* o *InteRRaP*). La decisión sobre qué módulo *domina* a los otros en un contexto dado (por ejemplo determina las acciones externas a ser llevadas a cabo) es realizada o bien de una forma

centralizada o bien de una manera descentralizada. Por ejemplo, en COSY un *filtro* específico decide entre comportarse de manera reactiva y comportarse de manera pro-activa, mientras que en *InteRRaP* los diferentes componentes analizan secuencialmente (desde abajo hasta arriba) una situación hasta que uno de ellos decide que es *competente* para tratar con ello.

A continuación se describen algunas de las arquitecturas híbridas mencionadas:

1. ***COSY*** [Burmeister 1992]: La arquitectura de agente *COSY* es una arquitectura BDI híbrida que fue desarrollada específicamente para el entorno *DASEDIS* (*Development And Simulation Environment for Distributed Intelligent Systems*). Los agentes *COSY* poseen cinco componentes principales:

   a) Sensores: Reciben las percepciones que no corresponden a comunicaciones.

   b) Actuadores: Permiten al agente realizar acciones sobre el entorno (que no sean actos de comunicación)

   c) Comunicaciones: Este componente le otorga al agente la habilidad de enviar mensajes.

   d) Intenciones: Contiene objetivos a largo plazo, responsabilidades, ... y los elementos de control que toman parte en el razonamiento y la toma de decisiones del componente de cognición.

   e) Cognición: Es el responsable de compatibilizar las intenciones del agente con sus creencias sobre el mundo, y elegir una acción apropiada a realizar. Este componente incluye la base de conocimiento que contiene las creencias del agente, así como tres componentes procedurales:

      - Un componente de ejecución de *scripts*, donde un *script* es un plan para conseguir un objetivo.
      - Un Componente de ejecución de protocolos, donde un protocolo es un diálogo que representa una estructura de cooperación entre dos o más agentes.
      - Un componente de reacción, decisión y razonamiento: Este componente mantiene una agenda de *scripts* activos que pueden ser invocados para satisfacer una de las intenciones del

agente o en respuesta a la situación actual del mismo. Posteriormente un componente de filtrado elige entre los *scripts* que compiten para su ejecución.

2. ***TouringMachines*** [Ferguson 1992]:

   Como se ve en la figura 2.5, la arquitectura de agente *TouringMachine* refleja fielmente la definición de agente dada por Wooldridge y que se comentaba anteriormente. Las tres características que dotaban de flexibilidad al agente, vienen aquí reflejadas cada una en una capa de control del agente:

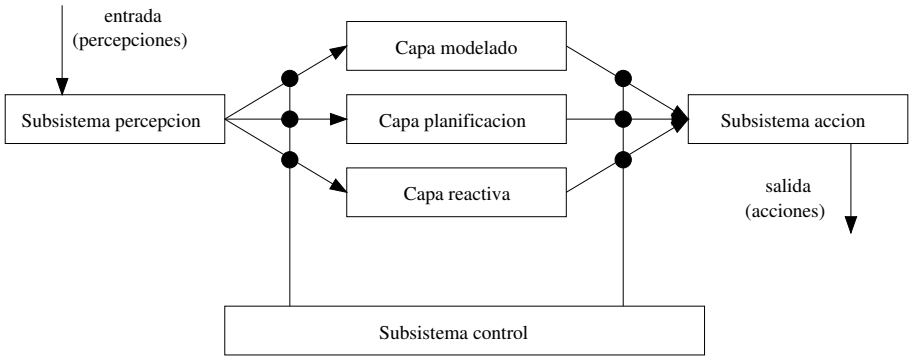

Figura 2.5: Arquitectura híbrida horizontal - *TouringMachines*

   - *Capa Reactiva*: calcula una respuesta reactiva.
   - *Capa de Planning*: Se encarga de realizar planes para calcular una respuesta más deliberada. Es, por tanto, la capa encargada del aspecto pro-activo del agente.
   - *Capa de Modelado*: Modela el comportamiento de los otros agentes que hay en el sistema, tratando de predecir qué es lo que van a realizar. Se encarga, por tanto, del aspecto social del agente.

Arquitectura híbrida horizontal

   Como ya se ha comentado, ésta es una arquitectura híbrida horizontal, con lo que estas tres capas realizan sus procesos en paralelo. En cada ciclo de operación, cada una de estas tres capas tratan de calcular una única acción a realizar. Al final de cada ciclo, se elige una única acción a realizar de entre todas las calculadas por las tres capas.

   Para coordinar tanto el funcionamiento como el acceso al medio de estas tres capas existe un módulo de control (*control framework*). Este

módulo está implementado mediante un conjunto de reglas de control. Estas reglas pueden ser de dos tipos: *censoras* (controlan el acceso al subsistema de percepción por parte de las capas), y *supresoras* (controlan el acceso de las capas al subsistema de acción).

3. ***AIS*** [Hayes-Roth 1995]:

   Otro ejemplo de arquitectura híbrida vertical sería la de *AIS*, o su posterior adaptación al control de un robot *B-Robot* [Uckun 1993] donde se pueden distinguir tres capas (que operan concurrentemente y asíncronamente, y se comunican por paso de mensajes) según la analogía que presentan con el sistema nervioso central de un mamífero:

   Arquitectura híbrida vertical

   - *Médula Espinal*: Esta capa es la capa reactiva del agente, ya que aunque se encarga de enviar la información percibida por los sensores al resto de capas, es capaz de realizar actos reflejos, es decir, decidir realizar acciones sin que entren en juego el resto de capas.
   - *Estructuras Subcorticales*: Existen tres estructuras de este tipo, Coordinación Motor (encargado de traducir a instrucciones para el motor del robot las acciones de alto nivel a realizar por el robot), Coordinación Sensorial (encargado de refinar la percepción y de activar nuevas percepciones), y Control Sensorimotor (encargado de corregir posibles errores en cuanto a la posición del robot).
   - *Córtex*: Esta capa, que posee una arquitectura de blackboard basada en BB1, se encarga de la gestión de objetivos y planes del robot, así como la gestión del histórico y de los mapas periféricos, topográficos, y conceptuales del entorno.

4. ***InteRRaP*** [Muller 1994]:

   Ésta es una arquitectura híbrida vertical formada por tres capas como se observa en la figura 2.7. Esta configuración puede parecer similar a la presentada en las *TouringMachines*, pero presenta dos diferencias principales. La primera es que las capas están jerarquizadas, no son independientes (es una arquitectura vertical, no horizontal). La segunda es que los agentes que siguen esta arquitectura se comunican entre sí, construyendo planes compartidos para lograr objetivos comunes.

   Arquitectura híbrida vertical

   - *Capa de Conducta*: Es la única capa que puede acceder al medio. Es por tanto, la encargada de recoger los datos de los sensores, y decidir si hay que enviarlos a capas superiores buscando alguna que

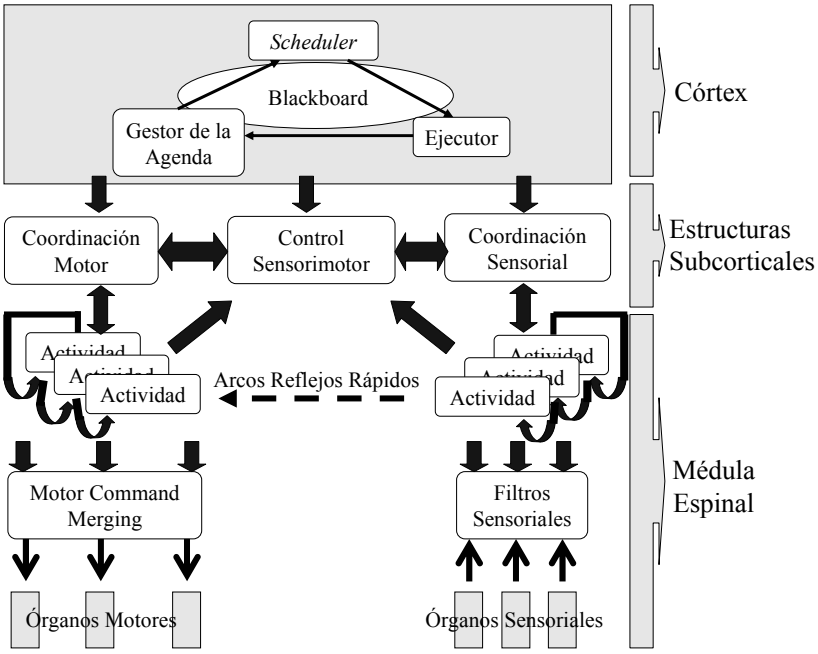

Figura 2.6: Arquitectura híbrida vertical **AIS**/AIBot o **B-Robot** junto con la división según el símil de las capas del sistema nervioso central (Las cajas interiores al círculo son estructuras de datos, los rectángulos exteriores son procesos, y las flechas representan el flujo de datos)

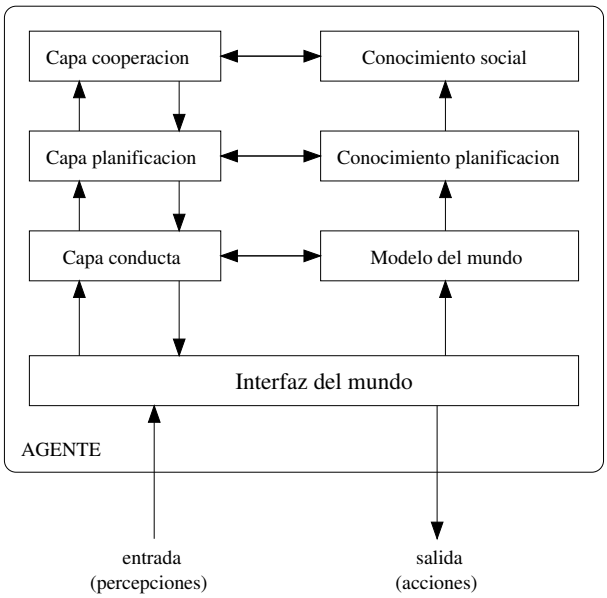

Figura 2.7: Arquitectura híbrida vertical - InteRRaP

sepa tratar esa información. Además recibirá las posibles acciones a realizar.

Esta capa le permite al agente responder frente a ciertas situaciones críticas por medio de los llamados *patrones de comportamiento* (*PoB*). De hecho, existen dos tipos de patrones de comportamiento: los *reactivos* (*reactors*), que se activan frente a eventos externos proporcionando el comportamiento reactivo del agente, y los *procedurales*, que se activan cuando así lo requiere la *capa de planificación*.

- *Capa de Planificación*: Se encarga de la planificación local, proporcionándole al agente una habilidad de deliberación a más largo término.
- *Capa de Cooperación*: Se encarga de la planificación cooperativa. De esta manera, extiende la funcionalidad de planificación de un agente para construir planes por y/o para múltiples agentes que permiten resolver conflictos y cooperar. Para ello mantiene modelos de los otros agentes en el sistema, y es capaz de comunicarse y coordinarse con ellos para construir un plan común (*joint plan*).

2.3. Meta-Razonamiento

En su sentido más amplio, y tal y como define [Russell 1998], el *meta-razonamiento* es cualquier proceso interesado en la operación de otro proceso computacional dentro de la misma entidad. Este término está relacionado con una distinción conceptual entre deliberación en nivel de objeto sobre entidades externas (por ejemplo considerar los méritos de los diversos movimientos de apertura que se pueden hacer en un juego de ajedrez) y deliberación meta-nivel sobre entidades internas (por ejemplo, decidir que no vale la pena perder mucho tiempo deliberando sobre qué movimiento de apertura hacer) [Russell 1998]. Dicho de otro modo, utilizando la nomenclatura de [Raja 2001], un agente puede realizar dos tipos de decisiones:

- Las decisiones de meta o macro-nivel gestionadas por el controlador de meta-nivel.
- Las decisiones de planificación o micro-nivel gestionadas por el controlador de nivel de dominio.

El controlador de meta-nivel se debe diseñar para tomar decisiones rápidas y baratas sobre cuantos recursos se deberían gastar en acciones de dominio y cuantos en acciones de control. Las decisiones de control iniciales se clasifican además, según [Raja 2001], en:

Decisiones rápidas y baratas sobre asignar recursos entre acciones de dominio y de control

- *Decisiones de Coordinación*: Dictan si se coordina o no con otros agentes y cuanto esfuerzo se debería gastar en coordinación.
- *Decisiones del Planificador*: Dictan si se invoca o no al planificador del nivel de dominio y cuanto esfuerzo debería ser gastado por el planificador.
- *Decisiones de **Slack***: Prescribirán cuanto tiempo libre o *slack* debería ser incluido en una planificación para tratar con eventos inesperados.

Las razones para incorporar la capacidad de meta-razonamiento dentro de un agente inteligente son:

- Permitir al agente controlar sus deliberaciones en nivel de objeto: decidir cuales realizar y cuando parar de deliberar y actuar.
- Permitir al agente generar comportamientos computacionales y físicos, tales como planificar para obtener información, que requiere razonamiento introspectivo o reflexivo.
- Permitir al agente recuperarse de errores en sus deliberaciones en nivel de objeto.

Problema: Secuenciar actividades de dominio y de control sin consumir demasiados recursos

Hay que tener en cuenta que tal y como se comenta en [Raja 2001] un agente no está ejecutándose de forma racional si en el momento en el que ha calculado una acción ya no es aplicable. Así, un agente debería planificar y/o coordinar tan sólo cuando la mejora esperada supere el coste esperado. Si se gastan recursos en cantidades significativas para realizar esta meta-decisión, entonces las meta-meta-decisiones sobre si se deben gastar esos recursos deberían ser hechas en el contexto de la ganancia global de la utilidad del sistema. Para hacer esto, un agente tendría que conocer el efecto de todas las combinaciones de acciones antes de tiempo, lo cual es intratable para cualquier problema de tamaño razonable. El problema de cómo aproximar este ideal de secuenciamiento de actividades de dominio y control sin consumir

demasiados recursos en el proceso, es el *problema de control de meta-nivel* para un agente racional de recursos limitados.

De esta manera, y siguiendo también lo expuesto en [Raja 2001], las actividades del agente pueden ser clasificadas en general en tres categorías: *actividades de dominio*, *actividades de control* y *actividades de control de meta-nivel*. Las *actividades de dominio* son acciones primitivas ejecutables que logran las diversas tareas de alto nivel. Las *actividades de control* son de dos tipos, *actividades de planificación* que eligen los objetivos de alto nivel, establecen restricciones sobre como lograr dichos objetivos y secuencian las actividades del nivel de dominio; y *actividades de coordinación* que facilitan la cooperación con otros agentes con tal de lograr los objetivos de alto nivel. Las *actividades de control de meta-nivel* optimizan la ejecución del agente distribuyendo en los momentos adecuados las cantidades de procesador apropiadas y otros recursos a las actividades de control y de dominio.

Teniendo en cuenta que el modelo de meta-razonamiento busca el equilibrio entre realizar alguna computación atómica y ejecutar una acción *real* que afectará a su entorno [Russell 1989], cuando evalúa una computación tiene en cuenta sus dos efectos en la utilidad:

- Hace que pase el tiempo, y por lo tanto puede incurrir en un coste de oportunidad. Específicamente, esta computación causará que el agente posponga la ejecución de la siguiente acción *real* por al menos la duración de un paso computacional.

- Una computación tendrá algún efecto sobre las acciones reales elegidas por el agente.

Hay dos posibles resultados de una computación. El más simple es que puede cambiar la que el agente cree que es la mejor acción. El segundo y más difícil de analizar es cuando la computación no causa realmente un cambio en la elección de la acción, sino que añade alguna información al estado del agente. Esa información adicional, en cambio, causaría que computaciones posteriores cambien la elección de la acción. Esto es llamado *utilidad indirecta* de las computaciones.

2.4. Sistemas de Tiempo Real

Un *Sistema de Tiempo Real* (STR) es un sistema en el cual la corrección del sistema depende no sólo del resultado lógico de la computación, sino también del momento en el que se produce dicho resultado [Stankovic 1988].

De forma habitual, un STR está formado por un conjunto de tareas caracterizadas por un plazo máximo de ejecución (*deadline*), un período, un tiempo de ejecución en el peor caso y una prioridad. Un *deadline* define el intervalo de tiempo más grande en el que el sistema puede proporcionar una respuesta. Si la respuesta se obtiene después de ese tiempo, no será probablemente útil. De hecho, en base a lo que ocurre si se obtiene una respuesta después de su plazo, se diferencian dos tipos de STR:

- Sistemas de Tiempo Real Estricto *Hard Real-Time Systems (HRTS)*: sistemas donde la ejecución de una tarea después de su *deadline* es completamente inútil. Los sistemas de este tipo son sistemas críticos en los que si no se cumplen las restricciones temporales críticas las consecuencias pueden ser catastróficas.
- Sistemas de Tiempo Real Flexible *Soft Real-Time Systems (SRTS)*: se caracterizan por el hecho de que la ejecución de una tarea después de su *deadline* tan sólo decrementa la calidad del resultado de la tarea.

El resto de esta sección revisa los conceptos de tiempo real estricto necesarios para el presente trabajo.

2.4.1. Planificación de Sistemas de Tiempo Real

La forma más habitual de modelar una aplicación de tiempo real es mediante un conjunto de tareas que cooperan entre sí. Estas tareas deben ejecutar sus acciones dentro de intervalos de tiempo bien definidos. Los requisitos temporales de un sistema de tiempo real especifican cuáles son los intervalos de tiempo válidos para la ejecución de cada una de las tareas. El requisito temporal más común es el de plazo máximo de respuesta (*deadline*).

Como la funcionalidad requerida en un sistema de tiempo real es muy diversa, existen distintos tipos de tareas, que dependen de las características de la aplicación. Una tarea se dice que es crítica (*hard*) si el incumplimiento

de alguno de sus requisitos temporales, incluso ocasionalmente, supone un fallo intolerable por sus consecuencias en el sistema controlado. Por el contrario, una tarea es acrítica (*soft*) cuando siendo deseable el cumplimiento de sus requisitos temporales, se puede tolerar el incumplimiento ocasional de alguno ellos. Se denominan tareas periódicas a las que se ejecutan repetidamente a intervalos de tiempo regulares iguales a su periodo. Normalmente las tareas periódicas deben llevar a cabo sus acciones dentro de plazo en todas las ocasiones en que se ejecuten. Por el contrario, una tarea aperiódica se ejecuta de forma irregular, en respuesta a algún evento que ocurre en el sistema controlado. Cuando una tarea aperiódica sea crítica y deba ejecutarse en un plazo de tiempo estricto, la denominaremos tarea esporádica. Las tareas esporádicas deben tener asociada también una separación mínima de tiempo entre dos activaciones sucesivas, a la que se denomina período o intervalo mínimo entre llegadas. Sin ésta restricción no sería posible garantizar el cumplimiento de los plazos [Stankovic 1988].

Así, en este tipo de sistemas se debe asegurar en la fase de diseño que todas las tareas con restricciones temporales críticas podrán cumplir sus plazos máximos de ejecución o *deadlines*. Para ello, teniendo en cuenta la política de planificación a utilizar se realiza un análisis *off-line* de la planificabilidad (test de planificabilidad o de garantía).

En la literatura se pueden encontrar numerosas políticas de planificación, siendo una de las más utilizadas la denominada por prioridades fijas expulsivas [Audsley 1995]. Esta política se basa en la ordenación de las tareas de acuerdo a una prioridad fija que se les asigna a priori. Esta ordenación, aunque se realiza en tiempo de ejecución para seleccionar de entre todas las tareas activas cual es la siguiente que se debe ejecutar, permite la construcción del ya mencionado análisis *off-line* de la planificabilidad en la fase de diseño del sistema.

2.4.2. Cambios de Modo

Una solución para el diseño de sistemas donde hay un gran número de tareas para ejecutar, con unas exigencias de recursos de cómputo muy grandes, pudiendo ser incluso más grandes que las disponibilidades de los mismos, es su división en distintos *modos de funcionamiento*.

Modo de funcionamiento definido por el conjunto de tareas necesarias en ese momento

Según [Pedro 1999] en un sistema monoprocesador *un modo se define por el*

comportamiento del sistema, el cual se describe por medio de un conjunto de funciones permitidas y sus requisitos temporales y, por tanto, mediante un plan, que consiste a su vez en un conjunto de procesos y sus parámetros temporales. Así, un modo de funcionamiento va a venir definido por el conjunto de tareas necesarias en ese momento. Con esto se reduce el número de tareas que van a estar compitiendo por los recursos de forma simultánea, y, por tanto, se reducen las necesidades de recursos del sistema.

Objetivo: Liberar el consumo de recursos por tareas innecesarias

En un sistema de IATR que se ejecute en un único procesador, la definición de distintos modos de funcionamiento está aún más justificada, teniendo en cuenta que este tipo de sistemas poseen tareas con elevado tiempo de cómputo, con lo que la utilización de distintos modos de funcionamiento ayuda a liberar el consumo de recursos por tareas innecesarias, para que se puedan utilizar por las tareas que realmente lo necesiten.

Se pueden encontrar en la literatura relacionada con los sistemas de tiempo real distintas aproximaciones a la gestión de los **cambios de modo** [Sha 1989, Tindell 1992]. Debido a su completitud y actualidad, se ha tomado como base para su estudio el trabajo de [Real 2000]. Según ello, desde el punto de vista del estudio de la transición entre dos modos, se hacen necesarias las siguientes definiciones:

MCR –*Mode Change Request* o *Petición de Cambio de Modo*– : Es el evento que inicia la transición entre dos modos, el actual (al que se suele llamar *previo*) y el *nuevo*. Este evento puede ser externo o interno (generado por una tarea).

Por tanto, como respuesta a una *MCR* se produce una transición entre modos. Esta transición puede no ser instantánea, porque puede haber tareas del modo previo a las que se debe dejar que terminen, a la vez que puede haber tareas del modo nuevo que hay que activar desde este momento. Esto supone que durante la transición entre modos la carga del sistema no corresponde a la de ninguno de los dos modos implicados, por lo que aunque estos modos hayan sido asegurados mediante un análisis de planificabilidad, también hay que analizar la transición entre dichos modos para asegurar la consistencia de los datos compartidos así como el cumplimiento de los plazos máximos de ejecución de las tareas que deban ejecutarse completamente antes, durante y después de dichos modos.

Protocolo de Cambio de Modo: Mecanismo que permite realizar la transición entre dos modos de funcionamiento cuando llega una *MCR*. Este mecanismo debe permitir cambiar entre modos de forma controlada, permitiendo conocer a priori cómo se va a comportar el sistema durante la transición entre modos.

Tipos de tareas: Si el sistema posee m modos de funcionamiento, denominados $\{M_1, M_2, ..., M_m\}$, una tarea τ_i se caracteriza de la siguiente manera:

$$\tau_i = \{\tau_i^{M_1} = (C_i^{M_1}, T_i^{M_1}, D_i^{M_1}),$$
$$\tau_i^{M_2} = (C_i^{M_2}, T_i^{M_2}, D_i^{M_2}),$$
$$...$$
$$\tau_i^{M_m} = (C_i^{M_m}, T_i^{M_m}, D_i^{M_m})\}$$

Es decir, una tarea se define por tantos perfiles como modos de funcionamiento hay en el sistema. Para indicar si una tarea pertenece o no a un modo, se utiliza el tiempo de cómputo en el peor caso (*Worst-Case Execution Time,* —*WCET*—), de forma que una tarea pertenece a un modo de funcionamiento si y sólo si su tiempo de cómputo en el peor caso en ese modo de funcionamiento no es 0.

Con respecto a la transición entre modos de funcionamiento, las tareas se clasifican en:

- *Tareas pertenecientes al modo previo*: A su vez se dividen en:
  - *Tareas Completadas*: Se permite que terminen su ejecución.
  - *Tareas Abortadas*: Se eliminan en cuanto llega la *MCR*.
- *Tareas pertenecientes al modo nuevo*: A su vez se dividen en:
  - *Tareas Cambiadas*: Son tareas que ya estaban en el modo previo, pero que sufren alguna modificación en sus características (período, *deadline*, o incluso *wcet*).
  - *Tareas Invariables*: Son tareas que no varían entre ambos modos.
  - *Tareas Nuevas*: Aquellas que no existían en el modo previo.

Retardo de Incorporación de una tarea τ_i –Y_i– : Es el tiempo que debe transcurrir desde la *MCR* hasta que se activa la tarea τ_i por primera vez dentro del modo nuevo.

La principal utilidad del uso de estos retardos de incorporación, es controlar la sobrecarga que ocurre durante una transición entre modos con tal de que dicha transición sea planificable.

Plazo de Cambio de Modo de una tarea τ_i –$\mathcal{D}_i$– : Mide con respecto a la *MCR* el máximo tiempo que puede tardar en ejecutarse completamente la primera invocación de una tarea nueva τ_i. Teniendo en cuenta que el tiempo de respuesta de dicha tarea es R_i, se debe cumplir que:

$$R_i + Y_i \leq \mathcal{D}_i$$

Requerimientos de la transición entre modos: Existen una serie de propiedades o requerimientos a tener en cuenta a la hora de estudiar la transición entre modos de funcionamiento, y los protocolos de cambio de modo:

- *Planificabilidad*: Aún durante la transición entre modos, todas las tareas críticas del sistema (sean del modo previo o del nuevo) deben cumplir sus *deadlines*. Como ya se ha comentado, se puede utilizar los retardos de incorporación de las tareas nuevas para disminuir la sobrecarga y así permitir cumplir la planificabilidad de todas las tareas. Sin embargo, hay que tener en cuenta que el objetivo debe ser minimizar la duración de la transición entre modos.
- *Periodicidad*: Las tareas invariables deben mantener su periodicidad sin ser afectada ésta por la transición entre modos.
- *Consistencia*: El uso de recursos compartidos debe mantenerse consistente durante la transición entre modos. Así, hay que tratar de evitar situaciones anómalas tales como que una tarea nueva sea bloqueada por una tarea previa. Para ello, una tarea nueva no debe empezar antes de que hayan terminado todas las tareas previas que puedan utilizar alguno de los recursos que necesita.
- *Prontitud*: Las tareas más urgentes del nuevo modo deberían empezar lo antes posible y no ser retrasadas por tareas menos urgentes del modo previo. Este requerimiento puede ser logrado asignando distintos valores a los *plazos de cambio de modo*.

Objetivo: Encontrar retardos adecuados de introducción de las nuevas tareas

El objetivo de un protocolo de cambio de modo es encontrar los valores en los retardos de introducción de las tareas del nuevo modo que permitan cumplir los requerimientos anteriores.

De acuerdo a la forma en la que tratan las tareas invariables, los protocolos de cambio de modo se clasifican en:

Protocolos con continuidad: Las tareas invariables se ejecutan sin tener en cuenta que se produce un cambio de modo.

Protocolos sin continuidad: Las tareas invariables pueden ver alterada su activación.

Por otra parte, se establece otra clasificación paralela de los protocolos de cambios de modo teniendo en cuenta la forma en la que combinan las tareas del modo previo y del modo nuevo:

Protocolos Síncronos: Las tareas del modo nuevo no se activan hasta que no han finalizado todas las tareas completadas del modo previo. En este tipo de protocolos, por lo tanto, no hay que realizar ningún análisis de la planificabilidad de la transición entre modos, pues en el período de transición lo único que ocurre es que van eliminándose las tareas del modo previo que no sean invariables.

Protocolos Asíncronos: Durante la transición, se pueden ejecutar a la vez las tareas del modo previo y del nuevo.

A continuación se detalla el funcionamiento del cambio de modo asíncrono (en concreto según el protocolo propuesto por [Real 2000]) por su interés en el presente trabajo desarrollado.

2.4.2.1. Cambio de Modo Asíncrono [Real 2000]

En un cambio de modo de este tipo, la activación de cada tarea del modo nuevo se retrasa hasta que han acabado las tareas del modo antiguo que intenten acceder a los mismos datos para asegurar el requerimiento de consistencia. Además estos retardos deben asumir también el requerimiento de planificabilidad de todas las tareas en funcionamiento.

2.4.2.1.1. Cálculo de los Retardos

En [Real 2000] se presentaba un algoritmo para calcular los retardos de cada una de las tareas del nuevo modo. Este algoritmo busca los mínimos re-

tardos para las tareas nuevas (Y_i) y para las tareas inalteradas (Z_i) que hacen la transición entre modos planificable a la vez que mantienen la consistencia en los accesos a los datos entre tareas previas, inalteradas y nuevas. Para conseguir este último objetivo se trata de retrasar el comienzo de la ejecución de una tarea nueva hasta que hayan finalizado todas las tareas previas que puedan hacer peligrar la consistencia de los datos con dicha tarea nueva.

Retrasar la ejecución de una tarea nueva hasta que hayan acabado todas las tareas previas que accedan a sus mismos datos

Dado que el algoritmo de planificación de tareas que se utiliza es por prioridades fijas expulsivas, para poder calcular el retardo de una tarea τ_i tan sólo hace falta tener en cuenta el tiempo de respuesta de la tarea previa de menor prioridad que podría producir algún conflicto con la tarea τ_i (τ_{L_i}).

Además, el retardo de τ_i será como mínimo el tiempo de respuesta de τ_{L_i} (R_{L_i}) en la peor situación para la transición, esto es, cuando la activación de τ_{L_i} se da en el mismo instante que la MCR.

De esta manera, el algoritmo define $\vec{Y}^r = (Y'_1, Y'_2, ..., Y'_n)$, donde n es el número de tareas del nuevo modo de funcionamiento, de forma que cada componente de este vector almacena el tiempo de respuesta de la correspondiente τ_{L_i} (R_{L_i}), si dicha tarea existe, o bien almacena un cero en caso contrario. El vector $\vec{Y}^r$ contendrá, al final del algoritmo los retardos necesarios para cada una de las tareas del modo nuevo para conseguir una transición planificable que mantenga la consistencia de los recursos compartidos (si esto es posible).

Los pasos del algoritmo serían los siguientes:

1. $\vec{Y}^r = (0, 0, ..., 0)$
2. Se asigna arbitrariamente, para cada tarea τ_i, el retardo máximo[1] ($Y_{i,max}$), con la única limitación de que cumplan el requerimiento de planificabilidad de la transición, sin tener en cuenta la consistencia de los datos.
3. Para todas las tareas τ_i, se inicializa a 0 el retardo mínimo ($Y_{i,min}$).
4. Repetir
   - *a*) Se le asigna a todas las tareas el retardo máximo $Y_{i,max}$.
   - *b*) Se ejecuta el test de planificabilidad con la asignación actual de retardos.

[1] Igual al mayor de los *deadlines* de las tareas del modo previo

c) Si la transición no es planificable, el algoritmo termina informando de esta situación. Por otro lado, en el caso en el que la transición sea planificable con los retardos máximos, se busca reducir los mismos:

1) Se realizan reducciones en los retardos en orden decreciente de prioridad de las tareas, para así minimizar los retardos de las tareas más prioritarias. Comenzando por el valor inicial (máximo) del retardo, se prueban valores más pequeños, siempre en el rango $[Y_{i,min}..Y_{i,max}]$, para que la transición siga siendo planificable.

2) Se elige el mínimo de dichos valores y se pasa a analizar la siguiente tarea en orden de prioridad decreciente.

3) Se utiliza una búsqueda binaria para buscar el retardo mínimo, con el objetivo de reducir el número de veces que se debe realizar el test.

d) Se actualiza $\vec{Y}^{r'}$ con los nuevos tiempos de respuesta de las tareas conflictivas. En este punto, se tiene un valor numérico, aunque provisional, para los tiempos de respuesta de las tareas durante la transición. Como ya se ha comentado se utilizan estos tiempos de respuesta para calcular los retardos mínimos que aseguran la consistencia, teniendo en cuenta que todas las tareas se activan en el mismo instante en el que llega la MCR.

e) Si los valores mínimos para asegurar la consistencia son los mismos que en la iteración anterior, entonces el algoritmo termina y se ha encontrado una asignación planificable y consistente.

f) Si no, $\vec{Y}^{r}$ es actualizado con los valores de $\vec{Y}^{r'}$ para la siguiente iteración.

g) Los límites inferiores para los retardos para la consistencia se actualizan y comienza una nueva iteración.

Este algoritmo converge en el mínimo retardo para la consistencia en, como mucho, tres iteraciones.

2.5. Inteligencia Artificial en Tiempo Real

En los últimos tiempos la Inteligencia Artificial en Tiempo Real ha emergido como una importante línea de investigación para la resolución de problemas complejos en los que se requiere *inteligencia* y respuestas en tiempo real.

Según [Stankovic 1993], la *nueva generación* de sistemas de tiempo real estrictos deberían caracterizarse por su flexibilidad y adaptatividad, características difíciles de encontrar en los rígidos sistemas de tiempo real tradicionales. De esta forma, el empleo de técnicas de Inteligencia Artificial y, en concreto, del paradigma de agente/sistemas multi-agente parece lo bastante apropiado para el desarrollo de sistemas de tiempo real estricto en entornos dinámicos y/o no completamente especificados.

Es evidente que dentro del campo de los agentes/sistemas multi-agente uno de los objetivos es construir sistemas capaces de tomar decisiones de forma autónoma y flexible, y de cooperar con otros sistemas dentro de una sociedad. Un entorno de tiempo real, por su propia naturaleza, afectará directamente al diseño de la arquitectura de agente más apropiado para dicho entorno. Por ejemplo, dos características propias de este tipo de sistemas, como son su carácter no-determinista y su carácter dinámico, afectarán directamente a la arquitectura del sistema. Por tanto, si queremos aplicar el paradigma de agente/sistema multi-agente a un sistema que trabaje en un entorno con características de tiempo real, es necesario definir una estructura apropiada para el empleo del paradigma en un entorno de tiempo real estricto.

En el dominio de los sistemas de control en tiempo real complejos, estas técnicas han sido contempladas con renovado interés, ya que aparecen como una prometedora aproximación para tratar con la creciente complejidad de los sistemas a controlar, por ello las tecnologías de sistemas de tiempo real e inteligencia artificial han aunado esfuerzos con el fin de desarrollar Sistemas de Inteligencia Artificial en Tiempo Real (SIATR). Una precisa definición de cual es el objetivo de la IATR puede ser encontrada en [Musliner 1995]: *combinar los métodos de ejecución garantizada de los sistemas de tiempo real con la planificación de IA, la resolución de problemas, y los mecanismos de adaptación para construir un sistema de control inteligente y flexible que pueda, dinámicamente, planificar sus propios comportamientos y garantizar que estos comportamientos satisfacen plazos máximos de ejecución estrictos*. Por ello, diversas aproximaciones han

sido recientemente propuestas para adaptar técnicas de IA a requerimientos de tiempo real. Principalmente se ha trabajado en:

- La modificación de algoritmos tradicionales de la IA para mejorar su predecibilidad.
- Arquitecturas software de IA adaptadas a operaciones de tiempo real.
- Modificación de las políticas tradicionales de *scheduling* en sistemas de tiempo real para adecuarlas a la utilización de algoritmos menos predecibles.

Sin embargo, frecuentemente no se utilizan definiciones estrictas de tiempo real en la comunidad de IA, en concreto en la de IATR, se han desarrollado sistemas de IA sin prestar mucha atención a la limitación de recursos que motiva a la comunidad de sistemas de tiempo real. La atención se centra normalmente en la consecución de objetivos de alto nivel, más que en la consecución de requerimiento del caso peor. Normalmente, el sistema conseguirá, en promedio, la calidad necesaria para el tiempo requerido, pero no existe ninguna garantía de que una tarea concreta pueda alcanzar una calidad determinada. Este es el principal inconveniente para la utilización de estas arquitecturas o técnicas cuando tenemos tareas de tiempo real estricto. Cuando los sistemas de IA han de ser trasladados desde los laboratorios de investigación a aplicaciones del mundo real, están sujetos a las restricciones temporales del entorno en el que operan. Por ello, es necesario desarrollar métodos de alto nivel de IATR, y también es importante desarrollar y adaptar métodos de bajo nivel de sistemas de tiempo real para proporcionar el soporte adecuado a los niveles superiores.

Dentro de este ámbito, se pueden encontrar diversas aproximaciones, de entre las que destacan las que se relatan a continuación.

2.5.1. La arquitectura CIRCA / SA-CIRCA

Una de las soluciones que aparecen en cualquier referencia bibliográfica siempre que se habla de sistemas de tiempo real que incorporen inteligencia artificial es la de la arquitectura CIRCA (*Cooperative Intelligent Real-Time Control Architecture*, –Arquitectura de Control de Tiempo Real Inteligente y Cooperativa–), o como se denomina últimamente [Goldman 2001]

[Musliner 2002] SA-CIRCA (*Self-Adaptive CIRCA* —CIRCA Auto-adaptativo—). Aunque es bien cierto que dicha arquitectura incorpora aspectos tanto de sistemas en tiempo real como de sistemas de inteligencia artificial, no se puede considerar como una aplicación de la idea que subyace en la IATR, que como se ha comentado previamente, es adaptar los métodos de IA para trabajar con restricciones de tiempo real. Esto es debido a que CIRCA es una arquitectura compuesta por tres módulos funcionales que operan en paralelo sin competir por los recursos (figura 2.8):

3 módulos paralelos que *no* compiten por recursos

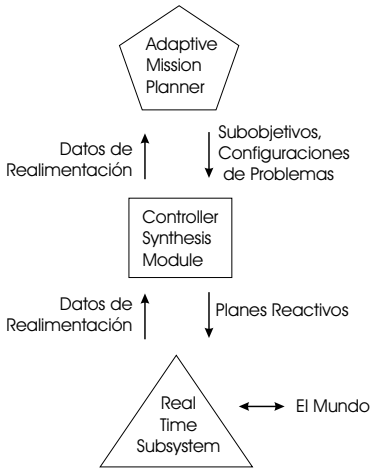

Figura 2.8: División en módulos de la arquitectura CIRCA

- RTS (*Real-Time Subsystem* –Subsistema de Tiempo Real–): Este módulo ejecuta planes de control en tiempo real predecibles que perciben el estado del mundo y responden con acciones para preservar la seguridad del sistema y tratar de conseguir sus objetivos.
- CSM (*Controller Synthesis Module* –Módulo de Síntesis de Controladores–): Este módulo es el encargado de construir dinámicamente los planes de control reactivos que ejecuta el RTS.
- AMP (*Adaptive Mission Planner* –Planificador de Misión Adaptativo–): Este tercer y último módulo es el control de más alto nivel de CIRCA. El AMP es el responsable de dividir la misión global del sistema en fases (*regiones de competencia*) más pequeñas y que intersectan. Cada una de estas fases viene definida por un modo o intervalo de tiempo que comparte un conjunto de objetivos comunes, amenazas y dinámica. Cada

una de estas regiones puede ser cubierta por un plan de control reactivo sintetizado automáticamente. El AMP le encarga al CSM que cree estos nuevos planes de control bien por adelantado antes de que comience la misión, bien en ejecución, cuando las condiciones cambian. Esta síntesis (o planificación) de controladores on-line le proporciona un mecanismo de adaptación (*self-adaptation*) a esta arquitectura.

De esta manera, el AMP controla el razonamiento de SA-CIRCA mediante:

- La modificación de las configuraciones del problema para el CSM.
- La invocación (o parada) del CSM.

El AMP determina para qué fase de la misión está intentando el CSM construir un controlador y cuánto debería trabajar para hacerlo así, modificando las configuraciones del problema de la fase (estado inicial del mundo, objetivos a lograr, amenazas presentes, transiciones de estado que pueden existir debidas al mundo, y acciones disponibles al agente para afectar al mundo).

SA-CIRCA no dispone de recursos ilimitados. De hecho, sufre el problema de *reactividad limitada* (*bounded reactivity*): sólo puede monitorizar y reaccionar a un número limitado de amenazas de forma concurrente.

El AMP descompone el problema en las fases apropiadas para las cuales el CSM genera planes de control con restricciones de tiempo real que preservan la seguridad del sistema. Este proceso de planificación se realiza antes de la ejecución, y continúa conforme el sistema ejecuta otras porciones del plan de alto nivel. Así, cuando el AMP genera una nueva configuración del problema se la envía al CSM. Éste se ejecuta durante una cantidad de tiempo fija (un *quantum* de planificación), y entonces o bien termina con un nuevo plan del espacio de estados o bien es interrumpido cuando su *quantum* se ha consumido. Esto es una limitación de SA-CIRCA, el no sacar beneficio de las computaciones cuya duración sea superior a un *quantum* de tiempo.

Cabe destacar un par de peculiaridades de SA-CIRCA: la primera es la división real entre los sistemas de IA y de Tiempo Real, con lo que la ejecución del primero no se ve afectada por el cumplimiento de las restricciones temporales del segundo. La segunda peculiaridad de este sistema es que el sistema de IA tiene como misión cambiar el sistema de tiempo real.

2.5.1.1. Planificación de la Deliberación [Goldman 2001]

Este es el término utilizado en SA-CIRCA para referirse al proceso de meta-razonamiento (*Deliberation Scheduling*). Así, este proceso es el encargado de decidir qué aspectos del sistema deberían ser mejorados, qué métodos de mejora se deberían elegir, y cuánto tiempo se debería dedicar a cada una de estas actividades.

El proceso de planificación de la deliberación lo realiza el AMP utilizando modelos estocásticos del proceso de síntesis de controladores para dedicar esfuerzo computacional a la mejora del controllador a lo largo de las fases de la misión. El modelo del proceso de síntesis dirige al primer aspecto de tiempo real de la planificación de la deliberación: intenta predecir cuanto tiempo necesitará el proceso de síntesis de controladores para un tipo particular de mejora para una fase concreta de la misión.

El segundo aspecto de tiempo real de la planificación de la deliberación es el coste temporal de la propia decisión de meta-nivel. La forma de tratar con este aspecto es desarrollando heurísticas computacionalmente factibles que hacen las decisiones de la planificación de la deliberación más rápidas. La que actualmente se está utilizando se basa en *mirar* un estado adelante de sus elecciones de acciones inmmediatas y elegir la acción que resulta en el estado (plan global de la misión) con la utilidad esperada más alta.

No se plantea el problema de equilibrar deliberación y acción

Por otro lado, cabe destacar que debido a las peculiaridades de SA-CIRCA, la planificación de la deliberación no es un proceso de meta-razonamiento al uso, pues no se tiene *utilidad indirecta* (en un *quantum* se producirá una utilidad directa en la forma de un nuevo plan para una fase determinada, o no producirá nada en absoluto), pero sobre todo porque no se plantea el problema de equilibrar la deliberación frente a la acción (el motor de ejecución de SA-CIRCA, el RTS, fue diseñado para ejecutarse en paralelo con el subsistema de IA y no para competir por recursos computacionales). Así, la planificación de la deliberación de SA-CIRCA involucra tan sólo balances entre actividades de *planificación* alternativas, no entre planificación y acción.

Según las últimas referencias disponibles, el trabajo en curso sobre SA-CIRCA trata de controlar en tiempo real la actividad de planificación (deliberación) del CSM, de modo que el AMP pueda gestionar y controlar esa deliberación via un proceso de negociación.

2.5.2. DECAF

Otra de las aproximaciones de IATR es la de DECAF (*Distributed, Environment-Centered Agent Framework*) [Graham 2001], que es una arquitectura de agente híbrida para entornos de tiempo real con restricciones temporales de tipo *soft* (un fallo no provoca una catástrofe en el sistema).

La arquitectura de DECAF (figura 2.9) proporciona cinco funciones básicas:

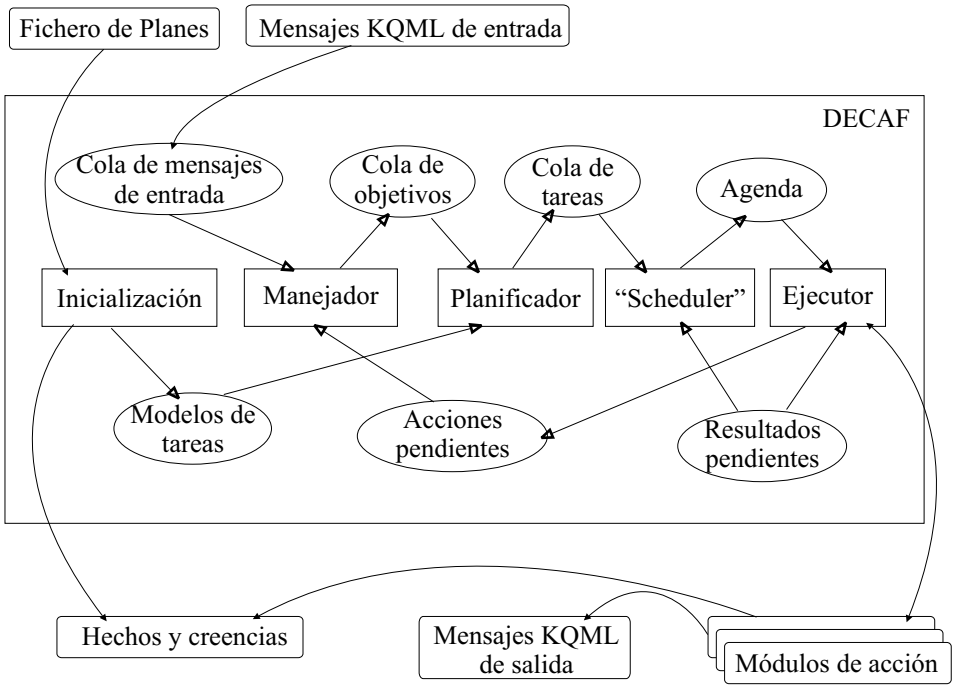

Figura 2.9: La arquitectura de agente DECAF

- *Inicialización*: toma un plan como entrada y produce un conjunto de patrones de tareas para ejecutar.
- *Gestión*: Determina nuevos objetivos para el agente a partir de los mensajes de entrada.
- *Planificación*: Toma un objetivo determinado y los patrones de tareas que lo pueden satisfacer y produce una instanciación concreta del plan que encola en la lista de tareas pendientes.

- *Scheduling*: Planifica cómo ejecutar las tareas pendientes, construyendo una agenda que establece la secuencia de ejecución del agente. Las acciones mantienen una relación de orden parcial, indicando qué acciones pueden ejecutarse en paralelo y entre cuales hay una prelación determinada.
- *Ejecución*: Genera las intenciones del agente a partir de la agenda de acciones. Para DECAF, una intención comprende el compromiso a más bajo nivel de los cursos de acción del agente. Su responsabilidad principal es determinar cuándo una acción se ha completado y determinar qué acciones se pueden ejecutar como consecuencia de esta finalización.

Las *acciones* son las unidades fundamentales planificables de DECAF. Tienen asociado un *perfil de ejecución*, que ayuda al *scheduler* a determinar la secuencia óptima de ejecución para el conjunto de tareas que debe ejecutar. Es un vector que consta de tres componentes: el coste, la duración y la utilidad de la acción.

A partir de los perfiles de ejecución de las acciones, DECAF determina la *función de acumulación característica* (CAF) de cada tarea. Si la tarea está formada por n subtareas, puede ocurrir que sólo deba ejecutarse una de ellas o que sea necesario completarlas todas. La CAF permite determinar qué subconjunto de subtareas ejecuta DECAF.

2.5.3. ASTRO

Éste es un último ejemplo de aproximación de IATR. Según sus propios autores lo denominan, ASTRO [Occello 1998a] se presenta como un *agente desintegrado*, pues está dividido en una serie de módulos de acuerdo a una división funcional del mismo. De esta manera, los diferentes módulos que necesita este agente son:

- Una representación del mundo:

  El modelo del mundo del agente comprende su conocimiento sobre el entorno, los estados internos de otros agentes, y su propio estado interno. El estado interno adecuado incluye, en concreto, los planes que están siendo ejecutados o que el agente toma en consideración. Este modelo es mantenido por un proceso de interpretación de los datos sensoriales.

- Módulos de Percepción y Comunicación:

  El conocimiento sobre el entorno es construido por módulos de percepción. Los otros agentes son percibidos por medio de los módulos de comunicación.

- Módulos de Control:

  Para asegurar la reactividad del agente, un evaluador examina contínuamente el modelo del mundo. Los módulos de control del agente detectan situaciones frente a las cuales el agente necesita reaccionar, las evalúa, y decide tomar las acciones apropiadas, las cuales pueden ser de la forma de crear, suspender o eliminar objetivos (cambiar el contexto del proceso de planificación y ejecución). La supervisión continua de la situación del agente asegura que el agente puede reaccionar a eventos impredecibles en cualquier momento. El mismo mecanismo tiene en cuenta las interacciones con otros agentes usando unos protocolos de interacción propios.

- Módulos de Razonamiento:

  El proceso de razonamiento está formado por módulos de *planning*, de *scheduling* y de secuenciado. Siempre que se crea un objetivo (o se modifica) se busca un plan que logra el objetivo. Esta tarea es realizada por módulos de *planning*.

En cuanto a la estructura de los planes, cada plan tiene un identificador y un *deadline* asociado. La ejecución del plan requiere la ejecución de varios objetivos locales. Cada objetivo local tiene un identificador y una prioridad asociada. La ejecución de cada objetivo local puede ser lograda ejecutando una de las varias acciones alternativas. Cada acción tiene una duración (tiempo de ejecución) y un valor de satisfacción asociado.

El propósito del algoritmo de planificación (*scheduling*) es, en primer lugar, planificar las acciones para cumplir su *deadline* y obtener un valor máximo de la utilidad y de la satisfacción.

Para ello se utiliza el siguiente algoritmo:

1. Para cada plan, se realiza una planificación (*schedule*) de todos los objetivos locales con duración mínima. Si se viola un *deadline* en cualquier fase, se aborta el plan y se acaba.

2. Para un objetivo local con prioridad máxima de los restantes objetivos locales:

   a) Si la acción ya ha comenzado, pasar al paso b. Si no, encontrar la acción con satisfacción máxima del resto de acciones.

   b) Si se viola el *deadline* debido a una nueva acción, pasar al paso 1. Si no, reemplazar la acción con duración mínima por la acción con satisfacción máxima.

Como se puede observar en su primer paso, este algoritmo puede dar lugar a que durante la ejecución del agente se llegue a violar algún *deadline* del mismo. Por este motivo no se puede aplicar a sistemas de tiempo real estricto o *hard real-time systems*. De todas formas, hay que comentar que suponiendo que las acciones con valores de satisfacción más altos requieren más tiempo de ejecución, este algoritmo asegura que se consigue una planificación que cumpla los *deadlines* con una satisfacción mínima siempre que esta planificación exista.

El modelo de agente ASTRO está implementado usando una arquitectura de *blackboard* de tiempo real.

Los módulos reaccionan a las modificaciones del *blackboard*, para sus activaciones e inhibiciones. Trabajan en un contexto local que forma parte de los datos del *blackboard*.

Un blackboard de dominio contiene datos del dominio (usado para la resolución del problema). Un mecanismo de control se encarga de la comunicación entre módulos y del control de la gestión de la actividad de los módulos. Los datos de control (resumen del estado de la solución) son almacenados en un *blackboard* de Control gestionado por la unidad de Control. Los módulos se comunican con el mecanismo de control por medio de eventos.

Todas las comunicaciones son gestionadas por el controlador. Esta unidad de control del *blackboard* asegura la estimulación e inhibición de los módulos según sus especificaciones.

El comportamiento de un módulo es descrito a través de sus interacciones con los datos de control, y es gestionado por medio del intercambio de mensajes con la unidad de control. Un módulo se integra en el sistema por la especificación de su comportamiento frente a los datos del *blackboard*. Una

especificación externa del comportamiento del módulo puede ser expresada por un objetivo, unas precondiciones de activación y unas condiciones de interrupción. La unidad de control recibe eventos de los módulos y emite señales de control hacia ellos. Se activan los módulos que tienen validadas todas sus condiciones. Las señales de inhibición activan el procesamiento de excepciones en módulos. La unidad de control es independiente de la aplicación.

Arquitectura híbrida vertical

En resumen, se puede ver ASTRO (o CELLO como se le denominaba en las primeras fases de su desarrollo en [Occello 1997, Occello 1998b]) como una arquitectura de agente híbrida vertical compuesta de dos capas:

- Percepción / Comunicacíón: Se encarga de la comunicación con el exterior, bien con el entorno, bien con otro agente.
- Razonamiento / Adaptación: Está formado a su vez por dos submódulos:
  - *Planner*: Es activado por un nuevo objetivo, y su función es encontrar un plan que satisfaga dicho objetivo.
  - *Scheduler*: Elige un plan, calcula todas las acciones para ese plan y las secuencia adecuadamente en la agenda de acciones. Si el plan no puede ser secuenciado, se genera un nuevo evento para solicitar la replanificación.

Por otra parte hay que tener en cuenta que un módulo de acción genera órdenes para el dispositivo físico cada período de tiempo $\triangle t$. Además, un módulo de acción puede ser ejecutado por la unidad de control del *blackboard* como reacción a un evento de activación emitido por el módulo *Ejecutor*. Una vez ha comenzado la fase de ejecución, el agente está en un modo percepción/acción, ya que se está ejecutando un plan. Los módulos de percepción reaccionan a modificaciones dinámicas del entorno.

Se pueden adoptar dos aproximaciones para realizar la percepción:

- Elegir un bucle de control para evaluar la percepción.
- Elegir invocar un módulo de evaluación como respuesta las ocurrencias de eventos de percepción.

La *Comunicación* gestiona el buzón y emite eventos internos. Un *Evaluador de la Comunicación* evalúa las consecuencias de la recepción del mensaje (según un protocolo).

El *Scheduler* es activado por una adaptación necesaria. Planifica la acción almacenada en el *blackboard* con una fecha de activación más inmediata para que el ejecutor emita inmediatamente el correspondiente evento interno de apertura de activación.

2.6. Conclusiones

En este capítulo se ha realizado una panorámica a los diferentes ámbitos y conceptos en los que se enmarca el trabajo presentado en este documento. En concreto, estos ámbitos son el de la inteligencia artificial y el de los sistemas de tiempo real.

Por su relevancia para el trabajo desarrollado y que se presentará en los siguientes capítulos, se han revisado los conceptos propios de la inteligencia artificial de *agente* (el propio concepto de qué es un agente junto con diversos ejemplos de los diferentes tipos de arquitecturas de agente existentes) y las ideas que fundamentan el *meta-razonamiento*, y en el caso de los sistemas de tiempo real se ha incidido en la planificación en este tipo de sistemas y en el concepto de *cambio de modo*.

El capítulo termina comentando las principales aplicaciones propias de la fusión de las dos disciplinas anteriores, la Inteligencia Artificial en Tiempo Real, de entre las que cabe destacar CIRCA/SA-CIRCA como la única capaz de trabajar con restricciones temporales críticas, pero teniendo la limitación de que no permite incorporar técnicas de inteligencia artificial para resolver el problema del dominio, sino tan sólo para el problema de meta-nivel. Además, debido a esta división entre el nivel de objeto y el meta-nivel tan acusada, y a que no es capaz de interrumpir y reanudar sus procesos de meta-razonamiento, *desperdiciando* aquellas computaciones que necesiten más tiempo del *quantum* especificado, este proceso de meta-razonamiento atípico no produce ninguna utilidad indirecta. Además, no controla consideraciones sobre el tiempo que le cuesta realizar un cambio del sistema de tiempo real (RTS).

En cuanto a los otros dos ejemplos incluidos corresponden a dos arquitec-

turas de agente híbridas que trabajan en entornos con restricciones temporales. Ahora bien, en ninguno de ambos casos se puede trabajar con restricciones temporales críticas.

3

Descripción de la arquitectura de Agente ARTIS ($\mathcal{AA}$)

"Aquel que conoce a su enemigo y a sí mismo
no debe temer el resultado de cien batallas."
"El Arte de la Guerra"
– Sun Tzu

3.1. Introducción

Este capítulo se encuentra enmarcado dentro de la parte del estado del arte de este trabajo pues presenta una revisión del estado previo al presente trabajo de la arquitectura de agente ARTIS.

De esta manera, después de una definición, se hace una revisión de la arquitectura según los diferentes niveles de abstracción o modelos definidos en la misma, para acabar presentando el Módulo de Control.

3.2. Definición

La arquitectura de Agente ARTIS ($\mathcal{AA}$) constituye una extensión del modelo de pizarra [Nii 1986a, Nii 1986b] adaptado para entornos de tiempo real estrictos. Un $\mathcal{AA}$ combina y potencia las ventajas de dos áreas de investigación que han confluido en los últimos años, como son la Inteligencia Artificial (IA) y los Sistemas de Tiempo Real (STR). Un $\mathcal{AA}$ pretende aunar la flexibilidad de las técnicas de IA, con la predecibilidad de los métodos utilizados en el desarrollo de STR.

Arquitectura híbrida vertical

De acuerdo a las clasificaciones de arquitecturas de agentes existentes [Muller 1994], comentadas en la sección 2.2.2, se puede considerar la arquitectura de $\mathcal{AA}$ como una arquitectura de agente híbrida vertical que tiene como característica diferenciadora el poder gestionar tanto tareas con restricciones temporales críticas como tareas no acotadas temporalmente.

Para poder gestionar estas tareas con restricciones temporales críticas se dispone de un análisis de la planificabilidad [García-Fornes 1997] que, en la fase de diseño del agente, comprueba si se van a poder cumplir o no dichas restricciones temporales críticas, es decir, si el sistema es *planificable* o no. Esto le da la oportunidad al diseñador de ajustar los parámetros del agente hasta que éste sea planificable, es decir, hasta que dicho análisis determine que va a poder cumplir sus restricciones temporales críticas.

El tipo de problemas a los que normalmente se aplicará esta arquitectura será el formado por problemas con restricciones temporales críticas donde las tareas que posean estas restricciones críticas consumirán un porcentaje muy bajo del tiempo de CPU, quedando el resto de tiempo de CPU disponible para otro tipo de tareas del sistema. Así, en un $\mathcal{AA}$ este tiempo se va a dedicar para realizar tareas que, sin ser críticas para el agente, van a mejorar el funcionamiento de éste.

Niveles de Abstracción de un $\mathcal{AA}$: formal, usuario y de Sistema

Tal y como se refleja en la figura 3.1, la arquitectura es presentada a diferentes niveles de abstracción (o modelos). Así, se establecen tres modelos [Julián 1999], el de más alto nivel (y más formal), el de usuario, y el de más bajo nivel (y directamente ejecutable), junto con las traducciones de uno a otro.

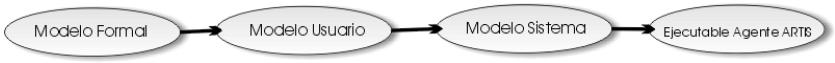

Figura 3.1: Niveles de Abstracción en la arquitectura de Agente ARTIS

3.3. Modelo Formal

La especificación formal de un sistema tiene como objetivo principal la formalización de la arquitectura y sus operaciones. Los beneficios obtenidos de la especificación serán un mejor entendimiento de la descripción del sistema y un acercamiento a la integración de la teoría de agentes y la práctica, intentando salvar el hueco existente entre las dos. Actualmente hay abierta una línea de investigación dentro de la arquitectura de $\mathcal{AA}$ encargada del desarrollo del modelo formal de un $\mathcal{AA}$ [Julián 1999, Rebollo 2003] así como de un método de desarrollo orientado a $\mathcal{AA}$ [Julián 2002a, Julián 2002c].

A continuación se detalla la parte del modelo formal de relevancia al presente trabajo (dejando la explicación de los conceptos que hay detrás de la formalización para la siguiente sección donde se presenta el modelo de usuario).

3.3.1. Agente ARTIS

Formalmente, un agente ARTIS puede ser visto como una tupla:

$$AA = (U_{AIA}, f_{AIA}, B)$$

donde:

U_{AIA} = Conjunto de agentes internos (*in-agents*).

f_{AIA} = Una función de selección de *in-agents* (implementada por el Módulo de Control).

$B = \bigcup B_{AIA}$ Conjunto de creencias las cuales representan el entorno y el estado interno del agente. Este conjunto está compuesto por las creencias de todos los *in-agents* que lo componen.

3.3.2. Agente Interno (*in-agent*)

Formalmente un *in-agent* puede ser visto como una tupla:

$$IN - AGENT = (B_{AIA}, L_r, f_r, L_c, f_c, D, T)$$

donde:

B_{AIA} = Conjunto de creencias que representan su estado interno y su visión del entorno. Todos los datos en este conjunto tienen una marca temporal empleando para ello una lógica temporal.

L_r = Lista de todos las posibles acciones reflejas conocidas por el *in-agent*.

f_r = Función de selección de una acción refleja del conjunto L_r según el estado de B_{AIA}.

L_c = Lista de todos las posibles acciones cognitivas conocidas por el *in-agent*.

f_c = Función de selección de una acción cognitiva del conjunto L_c según el estado de B_{AIA} , D, y una acción inicial (resultado de f_r).

D = *Deadline* (plazo máximo de ejecución).

T = Periodo.

En un principio, un *in-agent* lee los nuevos valores de entrada (percepciones) en los que está interesado de su entorno (función *Read*). Estos nuevos valores son añadidos al conjunto de creencias existente (B_{AIA}) con información temporal referente al instante de inicio del periodo en que son leídos. Este conjunto de creencias es empleado por la función f_r para seleccionar, en un periodo de tiempo acotado, una primera respuesta (*acción refleja*) del conjunto L_r. Posteriormente, la función f_c intentará conseguir una respuesta mejor. Si no fuese posible refinar la respuesta debido al tiempo disponible, la función devolvería el valor *not completed* indicando de esta forma que las acciones a realizar por el *in-agent* serán las obtenidas por la función f_r, en caso de haber tenido tiempo suficiente se ejecutará la acción obtenida después del proceso de cognición. El proceso se repetirá periódicamente cada *T* unidades de tiempo.

3.4. Modelo de Usuario

Este modelo está formado por un conjunto de lenguajes de usuario de alto nivel que permite la especificación de este tipo de agentes [Carrascosa 1997].

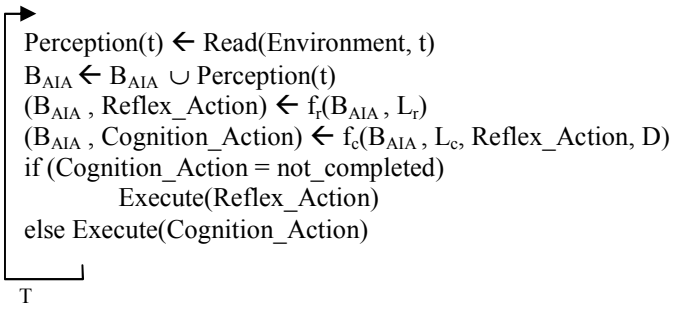

Figura 3.2: Ciclo de un *in-agent*

Estos lenguajes permiten representar el conocimiento específico del problema para poder generar un $\mathcal{AA}$ adecuado. Desde un punto de vista del usuario de la arquitectura de $\mathcal{AA}$, se pueden encontrar dos tipos de conocimiento a representar [1]:

- *Conocimiento del Dominio*: información relativa al entorno del agente, los datos que necesita para resolver un problema (también llamado conocimiento factual).
- *Conocimiento de Resolución de Problemas*: relativo a los métodos para resolver problemas.

3.4.1. Conocimiento del Dominio

Creencia: visión parcial del entorno (información observable + deducida)

La representación del conocimiento del dominio en ARTIS se realiza mediante el concepto de *creencia*. Es imposible que un sistema pueda disponer de una visión completa del entorno en el que se encuentra en un instante determinado, si no que dispondrá de una visión reducida y en ocasiones no actualizada del mismo. Para representar dicha visión reducida de la realidad se emplea el concepto de *creencia*. Las *creencias* representan una visión parcial del entorno que incluye tanto la información que es capaz de percibir directamente como la información que infiere a partir de los datos percibidos y de otros datos deducidos. Una de las características de los entornos de tiempo

[1] Otra nomenclatura o clasificación posible de estas clases de conocimiento sería la que divide el conocimiento del dominio en el conocimiento del entorno y el conocimiento de resolución de problemas.

real es que se trata de entornos dinámicos que evolucionan con el tiempo. Su tratamiento, puesto que es una característica intrínseca del entorno, debe ser algo natural en el modelo que se utilice. De esta forma, el conjunto de las creencias de un $\mathcal{AA}$ se modela como una pizarra temporal.

Para adoptar el modelo de pizarra [Nii 1986a, Nii 1986b] a un entorno de tiempo real [Barber 1994, Crespo 1994], es necesario conocer el tiempo de ejecución de todas las tareas que integran el sistema. Existen varios trabajos enfocados a reducir la complejidad del razonamiento temporal [Onaindía 1997], como por ejemplo la aplicación de técnicas heurísticas que permiten alcanzar tiempos de ejecución razonables para problemas de gran tamaño en algunos contextos determinados [Van Beek 1990] [Van Beek 1996]. Sin embargo, la búsqueda en entornos dinámicos (tales como a los que se aplican los sistemas de tiempo real) debe garantizar algunas propiedades como:

- Predecibilidad: el proceso de búsqueda debe obtener una respuesta dentro de un intervalo de tiempo acotado.
- Interrumpibilidad: el proceso debe ser capaz de detenerse en cualquier instante de tiempo y devolver una respuesta válida.

Para representar adecuadamente la evolución que ha seguido el sistema a lo largo del tiempo, así como los posibles estados futuros que se pueden alcanzar a partir de la situación actual, podemos distinguir entre distintos tipos de información atendiendo a los siguientes criterios:

1. Según su carácter:
   - Observable: información externa que se adquiere directamente a través de los sensores del agente.
   - No observable: información generada a partir de la información de los sensores (o de otra información no observable) en el proceso de razonamiento.
2. Según su procedencia:
   - De entrada: información que desencadena el proceso de razonamiento.
   - Interna: representa el estado interno del agente.

- De salida: resultado del proceso de resolución que se lleva a cabo en el agente.

3. Según su estado temporal: Cada información tiene asociada un período de validez temporal en el que dicha información es válida. Ese período viene definido por un instante de inicio y uno de terminación. En base a este período de validez una información puede ser:

   - Actual: es el valor vigente para un dato. Conocemos su instante de inicio, pero no su instante de terminación.
   - Pasada: se trata de los valores que ha tenido un dato. Se conoce exactamente en qué intervalo de tiempo el dato ha tenido ese valor.
   - Futura: son hipótesis acerca del valor que puede tomar un dato en un instante de tiempo futuro. No es conocido ni el instante de inicio ni el de finalización, por lo que, en ambos casos, se trabaja con una estimación. Las informaciones futuras se clasifican en:

     a) Predicciones, si el dato hace referencia a información observable. Para que se cumpla debe corroborarse a través de la lectura del valor predicho a través de los sensores.

     b) Predicciones dependientes, cuando el dato pertenece a un dato no observable y su cumplimiento depende del cumplimiento de otra información temporal.

La pizarra temporal está estructurada como una taxonomía de *frames* compuesta por un conjunto de clases que pueden tener *slots* estáticos o temporales. Este último tipo de *slots* contiene información cuyo período de validez se restringe a un intervalo de tiempo concreto. Además, en este caso, se puede guardar también un histórico de los valores que ha ido tomando (junto con sus intervalos de validez) dicho *slot* temporal, además de un conjunto de predicciones de los valores futuros que va a tomar (también con su intervalo de validez predicho).

El proceso de razonamiento basado en el modelo de pizarra establece que la solución debe construirse incrementalmente. Cualquier paso de razonamiento puede aplicarse en cada una de las etapas de formación de la solución. De esta forma, la secuencia de construcción de la solución es dinámica y sigue un modelo de razonamiento oportunista (aplicar el conocimiento adecuado en el momento oportuno).

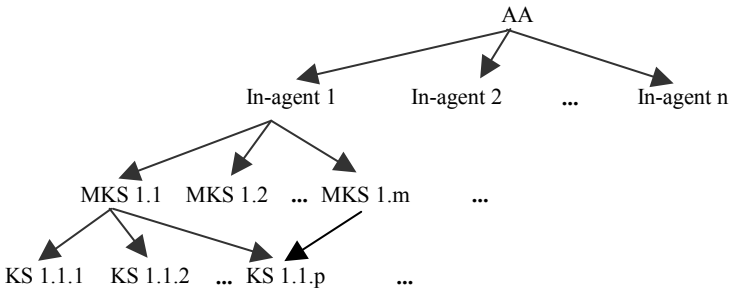

Figura 3.3: Jerarquía de entidades de un $\mathcal{AA}$.

Para poder especificar el conocimiento del dominio en la fase de diseño de un agente ARTIS, o lo que es lo mismo, definir las creencias de dicho agente, existe un *lenguaje de clases* de tipo *LISP* mediante el cual el diseñador especifica las diferentes clases, con sus *slots* estáticos o temporales, junto con instancias de dichas clases que componen las creencias del agente. Existe una herramienta denominada *InSiDE –Integrated Simulation and Development Environment–* [Julián 2000] que permite la especificación de dicho conjunto de creencias por medio de una interfaz gráfica que abstrae de la complejidad del lenguaje de clases.

3.4.2. Conocimiento de Resolución de Problemas

La organización en entidades dentro de la arquitectura presentada, proporciona una jerarquía de abstracciones que organizan el conocimiento para resolver modular y gradualmente el problema global. Así, se puede distinguir, de mayor a menor nivel de abstracción, las siguientes entidades: Agente ARTIS ($\mathcal{AA}$), agente interno (*in-agent*), fuente de conocimiento de nivel múltiple (MKS) y fuente de conocimiento (KS).

Cada uno de los módulos que componen un $\mathcal{AA}$ se corresponde con una de las entidades anteriores de la jerarquía de abstracciones (figura 3.3). De esta manera:

- En la raíz de esta jerarquía, que corresponde al nivel de abstracción mayor, estaría el Agente ARTIS ($\mathcal{AA}$) que modela al sistema completo y su interrelación con el entorno.
- En el primer nivel de abstracción, el $\mathcal{AA}$ está definido por un conjunto

de *in-agents*, cada uno de ellos con el conocimiento para resolver un determinado subproblema.

- Cada uno de estos *in-agents* se define a su vez por un conjunto de MKSs en el segundo nivel de abstracción. El comportamiento de un MKS se describe como un método interrrumpible que proporciona distintas soluciones al mismo subproblema con distintas calidades y distinto tiempo de cómputo (refinamientos sucesivos o como métodos múltiples).
- En el tercer nivel de abstracción, los MKSs se dividen en secuencias ordenadas de KSs (denominadas *niveles*) que contienen el conocimiento (procedural, basado en reglas o cualquier otra técnica de IA) que proporcionan las sucesivas soluciones, en caso de refinamientos sucesivos, o las soluciones alternativas, en caso de métodos múltiples.

Cabe resaltar que una de las principales razones para realizar la división en los distintos niveles de la jerarquía de abstracciones de un agente ARTIS es la de disfrutar de las ventajas de la descomposición modular (división de la complejidad y reutilización del código) [Botti 1999].

División de la complejidad y reutilización del código

Las técnicas de IA, por su propia naturaleza, tienen tiempos de cómputo no limitados o poco realistas lo que, a priori, imposibilita su utilización en sistemas de tiempo real, donde para poder garantizar la planificabilidad es necesario conocer los tiempos de cómputo en los peores casos de las entidades en ejecución. Por otro lado, el concepto de MKS [Carrascosa 1997] permite acotar el tiempo de cómputo para el caso peor de las entidades especificadas sin necesidad de, a priori, podar las funcionalidades de las técnicas de IA utilizadas, pudiendo así utilizar dichas técnicas en la resolución de problemas de tiempo real críticos.

MKS: permite acotar el tiempo de cómputo para el caso peor sin podar las funcionalidades

Tras esta visión global del conocimiento de resolución problemas del $\mathcal{AA}$, a continuación se presentan las diferentes entidades que componen la anterior jerarquía comenzando por la abstracción más simple (KS) para acabar en la raíz de dicha jerarquía (el propio agente ARTIS).

3.4.2.1. KS (*Knowledge Source*)

A partir de las creencias del $\mathcal{AA}$ es necesario disponer de procesos de resolución que permitan determinar que acción o acciones son más apropiadas

en cada momento. Para ello será necesario definir *fuentes de conocimiento* que incorporen el conocimiento de resolución necesario. El agente conoce el estado del entorno, pero también actúa sobre él y esta actuación puede provocar cambios en el mismo.

Una KS (*Knowledge Source*) o fuente de conocimiento, es la entidad de más bajo nivel dentro de la jerarquía de la arquitectura (y, por tanto, la abstracción mínima), y representa el conocimiento de resolución básico (métodos, heurísticas, etc.) para resolver alguna parte de un problema. Este conocimiento puede ser representado proceduralmente, mediante un lenguaje basado en reglas, o mediante la utilización de otros formalismos de representación del conocimiento. Podemos caracterizar una KS por el tipo de operaciones que realiza en el sistema, esto es, si realiza operaciones de entrada (KS de percepción), operaciones de cognición (KS de cognición) u operaciones de salida (KS de acción).

KS: Conocimiento de resolución básico

Una KS genérica se caracteriza por su tiempo de cómputo en el peor caso (*"worst case execution time"*, —*wcet*—) si éste es conocido (puede implementar una técnica de IA no acotada). Asociada a una KS estará la codificación del conocimiento que representa la operación a realizar por dicha KS. El rango y el dominio de dicha operación varía en función del tipo de KS, pero en general una KS tendrá como entrada un conjunto de tuplas variable/valor y proporcionará como salida un conjunto de tuplas variable/valor (formando parte, todas estas tuplas, del conjunto de creencias del agente). Como se ha comentado podemos distinguir entre:

- KS de percepción, la cual, leerá del mundo los valores de las variables observables y proporcionará como salida tuplas variable-observable / valor.
- KS de cognición, la cual, operará sobre los valores observables, no observables e internos calculando valores de variables internas, de salida o no observables.
- KS de acción, la cual, actuará sobre el mundo a partir de los valores de las variables de salida previamente calculadas.

Normalmente, una KS no es empleada de forma independiente sino como parte de un "Nivel". Un *nivel* es una secuencia de KSs que deben ejecutarse

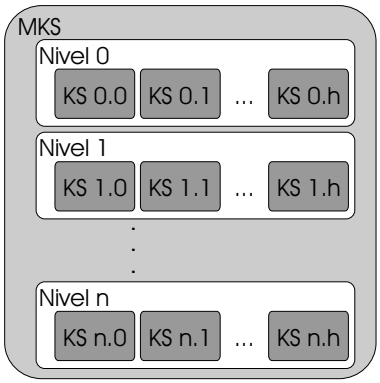

Figura 3.4: Estructura interna de una MKS.

de forma conjunta y en el orden especificado (esto permite la reutilización de KSs en diferentes niveles).

3.4.2.2. MKS (*Multiple-level Knowledge Source*)

MKS (*Multiple – level Knowledge Source*) o fuente de conocimiento múltiple, es la entidad que implementa el concepto de un *método anytime* o de un *método múltiple*, proporcionando diferentes soluciones al mismo problema con diferentes tiempos de computación y diferentes niveles de calidad.

Como se ha comentado, y tal como se puede ver en la figura 3.4, una MKS está formada por un conjunto de niveles. Un nivel está formado por un conjunto ordenado de fuentes de conocimiento. Todos los niveles de una MKS resuelven el mismo subproblema pero aportando cada uno de ellos una visión distinta de esta resolución, bien sea porque se implementan métodos múltiples, o porque se realizan diferentes refinamientos del mismo método.

La ejecución de una MKS puede ser interrumpida en cualquier momento retornando la solución del último nivel que se haya ejecutado de forma completa.

Normalmente, los primeros niveles obtienen soluciones más simples que necesitan menos tiempo de cómputo. Los niveles más complejos suelen utilizar predicciones, realizar optimizaciones, manejar reglas de control más complejas y, en general, utilizan técnicas cuyo tiempo de cómputo para el caso peor no es realista o no es conocido. Cuando la ejecución de una MKS es esencial para el buen funcionamiento del $\mathcal{AA}$, se exige que el primer nivel

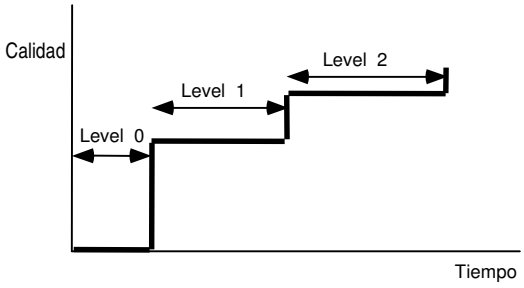

Figura 3.5: MKS *Anytime*

proporcione una solución en un tiempo de cómputo predecible y realista.

Dependiendo de la relación existente entre los diferentes niveles de una MKS, existen dos tipos de MKSs:

- MKSs *Anytime* o de Refinamiento: Un nivel del MKS refina la solución del nivel anterior. Esto supone que para que se pueda ejecutar un nivel se deben haber ejecutado completamente todos los niveles anteriores a éste, pues la solución se va refinando, consiguiendo una evolución de la calidad de la solución tal y como se muestra en la figura 3.5.
- MKSs de Métodos Múltiples: Cada uno de los niveles supone una solución alternativa al mismo subproblema, con la particularidad de que un nivel dado tendrá un tiempo estimado de cálculo en el peor caso superior o igual al de los niveles anteriores a él, ofreciendo una calidad superior o igual a la de dichos niveles. De esta manera, se puede ejecutar un nivel concreto sin que se hayan ejecutado todos los niveles anteriores. La evolución de la calidad en este caso es la que aparece en la figura 3.6.

3.4.2.3. *in-agent*

in-agent: conocimiento suficiente para resolver un problema

Un *in-agent* representa el conocimiento suficiente para determinar las acciones necesarias para resolver un determinado subproblema del problema global (este conocimiento puede incorporar técnicas de IA que proporcionan "inteligencia"para resolver el problema). El carácter de un entorno de tiempo real es periódico y tiene restricciones temporales estrictas, lo que hace que un *in-agent* se caracterice por un comportamiento periódico y disponga de

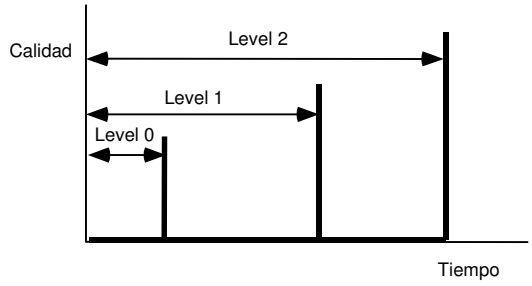

Figura 3.6: MKS de métodos múltiples

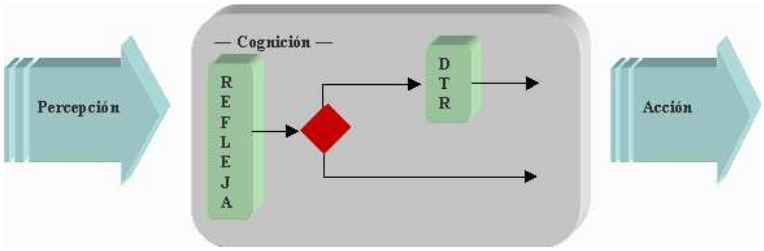

Figura 3.7: Estructura de un *in-agent*

un plazo máximo de ejecución (*deadline*) dentro del cual debe ser capaz de completar las acciones que crea más oportunas. El motivo por el cual a este componente se le ha llamado *in-agent* (*internal agent* –agente interno–) es porque puede ser considerado como un agente siguiendo la definición de agente dada por Russell [Russell 1995]. De esta manera, para proporcionar el comportamiento deseado, y según dicha definición, cada *in-agent* está formado por (figura 3.7):

- Una fase de Percepción: obtiene la información del entorno en la que está interesado el *in-agent*.
- Una fase de Cognición: está formado por dos capas:
  - Una capa Refleja: asegura una respuesta mínima (de calidad mínima en un tiempo acotado) al problema particular al que se aplica.
  - Una capa Deliberativa en Tiempo Real: calcula una respuesta razonada a través de un proceso deliberativo.

Nivel de Cognición formado por dos capas: Refleja y Deliberativa en Tiempo Real

- Una fase de Acción: lleva a cabo la última solución obtenida por el *in-agent*, sin importar en qué nivel de cognición fue calculada.

En cada activación, el *in-agent* debe decidir entre una respuesta refleja (proporcionada directamente por la capa refleja, que es siempre ejecutada hasta su finalización) y una respuesta más detallada proporcionada por la capa deliberativa en tiempo real. Esta decisión depende sobre todo del tiempo disponible para mejorar la respuesta, es decir, si es posible calcular una respuesta mejor antes de que se cumpla el plazo máximo de ejecución del *in-agent* (*deadline*).

Cabe destacar que en ocasiones, y dependiendo del problema que modele, un *in-agent* puede llegar a carecer de alguna de las capas anteriores, teniendo por tanto, *in-agents* con diferentes comportamientos.

Desde el punto de vista de conocimiento del dominio de un *in-agent*, la visión que tiene del entorno es una visión parcial representada por un conjunto de creencias. Un *in-agent* estará formado por sus propias creencias, un conjunto de restricciones temporales y una parte de percepción, una parte de cognición y otra de acción.

3.4.2.4. Agente ARTIS ($\mathcal{AA}$)

Un agente ARTIS (de acuerdo con la definición de Russell [Russell 1995]) es un agente que además es autónomo, reactivo, pro-activo y tiene continuidad temporal. De forma opcional, un agente ARTIS ($\mathcal{AA}$) puede incluir más características. Aunque el $\mathcal{AA}$ está pensado para trabajar adecuadamente bajo restricciones temporales críticas, algunas de las características opcionales que pueden incluir (tales como comunicación con otros agentes o un comportamiento social) pueden hacer imposible este comportamiento en tiempo real debido a las acciones impredecibles que suponen. Por lo tanto, es una decisión del diseñador del agente elegir qué características (y por tanto, qué comportamientos) va a tener el agente.

$\mathcal{AA}$: agente autónomo, reactivo, pro-activo y tiene continuidad temporal

El esquema básico de un $\mathcal{AA}$ estaría formado por:

1. Un conjunto de *in-agents*, los cuales representan la parte funcional del Agente ARTIS. Cada uno de estos *in-agents*, como se ha comentado anteriormente, puede dividirse en tres partes (percepción, cognición y acción). La percepción de un $\mathcal{AA}$ es la unión de las percepciones de todos los *in-agents*. La cognición de un $\mathcal{AA}$ es la unión de todas las cogniciones de los *in-agents*. Y, por último, la acción de un $\mathcal{AA}$ es la unión de

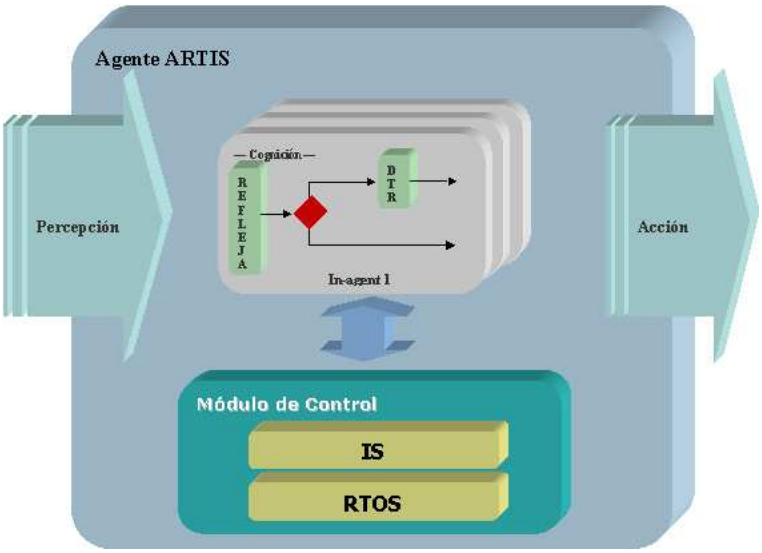

Figura 3.8: Esquema de un $\mathcal{AA}$

todas las acciones de los *in-agents*.

2. Las creencias de dicho Agente ARTIS, unión de las creencias de los diferentes *in-agents* que lo conforman.

3. Un módulo de control, compuesto por:

   - Un *Servidor Inteligente* ($\mathcal{IS}$), el cual es empleado para calcular una solución de mejor calidad en el tiempo disponible.
   - Un $\mathcal{RTOS}$ (*Real-Time Operating System*), el cual se encarga de la ejecución de los componentes críticos del conjunto de *in-agents*.

   De estas tres partes que componen el esquema básico del modelo de usuario de un $\mathcal{AA}$, el usuario debe especificar las dos primeras, siendo el Módulo de Control independiente de la aplicación y fijo. Como ya se comentó en la sección 3.4.1 de este mismo capítulo, el conocimiento del dominio se especifica por medio de un lenguaje de clases propio de tipo *LISP*. De igual manera, el conocimiento de resolución de problemas se especifica por medio de un *lenguaje de entidades*, también de tipo *LISP*. Por medio de este lenguaje, el diseñador de un $\mathcal{AA}$ debe indicar la división del problema a resolver en *in-agents*, con sus características temporales (período y plazo máximo de ejecución). La composición de cada uno de estos *in-agents*(MKSs de percepción y cognición, y KSs de

acción). Para cada MKS deberá especificar si es o no crítico, y la descomposición de los niveles de dichos MKSs en KSs. Para cada KS del agente el diseñador debe especificar un tiempo de ejecución en el peor caso y el código fuente de dicha KS.

Tal y como se especificó en el apartado 3.4.1, existe una herramienta denominada *InSiDE* que permite la especificación tanto del conocimiento del dominio como del conocimiento de resolución de un $\mathcal{AA}$ por medio de una interfaz gráfica que abstrae de la complejidad de los diferentes lenguajes mencionados (de clases y de entidades, respectivamente).

3.5. Modelo de Sistema

El Agente ARTIS especificado por medio de la herramienta *InSiDE –Integrated Simulation and Development Environment–* [Julián 2000] siguiendo los lenguajes comentados anteriormente es transformado en los siguientes módulos de bajo nivel:

- Un módulo $\mathcal{RTOS}$ (sistema operativo de tiempo real): es independiente del resto de la especificación del agente pues no varía sea cual sea ésta.
- Una memoria global ($\mathcal{KDM}$ *–Knowledge Data Manager–*): se obtiene mediante la compilación de la especificación de clases realizada por medio del lenguaje de clases.
- Un conjunto de tareas de bajo nivel: cada *in-agent* es traducido en una de estas tareas de bajo nivel. Estas tareas tienen el mismo período y plazo máximo de ejecución que su *in-agent* correspondiente. Una tarea de este tipo está dividida en tres partes que se corresponden con las que constituyen un *in-agent*:
  - Parte obligatoria (*mandatory*): incluye los procesos de percepción, la modificación del conjunto de creencias y la selección de la acción refleja a realizar.
  - Parte opcional (*optional*): se corresponde con el proceso de selección y ejecución de la parte cognitiva.
  - Parte final (*final*): coincide con la parte de acción de un *in-agent*.

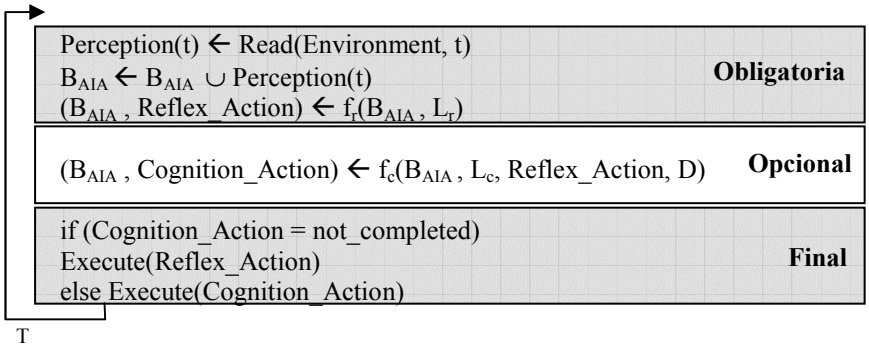

Figura 3.9: Correspondencia entre el ciclo de un *in-agent* y las partes de una tarea de bajo nivel.

En la figura 3.9 se puede observar la transformación de un *in-agent* en una tarea de bajo nivel con sus correspondientes partes claramente diferenciadas.

Cada una de estas tareas va a venir caracterizada por los siguientes parámetros: Prioridad, Período, *Deadline* y tiempo de ejecución en el peor caso (*wcet*) de cada una de sus partes.

3.5.1. Análisis de la Planificabilidad

La principal característica diferenciadora de la arquitectura de $\mathcal{AA}$ con respecto a otras arquitecturas de agente híbridas es su funcionamiento en entornos con restricciones temporales críticas. Esto es posible gracias a un análisis *off-line* de la planificabilidad de las tareas con restricciones temporales críticas que formen parte del modelo de sistema del agente. Este análisis permite comprobar en la fase de diseño del agente si se van a poder cumplir las restricciones temporales críticas asociadas al mismo. Para ello se aplica un análisis de garantía al conjunto de tareas de bajo nivel teniendo en cuenta que el cumplimiento de las restricciones críticas depende tan sólo de la ejecución de las partes obligatoria y final de las tareas.

Para realizar dicho análisis, se tiene en cuenta que la ejecución de dichas tareas las va a controlar el denominado *Planificador de Primer Nivel* utilizando para ello una política de planificación basada en prioridades fijas expulsivas (*Fixed Priority Pre-emptive Scheduling Policy*) [Audsley 1995]. De acuerdo a esta política se asigna una prioridad base a cada tarea, asignación que en este

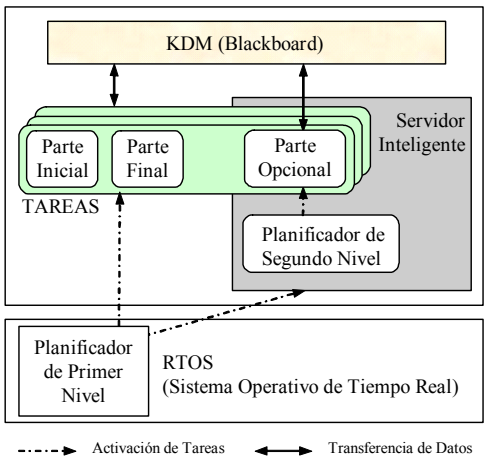

Figura 3.10: Modelo de Sistema de un $\mathcal{AA}$

caso sigue el método denominado *deadline monotonic*, que consiste en realizar la asignación de acuerdo a su plazo máximo de ejecución o *deadline*, de forma que dadas dos tareas, será más prioritaria aquella que tenga un menor *deadline*. En cada momento se ejecuta la tarea activa de mayor prioridad, de forma que si se activa una tarea con prioridad superior a la que se está ejecutando en un instante dado, la expulsa del procesador pasándose a ejecutar la más prioritaria.

Según lo dicho, las partes inicial u obligatoria y final de las tareas van a ser planificadas por el Planificador de Primer Nivel (*First-Level Scheduler*, –$\mathcal{FLS}$–), mientras que las partes opcionales de las tareas (así como las tareas aperiódicas acríticas que se proponen como parte de este trabajo) serán planificadas por el Planificador de Segundo Nivel (*Second-Level Scheduler*, –$\mathcal{SLS}$–) que forma parte del Servidor Inteligente ($\mathcal{IS}$) (ver figura 3.10).

El test de planificabilidad se aplica únicamente a las partes críticas de la tarea, es decir, se toma como tiempo de ejecución del caso peor (*wcet*) de la tarea, la suma de los *wcet* de su parte obligatoria y su parte final.

El módulo $\mathcal{RTOS}$, del que forma parte el $\mathcal{FLS}$, es el responsable de mantener en tiempo de ejecución, para cada nivel de prioridad existente, la cantidad de tiempo disponible que puede ser utilizada por la parte *inteligente* u opcional del sistema sin comprometer los plazos máximos de ejecución de las tareas con restricciones temporales críticas. Este tiempo, denominado *slack* u holgura, se calcula por medio de una adaptación del algorit-

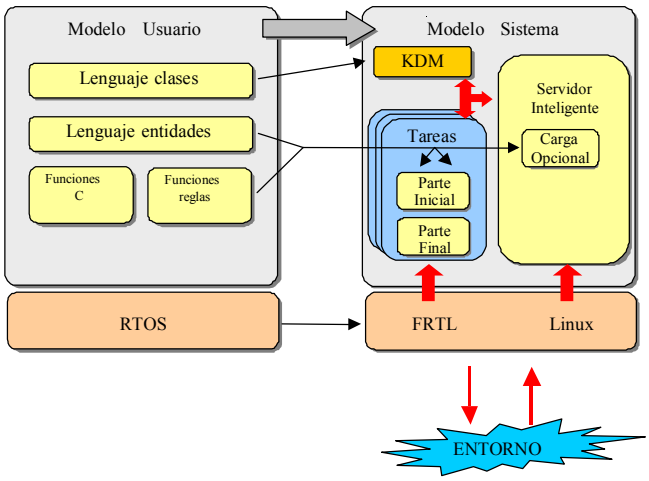

Figura 3.11: Correspondencia entre Modelo de Usuario y de Sistema.

mo de *Extracción de Holgura Aproximado Dinámico*[Davis 1993a, Davis 1993b, García-Fornes 1996] (*Dynamic Approximate Slack Stealing*, *–DASS–*).

Todos los componentes opcionales se ejecutan en el sistema a través de un proceso servidor (el $\mathcal{IS}$). En principio, esta componente no afecta al análisis off-line, ya que siempre se ejecuta en tiempo de holgura.

A este nivel, el $\mathcal{IS}$ es visto como una tarea especial que siempre se ejecuta a la prioridad más alta pero sólo le está permitido ejecutarse en intervalos de tiempo pre-establecidos (tiempo de holgura).

En la figura 3.11 se pueden observar los componentes de los modelos de usuario y de sistema de la arquitectura de un $\mathcal{AA}$, junto con la relación de traducción entre los distintos modelos.

El modelo de sistema de un agente ARTIS es directamente ejecutable en un sistema operativo de tiempo real propio basado en RT-Linux denominado *Flexible RT-Linux* ($\mathcal{FRTL}$) [Terrasa 2000, Terrasa 2002] que ofrece todos los servicios y el modelo de tareas de bajo nivel necesario para dicho modelo de sistema del agente ARTIS.

3.6. El Módulo de Control

El Módulo de Control de un agente ARTIS, de acuerdo a la estructura clásica de un sistema de *blackboard* tradicional, correspondería con el módulo de control del mismo. Es decir, es la parte encargada de controlar cómo y cuándo se van a ejecutar el resto de partes del sistema.

El *Módulo de Control* de un $\mathcal{AA}$ es la parte encargada de decidir cuándo razonar y cuándo actuar. Es el encargado de asegurar que siempre vaya a existir una respuesta del agente ante un cambio en su entorno y de que esa respuesta sea la mejor posible de acuerdo al tiempo que tiene para reaccionar así como al conocimiento del que dispone en ese momento.

Así, un $\mathcal{AA}$ se puede ver como un agente con una capa reactiva, cuya ejecución está asegurada, y que proporciona una respuesta rápida y simple que no tiene por qué ser la mejor en esa situación.

Además, un $\mathcal{AA}$ dispone de una segunda capa deliberativa cuyo objetivo es tratar de mejorar la respuesta de la capa reactiva. La ejecución de esta última capa viene condicionada a que exista tiempo disponible para su ejecución y a que el $\mathcal{AA}$ crea oportuno ejecutarla.

De esta forma, el Módulo de Control de un $\mathcal{AA}$ se va a dividir a su vez en dos submódulos, $\mathcal{RTOS}$ e $\mathcal{IS}$:

$\mathcal{RTOS}$: planificar elementos con restricciones críticas

$\mathcal{RTOS}$ (*Real-Time Operating System*): es el módulo encargado de controlar la ejecución de la capa reactiva del $\mathcal{AA}$, formada por las tareas con restricciones críticas del sistema. Este módulo debe controlar la ejecución de todos aquellos elementos que tengan restricciones temporales críticas de tal forma que puedan cumplirlas. Una vez satisfechas estas restricciones, pueden quedar intervalos de tiempo en los que el procesador esté ocioso intercalados entre las ejecuciones de los elementos críticos. Estos intervalos podrán ser utilizados por la segunda parte del módulo de control, el Servidor Inteligente ($\mathcal{IS}$), para realizar diversas funciones cuyo objetivo último es la mejora de la calidad de la respuesta del $\mathcal{AA}$.

$\mathcal{IS}$: ejecutar la capa deliberativa

$\mathcal{IS}$ (*Intelligent Server*): es el módulo encargado de controlar la ejecución de la capa deliberativa del agente.

3.6.1. $\mathcal{RTOS}$

Como se ha comentado antes, este módulo tiene como misión principal asegurar el cumplimiento de las restricciones críticas del agente. Para ello, va a realizar las siguientes funciones:

- Controlar la ejecución de todas las tareas con restricciones temporales críticas asegurando que no se incumpla ninguna. Para ello se va a utilizar la política de planificación de tareas *Fixed Priority Pre-emptive Scheduling Policy*, compatible con el análisis de planificabilidad del agente.
- Ceder el control del agente al $\mathcal{IS}$ durante el tiempo de holgura del sistema. El tiempo de holgura de un sistema de tiempo real se define como la cantidad de tiempo que el sistema puede dedicar a otros menesteres sin comprometer el cumplimiento de las restricciones temporales del sistema. Este tiempo lo calcula $\mathcal{RTOS}$ después de la ejecución de la parte inicial de una tarea utilizando el *Algoritmo de Extracción de Holgura Aproximado Dinámico* [Davis 1993a, Davis 1993b, García-Fornes 1996] que es compatible con el análisis off-line de la planificabilidad.
- Informar al $\mathcal{IS}$ del estado de ejecución de las tareas críticas del sistema. En concreto debe informarle cuando una tarea crítica ha finalizado su parte obligatoria (para que tenga en cuenta que puede ejecutar las partes opcionales de esa tarea si lo considera oportuno) y cuando una tarea ha finalizado su parte final (para que ya no intente ejecutar ninguna parte opcional de esa tarea). Esta comunicación se realiza mediante un mecanismo de mensajes ad-hoc entre $\mathcal{RTOS}$ e $\mathcal{IS}$.

3.6.2. El Servidor Inteligente ($\mathcal{IS}$)

Como su propio nombre indica es la parte del módulo de Control de un $\mathcal{AA}$ encargada de gestionar la inteligencia de dicho agente. Funciones principales:

$\mathcal{IS}$: gestiona la inteligencia del $\mathcal{AA}$

- Gestionar la activación / desactivación de las partes opcionales de las tareas críticas.
- Planificar la ejecución de la carga opcional del agente con el objetivo de maximizar la calidad de la respuesta.

3.6.2.1. Ciclo de Control

Funcionamiento dirigido por eventos

Se puede decir que el $\mathcal{IS}$ está implementado siguiendo una arquitectura de *blackboard* de control [Hayes-Roth 1988], similar a la del tipo utilizada en algunos módulos de control de los sistemas de blackboard tradicionales. Es, por tanto, un módulo cuyo funcionamiento está dirigido por eventos. En concreto, éste implementa una variación del *satisficing cycle* [Hayes-Roth 1993], de tal manera que el ciclo básico se puede describir de la siguiente forma:

```
Para siempre hacer {
    Trigger();
    Condition_Testing();
    Rating();
    scheduledAG = Schedule();
    Interpret( scheduledAG );
 }
```

***Trigger*:** Recepción de mensajes, y gestión de Niveles activos:

Control de qué niveles de las distintas MKSs (de ejecución no obligatoria para el sistema) que puedan estar activas, pueden ser elegibles para su ejecución en este ciclo. Para ello, debe tener en cuenta que si la MKS implementa métodos múltiples, todos los niveles serán elegibles, mientras que si no es así, tan sólo el siguiente nivel al último ejecutado con éxito será elegible. Además hay que tener en cuenta, en este último caso, que en cuanto un nivel no se ejecute con éxito, ya no se seguirá ejecutando más niveles de esa MKS.

Las entidades anteriores pueden estar activas bien porque se han activado por un evento en este ciclo o porque continúan activas del ciclo anterior. Todas las entidades elegibles se insertarán en una agenda (*triggering subagenda*).

***Condition Testing*:** Comprobación de condiciones de ejecución (principalmente se basa en comprobar que no se ha cumplido el plazo máximo de ejecución que tienen asociado):

Todas aquellas entidades de la *triggering subagenda* que cumplen todas las condiciones para poder ser ejecutadas son insertadas en la *condition-testing subagenda*.

***Rating*:** Ordenación de entidades activas:

Las entidades de la agenda *condition-testing subagenda* son evaluadas de acuerdo a la política de planificación de segundo nivel que esté activa en este momento. Utilizando ese valor calculado, son insertadas ordenadamente en la *rating subagenda*.

***Schedule*:** Selección de la siguiente entidad a ejecutar (*scheduledAG*):

De entre las entidades que estén en la *rating subagenda*, se selecciona una para su posterior ejecución de acuerdo al algoritmo de planificación de segundo nivel y a consideraciones temporales: tiempo estimado de ejecución (medio o en el peor caso, según el algoritmo de segundo nivel), tiempo sobrante disponible en el sistema sin que se incumplan las restricciones temporales de las partes críticas del sistema, plazo máximo de ejecución (si se dispone de él).

***Interpret*:** Ejecución de la entidad seleccionada (*scheduledAG*):

Lanza a ejecución la entidad seleccionada en la fase *Schedule*, durante un tiempo determinado (que deberá ser inferior al tiempo que le quede de ejecución al $\mathcal{IS}$, que será lo que no haya consumido aún del tiempo de holgura del sistema antes de la ejecución del $\mathcal{IS}$).

3.6.2.2. Planificador de Segundo Nivel

Como ya se ha comentado, el *Planificador de Segundo Nivel* está constituido por dos fases, la de valuación (*rating*) y la de planificación (*schedule*), comentadas brevemente en el apartado anterior. Dentro de la arquitectura de $\mathcal{AA}$ se incorporan un conjunto de políticas de planificación de segundo nivel, permitiéndole al diseñador decidir qué política utilizar, de acuerdo al comportamiento que se quiera (de más reactivo a más deliberativo). Las políticas implementadas son las que se presentan a continuación.

3.6.2.2.1. Políticas Reactivas

Tasan todos los niveles activos de la *condition-testing subagenda* eligiendo para su ejecución aquel que según su criterio es mejor en el instante actual:

EDF (*Earliest* Deadline *First*): Esta estrategia requiere que los niveles actuales de los *in-agents* de la *Rating Subagenda* se encuentren ordenados de menor a mayor *deadline*.

El algoritmo elige como nivel para ejecutar aquel que tenga un menor *deadline*. Si hay varios niveles con ese *deadline*, se elige el nivel actual del *in-agent* de mayor importancia. Si aún así hay varios niveles entre los que elegir, se elige el nivel del *in-agent* que tenga una calidad menor. Si todavía hay varios niveles candidatos, se selecciona el de menor duración. Y, por último, si aún hay más de un nivel a elegir, se elige el primer nivel encontrado.

Objetivo: ejecutar el mayor nº posible de niveles

El objetivo de esta estrategia es ejecutar el mayor número posible de niveles.

BIF (*Biggest Importance First*): La estrategia requiere que los niveles de la *Rating Subagenda* se encuentren ordenados de mayor a menor importancia.

Se elige como nivel actual para ejecutar aquel que pertenezca al in-agent de mayor importancia de la *Rating Subagenda*. Si hay varios *in-agents* con esa importancia, se elige el nivel actual del *in-agent* de menor calidad. Si todavía hay varios niveles entre los que elegir, se selecciona el de menor *deadline*. Si aún quedan varios niveles candidatos se elige el de menor duración y, finalmente, si hay más de un nivel para elegir, se selecciona el primer nivel encontrado.

Objetivo: Ejecutar los niveles de mayor importancia

Esta estrategia tiene como fin ejecutar los niveles de los *in-agents* de mayor importancia.

La *calidad* de un *in-agent* viene definida por la siguiente fórmula:

$$\frac{importancia * (n^{o}\ niveles\ del\ agente\ ejecutados)}{n^{o}\ total\ de\ niveles\ del\ agente}$$

ELDF (*Earliest Level* Deadline *First*): Esta estrategia es una pequeña variante de la EDF y sigue exactamente los mismos criterios, la única diferencia radica en que los *deadlines* manejados son distintos. Se cálcula el *deadline* asociado a cada nivel del MKS descontando del *deadline* del MKS los tiempos de cómputo para el peor caso de los niveles posteriores. Los niveles estarán ordenados de menor a mayor *deadline*.

El algoritmo elige como nivel para ejecutar aquel que tenga un menor *deadline*. Si hay varios niveles con ese *deadline*, se elige el nivel actual del

MKS de mayor importancia. Si aún así hay varios niveles entre los que elegir, se elige el nivel del MKS que tenga una calidad mayor. Si todavía hay varios niveles candidatos, se selecciona el de menor duración. Y, por último, si aún hay más de un nivel a elegir, se elige el primer nivel encontrado.

Objetivo: Ejecutar el mayor nº posible de niveles

El objetivo de esta estrategia es ejecutar el mayor número posible de niveles.

HQF (*High Quality First*): La estrategia requiere que los niveles activos de los MKS activos estén ordenados de mayor a menor calidad.

Se elige como nivel actual para ejecutar el de mayor calidad de los MKS de la *rating subagenda*. Si hay varios niveles con esa calidad, se selecciona el de menor *deadline*. Si aún quedan varios niveles candidatos, se elige el de menor duración. Y, finalmente, si hay más de un nivel para elegir, se selecciona el primer nivel encontrado.

Objetivo: Incrementar la calidad de la respuesta lo más rápido posible

El objetivo de esta estrategia es incrementar la calidad de la respuesta lo más rápido posible

HSF (*High Slope First*): La estrategia requiere que los niveles de los MKS de la *Rating Subagenda* se encuentren ordenados de mayor a menor pendiente de la función lineal asociada a cada nivel resultado de transformar el escalón correspondiente en una función lineal.

Se elige como nivel actual para ejecutar el de mayor pendiente. Si hay varios niveles con la misma pendiente, se elige el nivel de mayor calidad. Si todavía hay varios niveles entre los que elegir, se selecciona el de menor *deadline*. Si aún quedan varios niveles candidatos se elige el de menor duración. Y, finalmente, si hay más de un nivel para elegir, se selecciona el primer nivel encontrado.

Objetivo: Ejecutar los niveles con mejor relación calidad / tiempo de ejecución

Esta estrategia tiene como fin ejecutar los niveles que presentan mejor relación calidad tiempo de ejecución, en la esperanza de que se ejecute el mayor número posible de niveles que más calidad proporcionan.

MU (*Marginal Utility*): La estrategia requiere que los niveles de los MKSs de la *Rating Subagenda* se encuentren ordenados de mayor a menor utilidad marginal.

Dada una función de utilidad, la utilidad marginal [Etzioni 1991] se define como la derivada de la función de utilidad respecto a una variable de coste (tiempo en este caso).

Esta medida permite incorporar lo que en el campo económico se conoce como coste de oportunidad. Es decir, se tiene en cuenta que al ejecutar un nivel L_i se obtiene una utilidad, pero se deja de conseguir la utilidad proporcionada por otros niveles que se hubiesen ejecutado en el tiempo consumido por L_i.

Objetivo: Equilibrar la utilidad del plan y el esfuerzo de ejecutarlo

Mediante el uso de la utilidad marginal, se le permite al planificador equilibrar la utilidad del plan y el esfuerzo de ejecutarlo.

3.6.2.2.2. Políticas Deliberativas

Construyen un plan con los niveles activos de la *condition-testing subagenda*:

DBM (*Dean & Boddy Modified*): Basada en el algoritmo de planificación deliberativa de Dean y Boddy [Boddy 1994].

De acuerdo al tiempo de *slack* disponible y a los diferentes *deadlines* de los niveles activos, crea un plan asignando tiempo desde el fin del *slack* hacia el principio, de acuerdo a los *deadlines* de dichos niveles.

Objetivo: ejecutar el mayor nº de niveles

El objetivo de esta política es ejecutar el mayor número de niveles, aplazando la ejecución de aquellos niveles con un *deadline* más grande.

AEDF (*Anytime Earliest* Deadline *First*): Esta política deliberativa crea un plan, seleccionando los niveles a ejecutar mediante un criterio similar al de la política reactiva EDF, con la diferencia de que donde la EDF utilizaba la importancia para discriminar entre los niveles, esta política utiliza la denominada *utilidad estimada* del MKS ($\hat{U}_{MKS}$):

$$\hat{U}_{MKS} = (\sum(Utilidad\ Nivel\ ejecutado)) * importancia\ MKS$$

Objetivo: Maximizar la utilidad total, teniendo en cuenta la degradación

Para realizar esta planificación, cada agente posee una función de degradación que calcula la pérdida de valor de la solución conforme pasa el tiempo.

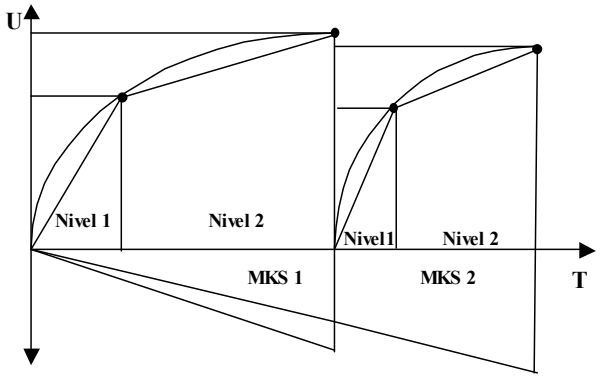

Figura 3.12: Ejemplo de plan de ejecución según utilidad estimada (curva interpolada) y función de degradación (función lineal decreciente en el tiempo).

Así, el plan que calcula esta política trata de maximizar la utilidad total, teniendo en cuenta esta degradación. Esta idea esta basada en los trabajos de Horvitz [Horvitz 1991].

AHUF (*Anytime Highest Utility First*): Al igual que ocurre con la política anterior, en esta política se crea un plan utilizando para ello un criterio de selección de niveles que es una variación del de una política reactiva (en este caso la BIF) cambiando las referencias a la importancia por referencias a la utilidad estimada.

Objetivo: Plan con el incremento más rápido de la utilidad estimada

SSS (*Slack Sizing Scheduling*): Esta política (también denominada *Planificación de Ajuste de Huecos de Tiempo*) se basa en las ideas base de dos políticas de planificación, las de Dean y Boddy [Boddy 1994] (planificar hacia atrás en el tiempo) y en la idea de Etzioni [Etzioni 1991] (utilizar el ratio calidad/duración de los niveles para discriminar entre los mismos).

Además, teniendo en cuenta las peculiaridades propias del $\mathcal{IS}$, en concreto, que se le va a invocar en distintos momentos, con distintos tiempos de ejecución (*slack*), esta política trata de adaptar los huecos de *slack* de que dispone. Para ello, va a poder solicitar a $\mathcal{RTOS}$ que se adelanten las partes finales de las tareas para modificar el tamaño de los siguientes huecos de *slack* que reciba (incrementando su tamaño). Esta petición tiene sentido, ya que el planificador de segundo nivel puede realizar una estimación de la cantidad de huecos que habrá debido a que conoce el número de tareas activas. Lo que trata de hacer es una estimación del tamaño del *slack* que va a tener disponible basándose en que cono-

Objetivo: Maximizar la calidad obtenida

ce los *deadlines* así como los *wcet* (*worst case execution time*) de las partes finales de las tareas activas.

El objetivo de esta política es maximizar la calidad obtenida.

3.7. Conclusiones

En este capítulo se ha presentado la arquitectura de agente ARTIS en su estado previo al presente trabajo. Esta arquitectura híbrida vertical adolecía de la falta de flexibilidad propia de los sistemas de tiempo real tradicionales, siendo esta característica incluso presente en toda la arquitectura, es decir, tanto en la parte que trabaja con restricciones temporales críticas como en la parte que incorpora la *inteligencia*, o deliberación.

Esta falta de flexibilidad incide en la eficiencia de los sistemas que se pueden construir por medio de dicha arquitectura.

Es por esto que se hacía necesaria una revisión de dicha arquitectura, incorporándole además unas capacidades de meta-razonamiento que le permitan adaptarse a cambios en las situaciones a las que deba enfrentarse.

Parte III

Descripción de Aportaciones

4

Visión Global del Trabajo

"The aesthetics of knowledge are at the heart of several key polarities in the history of Chinese thought...
A person can possess a wide and deep knowledge of factual material, but unless that person has a kind of ***insight, judgment, and above all good taste in the way that this knowledge is used or presented****, that person cannot be said to possess* ***shih****."*
"They Have a Word for It"
– Howard Rheingold

4.1. Introducción

Este capítulo retoma el tema de la motivación y objetivos del presente trabajo, entrando en detalle en los mismos una vez que se ha presentado el estado previo en los capítulos anteriores.

Además, se presentan las distintas aportaciones realizadas con el presente trabajo, y que serán extendidas en los siguientes capítulos de la presente parte.

4.2. Motivación

De entre todas las arquitecturas de agente presentadas en apartados anteriores, no existe ninguna que funcione en un entorno de tiempo real con restricciones temporales críticas. Es decir, aunque hay algunas que dicen trabajar en tiempo real, no son capaces de asegurar que se cumplirán las restricciones temporales que posean (es lo que dentro de la terminología de los sistemas en tiempo real se conoce como *soft real-time systems*).

En contraposición con estas arquitecturas, un $\mathcal{AA}$ va a poder asegurar en la fase de diseño el cumplimiento de sus restricciones temporales críticas, permitiéndole funcionar en entornos con restricciones temporales críticas (*hard real-time systems*).

Según se ha visto en el capítulo anterior, en el modelo de usuario, un $\mathcal{AA}$ se define en base a un conjunto de *in-agents* que se traducen en el modelo de sistema a un conjunto de tareas. La ejecución de estas tareas es gestionada por dos planificadores, uno al cargo del cumplimiento de las restricciones críticas del sistema y el otro del aprovechamiento del tiempo restante para la ejecución de las partes no esenciales. El primero de estos planificadores utiliza una política de planificación fija, mientras que el segundo utiliza la política que se establece en la fase de diseño. En ambos casos, estas políticas se mantienen fijas durante la ejecución del agente.

De esta manera, debido a que el conjunto de *in-agents* / tareas es único y constante, que las políticas de planificación que se usan también lo son, y que no existe ningún mecanismo adicional que permita adaptar el funcionamiento del agente a cambios en el entorno, se pone de manifiesto que existen una serie de limitaciones dentro de la arquitectura de $\mathcal{AA}$ que se plasman en la falta de capacidad para poder adaptarse a cambios significativos en el entorno, y una falta de flexibilidad en la gestión del tiempo disponible. Estos problemas son los que motivan el presente trabajo.

Tras un estudio inicial de la arquitectura de $\mathcal{AA}$ a bajo nivel, se ha llegado a la conclusión de que las limitaciones teóricas anteriores se plasman en las siguientes carencias:

1. El Módulo de Control no era capaz de adaptarse a cambios en el entorno por:

   - La incapacidad de modificar su comportamiento en ejecución, o lo que es lo mismo, carecer de la habilidad de *adaptar* su comportamiento a cambios significativos (del entorno o interno).
   - La imposibilidad de incorporar *meta-conocimiento* en el momento del diseño (tan sólo se podía elegir el algoritmo de planificación de segundo nivel que se usaría durante toda la ejecución del agente).

2. De igual forma, el $\mathcal{RTOS}$ se reducía a un planificador de primer nivel encargado de gestionar la ejecución tanto de las partes críticas del $\mathcal{AA}$ como del $\mathcal{IS}$, junto con el modelo de tareas adecuado.

3. El estado del $\mathcal{IS}$ estaba reducido a un planificador de segundo nivel encargado de gestionar la ejecución de las partes opcionales de los *in-agents* críticos del $\mathcal{AA}$.

4. Como ejemplo de la simplicidad que existía entre $\mathcal{IS}$ y $\mathcal{RTOS}$ el modelo de comunicación entre ambos módulos era muy restringido. Este modelo se reducía a tan sólo un posible mensaje de comunicación entre dichas partes del Módulo de Control. Este mensaje era generado por el $\mathcal{RTOS}$ para indicar al $\mathcal{IS}$ que podía planificar las partes opcionales de un *in-agent* concreto (porque había finalizado la ejecución de su parte inicial).

   Esta, por tanto, era la única posibilidad de comunicación entre estos dos planificadores independientes, estableciendo la mínima comunicación que era necesaria para permitir el funcionamiento del sistema, ya que ambos planificadores eran muy simples y su funcionamiento no estaba muy coordinado.

5. Hay que tener en cuenta que en la situación previa al presente trabajo todos los *in-agents* eran periódicos y permanecían activos durante toda la vida del agente. Esto conseguía que el agente presentase un comportamiento periódico y poco flexible a nuevos cambios en el entorno.

Por otra parte, cabe destacar que el trabajo aquí desarrollado además de permitir paliar las carencias de la arquitectura que se acaban de exponer, permite la incorporación de la capacidad de comunicación dentro de un agente ARTIS, lo que le dota al $\mathcal{AA}$ de sociabilidad y, como consecuencia permite su incorporación en un sistema multi-agente. Esta es la base de un trabajo paralelo al que aquí se presenta en el que se está desarrollando

una arquitectura/plataforma de sistemas multi-agente de tiempo real basados en la arquitectura de agente ARTIS a la que se ha denominado *SIMBA*, "**SI**stema **M**ulti-agente **B**asado en **A**RTIS" y que se describe en [Julián 2002b, Soler 2002, Soler 2003, Carrascosa 2003b].

4.3. Objetivo

El objetivo principal del presente trabajo es dotar a la arquitectura de *AA* de un mecanismo de *meta-razonamiento* para incorporar una serie de nuevas capacidades que le permitan al agente adaptarse a los cambios en el entorno de la forma más eficiente posible (para lo que se flexibiliza y mejora también el aprovechamiento del tiempo disponible por parte del Módulo de Control). Estas capacidades son las siguientes:

1. Detección de situaciones significativas de cambio.
2. Adaptación del comportamiento del agente.
3. Adaptación del proceso de razonamiento del agente, teniendo en cuenta las restricciones temporales del mismo, por medio de la introducción de los siguientes conceptos:
   a) El Grado de Reactividad.
   b) El Grado de Introspección.
   c) El Foco de Atención.
4. Se dejará preparada la arquitectura para la futura incorporación de la capacidad de aprendizaje.

En general, la primera de estas capacidades permitirá que el agente detecte cuando debe adaptarse mientras que el resto de capacidades van a aportar las distintas formas en las que el agente podrá llevar a cabo dicha adaptación.

La incorporación de estas capacidades supone la inclusión de un nuevo tipo de conocimiento en el modelo de usuario (con sus contrapartidas en el resto de modelos), y es el *meta-conocimiento* o conocimiento sobre el conocimiento. Para incluir este nuevo conocimiento se le debe dotar al desarrollador

de sistemas ARTIS de un mecanismo que le permita incorporar todo el *meta-conocimiento* propio del problema al que va a aplicar la arquitectura. Este último objetivo se traduce en la necesidad de un lenguaje de usuario (*lenguaje de control*) y de los mecanismos asociados para la gestión del conocimiento especificado mediante dicho lenguaje.

Por otra parte, para hacer corresponder con mayor certeza a las funciones que desempeña, se han rebautizado los dos componentes del módulo de control de la siguiente forma [Carrascosa 2003a]:

- El $\mathcal{RTOS}$ pasa a llamarse *Servidor Reflejo* (*Reflex Server —$\mathcal{RS}$—*) al encargarse como función principal del control de la capa reactiva del $\mathcal{AA}$.
- El $\mathcal{IS}$ pasa a llamarse *Servidor Deliberativo* (*Deliberative Server —$\mathcal{DS}$—*) pues como función principal gestiona la capa deliberativa del $\mathcal{AA}$.

A continuación, en las siguientes secciones se pasa a detallar las nuevas capacidades anteriormente expuestas, siguiendo el orden presentado con la salvedad de que se ha dejado en último lugar la capacidad de detección de situaciones significativas, por hacer uso del resto de capacidades.

4.4. Gestión de Diferentes Comportamientos

4.4.1. Introducción

Para poder implementar la mencionada gestión de comportamientos, y de forma similar a como ocurre en un ser racional, se van a distinguir dos tipos de comportamientos:

Comportam. Reflejos (*innatos* o *adquiridos*) y *Comportam. Deliberativos*

Comportamientos Reflejos: En este tipo de comportamientos el sistema reacciona frente a determinados estímulos externos sin necesidad de un proceso deliberativo. Estos comportamientos se suelen representar por conjuntos de pares *situación-acción*.

De acuerdo al origen de estos comportamientos, se dividen en:

Innatos: Corresponden a aquellos comportamientos reflejos que se tienen desde la creación del sistema. Por ejemplo, el acto reflejo de

cualquier animal de retirar una extremidad del lugar donde se encuentra cuando se le causa dolor sobre ella.

Estos comportamientos no son adquiridos, sino heredados, y están formados por conocimiento proporcionado por el desarrollador.

Adquiridos: Estos comportamientos son adquiridos por el sistema en el transcurso de su vida, mediante una compilación de experiencias anteriores. Por ejemplo, el acto reflejo de un portero de fútbol frente a un balón que se acerca a él a gran velocidad sería diferente del de alguien que no hubiese practicado nunca ese deporte ni adquirido por tanto dicho comportamiento reflejo. Esto le permite al sistema adaptarse mejor a nuevas situaciones y le proporciona un mayor grado de autonomía.

Comportamientos Deliberativos: En este tipo de comportamientos el sistema realiza un proceso deliberativo para calcular la acción/acciones o plan a seguir.

4.4.2. Gestión de Comportamientos Reflejos

La introducción de diferentes comportamientos reflejos dentro de un $\mathcal{AA}$ supone la realización de toda una serie de cambios a todos los niveles dentro del mismo, que incluyen desde la forma en la que se diseña un $\mathcal{AA}$ a la forma en la que el módulo de Control del mismo decide qué se debe ejecutar en cada ocasión. Además, en este caso se distingue entre comportamientos reflejos innatos y comportamientos reflejos adquiridos, exigiendo un tratamiento distinto cada uno de estos tipos.

4.4.2.1. Comportamientos Reflejos Innatos

Como ya se ha comentado, se va a considerar este tipo de comportamientos como aquellos comportamientos reflejos que el sistema tiene desde su nacimiento. Por tanto, van a ser comportamientos que han de ser especificados por el usuario programador en la fase de diseño del agente. De esta manera, en el modelo de usuario, se realizan las modificaciones pertinentes para poder especificar comportamientos, y que se recogen en el capítulo 6. Para poder gestionar estas extensiones, también se ha extendido el modelo de sistema, como se recoge en el capítulo 7.

4.4.2.2. Comportamientos Reflejos Adquiridos

Este tipo de comportamientos son los que el agente podría adquirir por medio de un algoritmo de aprendizaje apropiado, a partir de la información que va adquiriendo durante su ejecución.

Parte del trabajo aquí desarrollado ha consistido en dejar preparada la arquitectura para la fácil inclusión de dicho aprendizaje dentro de la misma. De esta manera, la implementación dentro del modelo de sistema del concepto de comportamiento permite la inclusión de este tipo. Además, el objetivo final es que estos comportamientos no sólo se puedan aprender, sino también olvidar, si no resultan útiles.

4.4.3. Gestión de Comportamientos Deliberativos

Comportam. Deliberativos **no** son sólo las partes opcionales de *Comportam. Reflejos*

Al igual que ocurre con la gestión de comportamientos reflejos, en este caso se tienen que extender tanto el modelo de usuario como el modelo de sistema del agente tal y como se detallará en los capítulos 6 y 7 respectivamente.

Según la procedencia del conocimiento de resolución de problemas que lo forman, existen dos tipos de comportamientos deliberativos:

- Aquellos determinados por el comportamiento reflejo que tenga el agente en cada momento. Este tipo será el de la mayor parte de los comportamientos deliberativos, ya que el proceso deliberativo consistirá, principalmente, en refinar la solución calculada por el comportamiento reflejo.
- Aquellos independientes del comportamiento reflejo del agente, y que serán activados y ejecutados en respuesta a eventos significativos. Estos comportamientos deliberativos responderán a una necesidad de adaptación o de pro-actividad por parte del agente.

4.5. Adaptación del proceso de razonamiento

Dentro de los sistemas de tiempo real, el recurso más importante es el procesador. Si además, el sistema es de inteligencia artificial en tiempo real, como

es el caso del $\mathcal{AA}$, este recurso adquiere si cabe mayor importancia al tener algoritmos con un consumo de tiempo de procesador muy grande y, en algunos casos, no acotado. Es por tanto esencial, en este tipo de sistemas, hacer un uso óptimo de dicho recurso. Más aún, si el sistema trabaja con restricciones temporales críticas, en las que el proceso de razonamiento que pueda realizar el sistema va a tener que estar limitado al tiempo disponible una vez asegurado el cumplimiento de dichas restricciones críticas, se convierte en fundamental la gestión de este recurso.

Por otra parte, el asignar tiempo de procesador para un determinado proceso de razonamiento, máxime cuando el proceso al que se le asigna tiempo puede no estar acotado y al consumir el tiempo asignado no haber conseguido ningún cálculo de utilidad para el sistema.

De esta manera, con el objetivo de flexibilizar y mejorar el aprovechamiento del tiempo disponible por parte del Módulo de Control de un $\mathcal{AA}$, se introducen una serie de conceptos que van a permitir adaptar el proceso de razonamiento, siempre teniendo en cuenta que el sistema deberá trabajar en un entorno con restricciones temporales críticas. Estos conceptos son el de *grado de reactividad*, el de *grado de introspección* y el de *foco de atención*.

4.5.1. Reactividad Variable: Grado de Reactividad

Según se ha visto en el capítulo 2, la reactividad es una característica esencial de un agente. Además, en ese mismo capítulo, en la sección 2.2.2, y según [Wooldridge 1995], una posible clasificación de las arquitecturas de agente las divide en reactivas, deliberativas e híbridas. La diferencia entre estos tres tipos de arquitectura no está en si una de ellas presenta un agente reactivo y las otras no, sino en cuan reactivos son los agentes de un tipo frente a los de otro.

Así, la reactividad estará presente en formas distintas en un agente deliberativo y en un agente reactivo. La reactividad de un agente reactivo es mayor, su respuesta frente a un estímulo es más rápida y más directa, mientras que el agente deliberativo elabora más su respuesta y, por lo tanto, necesita más tiempo para actuar. De esta manera, cada una de las diferentes arquitecturas de agente tiene un grado de reactividad fijo que puede funcionar bien en una situación inicial, pero que no es capaz de cambiar para adaptarse a otras situaciones. Así, se ve de forma intuitiva que la reactividad no es una

característica absoluta, sino que existe una diferencia *cuantitativa* en cuanto a la reactividad de las diferentes arquitecturas, con lo que la reactividad puede ser gradada.

De aquí se puede concluir que una arquitectura de agente que pueda cambiar su grado de reactividad será capaz de adaptarse de una forma más adecuada que como lo hacen las arquitecturas tradicionales, y funcionará de manera más parecida a un humano. Por ejemplo, cuando una persona ve un papel quemándose que no quiere perder, puede dedicar un momento a pensar cómo evitar que sea consumido por las llamas; mientras que si lo que se quema es su mano, la apartará del fuego inmediatamente, sin dedicarle ni un mero momento a pensar qué hacer. Este ejemplo tan simple presenta a un humano actuando en base a dos grados de reactividad diferentes de acuerdo a la situación en la que se encuentra. Esta idea tan básica es el punto de partida de la aportación que se recoge en esta sección: la gestión de una reactividad variable.

El *Grado de Reactividad* de un agente es un valor real entre 0 y 1 que indica cuánto esfuerzo va a dedicar dicho agente a deliberar. Este grado define dos situaciones extremas con infinitos estados intermedios. Estas situaciones extremas son:

- Si el grado de reactividad es cero, el agente trabaja en **Modo Reflejo**: No dedica tiempo a deliberar, a mejorar la primera solución que obtiene. Es el grado de reactividad de los agentes con arquitectura reactiva.
- Si el grado de reactividad es 1, el agente trabaja en **Modo Deliberativo**: Dedica todo el tiempo que puede para calcular la mejor solución a su problema. Es el grado de reactividad de los agentes con arquitectura deliberativa.

La capacidad de adaptar la reactividad de un agente es incluso más importante si dicho agente trabaja en un entorno con restricciones temporales críticas, donde no se tendrá un agente que pueda ser totalmente deliberativo, que trabaje en **Modo Deliberativo**, por tener que satisfacer dichas restricciones. En este caso, el proceso deliberativo no puede demorarse todo el tiempo que necesite, pues hay unas restricciones temporales que debe cumplir, y por tanto actuar antes del cumplimiento de los plazos máximos indicados por dichas restricciones. En este tipo de agentes, para un valor 1 del grado de reactividad, el agente trabajará en **Modo Deliberativo en Tiempo Real** $\mathcal{DTR}$ en

lugar de en **Modo Deliberativo**, dedicando todo su tiempo **ocioso** al proceso deliberativo. En este caso, se define el grado de reactividad con los extremos indicados en la figura 4.1.

Del **Modo Reflejo** al **Modo** $\mathcal{DTR}$

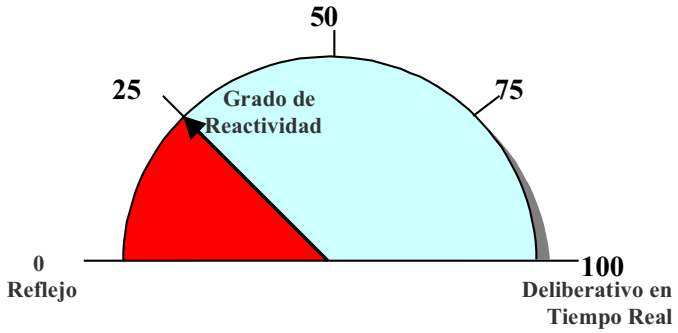

Figura 4.1: Grado de Reactividad de un $\mathcal{AA}$

4.5.2. Introspección variable: Grado de Introspección

Desde el punto de vista de la psicología general, se define *extrospección* [Larousse 2003] como *"el método de observación psicológica que no se basa en el análisis de los contenidos de conciencia sino en los que se derivan de los sentidos"*. Este concepto se contrapone al de *introspección* [Larousse 2003] definido en este caso como *"el método de autoobservación que se realiza de forma controlada, bajo condiciones experimentales. El objeto de conocimiento es el contenido mental de la propia persona que realiza la observación"*.

De forma similar, a la hora de estudiar el proceso deliberativo de un agente se puede distinguir entre la deliberación motivada como respuesta a cambios en el entorno y la deliberación sobre los objetivos propios del agente (en base a la pro-actividad del mismo).

De esta manera se define el *Grado de Introspección* como un valor entre 0 y 1 que indica la división en el tiempo de proceso que se realiza para indicar el tiempo que dedica el agente para atender a cambios en el entorno frente al tiempo que dedica a deliberar, a la introspección.

Conforme el *grado de introspección* tiene un valor más cercano a 0, el agente es más sensible a los cambios en el entorno, mientras que si el grado de

reactividad está más cercano a 1, está más centrado en sí mismo, es más *introspectivo*.

De forma similar a lo que ocurre con el grado de atención, el *grado de introspección* define dos situaciones extremas en el proceso de deliberación de un agente con infinitas situaciones intermedias. Estas situaciones extremas son:

- Si el *grado de introspección* es 0, la deliberación del agente es *extrovertida*, todo el proceso de deliberación está motivado por cambios en el entorno del agente.

- Si el *grado de introspección* es 1, la deliberación del agente es *introvertida*, el agente no está interesado en los cambios del entorno, y todo el proceso de deliberación es motivado por sus propios intereses u objetivos.

4.5.3. Concentración variable: Foco de Atención

Se define el concepto de *concentración* [Larousse 2003] desde el punto de vista de la psicología general como *"la capacidad de centrarse en unos estímulos determinados, desechando aquellos que no están relacionados con ellos, con el fin de focalizar la atención y optimizar la reflexión. Cuando se focaliza la atención, el campo de conciencia sufre un estrechamiento"*. De igual manera, la psicología general considera el concepto de *atención* [Larousse 2003] como *"la concentración selectiva de la actividad mental que implica un aumento de eficiencia sobre una tarea determinada, además de una inhibición perceptiva y cognitiva de las demás actividades. La atención no es una conducta por sí misma, sino que cualifica momentos determinados de una conducta total, en los cuales predomina la persistencia del contacto del sujeto con el objeto. Dicho objeto puede ser externo, y en tal caso se habla de atención externa, o interno, centrada en un estado del propio sujeto (introspección)"*.

El objetivo de la presente capacidad es el de aplicar los conceptos anteriores a la arquitectura de agente. Así, una forma de aumentar la eficiencia en el uso del tiempo de procesamiento en un agente consiste en poder *focalizar* dicho proceso, es decir, reducir el ámbito sobre el cual va a versar su proceso de razonamiento, para así emplear el tiempo de procesamiento disponible en aquello que sea de relevancia a la situación actual (variar la *concentración* del mismo).

De esta manera, se define un *foco de atención* como un subconjunto de creencias del agente que resultan significativos en alguna situación concreta. Tal y como presenta el concepto de *concentración*, estos subconjuntos de creencias se utilizarán para optimizar el proceso de razonamiento (y meta-razonamiento) priorizando el razonamiento acerca del foco o focos de atención activos.

El mecanismo de focalización no tiene porqué centrarse a un único foco, sino que pueden existir diversos focos de atención simultáneos, cada uno de ellos con un *grado de atención* distinto (de acuerdo a la importancia relativa de ese foco con relación a los demás focos activos).

4.6. Detección de Situaciones Significativas

Una de las capacidades básicas que se le incorpora por medio del presente trabajo al $\mathcal{AA}$ es la de detección de situaciones significativas (de cambio). Cabe destacar la importancia de esta capacidad, pues si el agente no es capaz de detectar una situación significativa de cambio, no podrá realizar las acciones oportunas que le permitan adaptarse a dicho cambio, es decir, sin esta capacidad, el resto de capacidades no se podrían aplicar.

Una situación va a venir definida por un estado del entorno junto con un estado interno de las creencias del propio agente.

De esta manera, una situación significativa de cambio será aquella frente a la cual el agente no podrá responder de la misma forma que lo estaba haciendo hasta ahora. Es decir, es una situación que obliga al agente a adaptarse, a cambiar, para poder enfrentarse a ella.

Con el objetivo de permitirle al diseñador de un $\mathcal{AA}$ especificar el proceso de *meta-razonamiento* que se encarga de implementar dicha adaptación se ha desarrollado un *Lenguaje de Control* que supone una de las principales aportaciones del presente trabajo al modelo de usuario del $\mathcal{AA}$.

Mediante este lenguaje de control, el diseñador de un $\mathcal{AA}$ puede especificar las diferentes situaciones significativas de cambio que podrá encontrarse el agente, y cuál es la forma de adaptarse a dicha situación, donde estas adaptaciones supondrán la utilización de alguna de las capacidades adicionales que se han incorporado al $\mathcal{AA}$ y que se han explicado en las secciones anteriores.

Para ello, se va a utilizar un mecanismo basado en eventos (que se detalla en el capítulo A), de tal forma que se le va a permitir al usuario especificar mediante este lenguaje la respuesta del Módulo de Control frente a los eventos que se hayan determinado de interés para el sistema. Estos pares *estímulo - respuesta* se implementan mediante reglas.

Así, este lenguaje de control permite especificar un conjunto de reglas de meta-razonamiento o *meta-reglas*, que se activan como respuesta a un evento significativo. Dicho evento, junto con la condición adicional que se puede asociar a cada meta-regla, permiten identificar la situación significativa de cambio, y por tanto cuándo se deben aplicar las acciones asociadas a la meta-regla.

Dos tipos: Reglas Críticas y Opcionales

Aunque la visión que se le va a dar al usuario es la de un lenguaje uniforme, hay que tener en cuenta que habrá dos tipos distintos de reglas, las *reglas críticas* (las que estén relacionadas con modificaciones de la parte con restricciones críticas del agente y que deberá gestionar el $\mathcal{RS}$) y las *reglas opcionales* (las que estén relacionadas con la gestión de la parte acrítica del agente, y que gestiona el $\mathcal{DS}$). Esta gestión de eventos se usará en concreto para:

- *Adaptar el $\mathcal{AA}$ a los cambios significativos de su entorno:* Esta adaptación en su grado más extremo supone un cambio de comportamiento activo del $\mathcal{AA}$. Esta modificación de comportamientos del $\mathcal{AA}$ la va a gestionar en su mayor parte el $\mathcal{RS}$, como ya se ha comentado al hablar de los comportamientos reflejos en la sección 4.4.
- *Modificar el $\mathcal{DS}$ para que sea adaptativo con respecto a las condiciones de carga:* Esto se logra cambiando el algoritmo de planificación de segundo nivel, y/o modificando el tiempo disponible entre las distintas fases del ciclo de control (*grado de introspección*).
- *Cambiar el foco de atención:* Esto supone que, sin cambiar por completo el comportamiento del agente, se varía la atención del mismo, haciendo que pase a deliberar sobre otro tema. También se podrá añadir otro foco de atención o cambiar el grado de atención de un foco.
- *Activar* in-agents *esporádicos:* Estos *in-agents* podrán ser críticos o acríticos (según posean restricciones temporales críticas o no). Aunque desde el punto de vista del modelo de usuario la acción a realizar es la misma, desde el punto de vista del modelo de sistema no se realizará lo mismo en los dos casos:

Activación de un *in-agent* esporádico acrítico: El $\mathcal{DS}$ lo activa y lo inserta en la *triggering subagenda*.

Activación de un *in-agent* esporádico crítico: El $\mathcal{RS}$ debe gestionar su activación. Para ello, se podría aplicar una de estas dos posibilidades:

1. Gestionarlo como un cambio de modo (según se presentó en 2.4.2), donde el nuevo modo contiene los *in-agents* / tareas que existían antes más el nuevo. Todas estas tareas esporádicas críticas se conocen en la fase de diseño, y además se puede imponer la restricción de que sólo puede haber una activa en cada momento, con lo que se pueden analizar en fase de diseño la planificabilidad de los distintos modos que salgan por la ejecución de las distintas tareas esporádicas que existan en el sistema.
2. El otro mecanismo que permitiría la ejecución de tareas esporádicas es el de ejecución de tareas con restricciones firmes [Terrasa 2000]. Cuando el $\mathcal{RS}$ decide que se debe activar esta tarea, se realiza un test de garantía dinámico para comprobar si se va a poder ejecutar de acuerdo al nivel de estrés del sistema. Pero a diferencia de las tareas críticas normales, el $\mathcal{RS}$ (si la condición de carga no lo permite) puede decidir no ejecutar la tarea.

La opción elegida en este caso es la primera, pues permite controlar en la fase de diseño del $\mathcal{AA}$ la sobrecarga que genera.

- *Modificar el grado de reactividad del $\mathcal{AA}$:* El *grado de reactividad* de un $\mathcal{AA}$ se implementa como el porcentaje máximo del *slack* disponible que se le deja al $\mathcal{DS}$ para la ejecución de la parte deliberativa del $\mathcal{AA}$. La variación de este porcentaje para que sea en cada momento el adecuado a la situación y necesidades del agente se realizará por medio de Meta-Reglas.

- *Desactivar niveles opcionales que habían sido interrumpidos previamente:* En el momento en el que el $\mathcal{SLS}$ lanza a ejecución un nivel opcional (en la fase *Interpret* de dicho módulo del $\mathcal{DS}$), lo hace dándole un intervalo máximo de tiempo para que acabe su ejecución (hay que volver a insistir en que se está trabajando con tiempo limitado). Si este nivel acaba su ejecución antes del intervalo especificado, todo sigue su ejecución de

forma normal, pero si el nivel no acaba su ejecución, entonces al llegar al límite del intervalo especificado se interrumpe la ejecución de este nivel, y se vuelve a reinsertar en las agendas para permitir la posterior continuación de su ejecución. De esta manera, si, mientras un nivel opcional se encuentra interrumpido, se ha modificado un *slot* que necesita para su cálculo, se puede descartar para una futura ejecución.

4.7. Aprendizaje

La capacidad de aprendizaje es considerada habitualmente como necesaria para el meta-razonamiento, pero debido a la magnitud del trabajo a desarrollar se dejó fuera del ámbito de este trabajo.

De todas maneras, aunque fuera del ámbito de la tesis, en los puntos en los que se ha detectado la posibilidad, se ha dejado preparada la arquitectura para su fácil incorporación.

Después de un análisis preliminar, los lugares donde se podría introducir la capacidad de aprendizaje serían los siguientes:

- Ajuste de los tiempos de ejecución en el peor caso (*wcet –worst case execution time–*) tanto de las diferentes KSs como de las diferentes fases del $\mathcal{DS}$: La modificación en tiempo de ejecución de estos tiempos ha sido considerada, dejando la arquitectura preparada para ello.
- Deducción de nuevas meta-reglas en tiempo de ejecución: Esta aplicación de los algoritmos de aprendizaje debe tener como base que las reglas que incorporan el *meta-conocimiento* se codifiquen dentro de estructuras de datos (en lugar de ser compiladas) en listas, por ejemplo, (con campos para la parte izquierda y para la parte derecha, codificando de alguna manera las operaciones lógicas a realizar en la parte izquierda). Así, con esta capacidad de aprendizaje, el sistema ganaría mucho en adaptatividad, pudiendo modificar las propias reglas conforme fuese aprendiendo, e incluso aprender nuevas reglas. Como se comentará posteriormente en el capítulo 8, ésta es la aproximación utilizada, y precisamente por la razón mencionada.
- Incorporación de los anteriormente mencionados *comportamientos reflejos adquiridos*: Tal y como ocurre en los dos casos anteriores, la arquitec-

tura está preparada para poder incorporar este tipo de comportamientos.

4.8. Trabajo a Realizar

Para proporcionar las funcionalidades que se plantean como objetivo del presente trabajo, se hace necesario extender tanto la visión como las capacidades que aporta la arquitectura en los diferentes modelos o visiones de la misma.

De forma más concreta, las contribuciones realizadas suponen:

En el Modelo Formal: Extender la definición formal de $\mathcal{AA}$, incorporando el concepto de *comportamiento*, así como el de *meta-razonamiento*. Esta extensión se detalla en el capítulo 5.

En el Modelo de Usuario: Extender la visión que tiene el usuario de la arquitectura, para lo que se debe:

- Ampliar el conocimiento del dominio, para permitir identificar los indicios de que ocurre una situación significativa de cambio.
- Incorporar en el conocimiento de resolución de problemas el concepto de comportamiento, lo que modificará no tan sólo la jerarquía de entidades sino la forma en la que un diseñador se debe plantear el desarrollo de un $\mathcal{AA}$. Adicionalmente, en este tipo de conocimiento se debe extender el concepto de *in-agent*para permitir la especificación de *in-agentsAperiódicos*.
- Incluir meta-conocimiento, lo que supone permitir tener tanto datos como un conocimiento de resolución de problemas propios de este conocimiento.

Estas extensiones al modelo de usuario se detallan en el capítulo 6.

En el Modelo de Sistema: Extender tanto el modelo de tareas como el Módulo de Control para dar soporte al nuevo modelo de usuario, así como para obtener un objetivo implícito: una utilización flexible del tiempo de procesamiento por parte del mencionado Módulo de Control. Estas extensiones se describen en el capítulo 7.

Tal y como se ha ido describiendo a lo largo de este capítulo, en los siguientes se detallan las extensiones que se han realizado en los diferentes modelos o niveles de abstracción.

5

Extensión al Modelo Formal de $\mathcal{AA}$

"Definir es saber. Por eso la definición justa es el más raro de los géneros."
"Critique sociale", I, Capital et travail
– Auguste Blanqui, (1805-1881)

5.1. Introducción

La inclusión de la abstracción de *comportamiento* dentro de la jerarquía de entidades de un $\mathcal{AA}$, junto con el *meta-razonamiento* obliga a un cambio en el planteamiento del modelo formal de un $\mathcal{AA}$.

En este capítulo se detalla la nueva aproximación al modelo formal de un $\mathcal{AA}$ que se produce por estas incorporaciones a dicha definición.

5.2. Definición de $\mathcal{AA}$

De esta manera, un agente ARTIS es una estructura de la forma:

$$\mathcal{AA} = \langle \mathcal{B}eh, G, B, \mu\mathcal{C}ontrol \rangle$$

donde:

- $\mathcal{B}eh = \{\mathcal{B}eh_1, \mathcal{B}eh_2, ..., \mathcal{B}eh_n\}$, es un conjunto de diferentes comportamientos del $\mathcal{AA}$ para diferentes situaciones complejas del entorno.
- $G = \{g_1^{[t_1,t_2]}, g_2^{[t_2,t_3]}, \ldots, g_n^{[t_n,t_m]}\}$, un conjunto de objetivos, los cuales expresan las motivaciones y las restricciones de tiempo real del agente. Los objetivos son representados como estados deseables y pueden estar acotados temporalmente. Estas restricciones temporales muestran el intervalo en el cual el objetivo debería ser obtenido. De esta manera, $g_1^{[t_1,t_2]}$ indica que se debería lograr el objetivo g_1 en el intervalo marcado por los instantes t_1 y t_2.
- B es el conjunto de creencias del $\mathcal{AA}$ en el instante actual. Todos los datos en este conjunto disponen de una etiqueta temporal mediante el empleo de una lógica temporal tal y como es expresado en [Crespo 1994, Barber 1994].
- $\mu\mathcal{C}ontrol$, es una tupla que se encarga de gestionar el meta-control del $\mathcal{AA}$, permitiendo adaptarse a cambios en el entorno, y tratando de mejorar la eficiencia de su proceso de razonamiento.

5.3. Definición de *Meta-Control*

De forma más concreta, se define el *Meta-Control* como

$$\mu\mathcal{C}ontrol = \langle \mathcal{B}_\mu, G_\mu, A_\mu, \mathcal{AFS}et \rangle$$

donde:

- $\mathcal{B}_\mu$, es el conjunto de creencias del $\mu\mathcal{C}ontrol$ (están almacenadas en un *Blackboard de Control*). Estas creencias son no observables, y reflejan conocimiento sobre el proceso de razonamiento, es por esto que se cumple que $\mathcal{B} \cap \mathcal{B}_\mu = \emptyset$.

- G_μ, objetivos propios del $\mu\mathcal{C}ontrol$. Estos objetivos están relacionados con la mejora de la eficiencia del proceso de razonamiento del $\mathcal{AA}$, así como la adaptatividad del agente a los cambios en el entorno. Estos objetivos están en un nivel de abstracción diferente a los objetivos del $\mathcal{AA}$, con lo que $G \cap G_\mu = \emptyset$.

- A_μ, conjunto de acciones que llevan a cabo el meta-razonamiento. Mediante estas acciones el *Meta-Control* cambia la forma en la que el $\mathcal{AA}$ trata de resolver el problema.

Foco de Atención se define por un conjunto de $\mu\mathcal{R}eglas$

- $\mathcal{AFS}et = \{\mathcal{AF}_i / \mathcal{AF}_i = \{\mu\mathcal{R}_j \in \mu\mathcal{R}uleSet\}\}$, conjunto de *Focos de Atención*, donde cada *Foco de Atención* se define a su vez por un conjunto de *Meta-Reglas*.

  Una *Meta-Regla* se define como $\mu\mathcal{R}_j : 2^G \times 2^{B \cup \mathcal{B}_\mu} \rightarrow \{A_\mu\}^+$, esto es una función que define la respuesta del *Meta-Control* frente a determinadas situaciones (definidas por los objetivos actuales del $\mathcal{AA}$ junto a un conjunto de creencias —del dominio y del *Meta-Control*—).

  Se distingue entre focos de atención activos y no activos. Los focos de atención activos determinan qué conocimiento es más importante en el instante actual pues es el que puede hacer variar la forma de actuar del agente.

5.4. Definición de Comportamiento

Un comportamiento $beh \subseteq \mathcal{B}eh$ se define como un conjunto no vacío de *in-agents*:

$$beh = \{a_{ij}\}$$

Los comportamientos contienen el conocimiento de resolución de problemas de un $\mathcal{AA}$. En un instante dado un $\mathcal{AA}$ sólo tiene activo un comportamiento, el cual viene definido por la situación en la que se encuentre el agente. Estos comportamientos representan la forma de actuar de un $\mathcal{AA}$ en situaciones complejas, y es por ello que son modelados mediante un conjunto de *in-agents*. Un *in-agent* es una entidad que proporciona una solución a un subproblema en un comportamiento concreto.

La gestión de situaciones es una de las funciones principales del meta-razonamiento

Así, se puede decir que un *estado* (interno junto con una representación del entorno) define una *situación* (representada por los objetivos y creencias actuales) que activa o permite que siga activo un *comportamiento* que determina el conjunto de *objetivos y restricciones* actuales para el agente, así como el *conocimiento* necesario para manejar la situación. La gestión de situaciones (por medio de la activación de comportamientos) es una de las funciones principales del meta-razonamiento del $\mathcal{AA}$. Esto concuerda con la definición de *Meta-Regla* ya que frente a diferentes situaciones las $\mu\mathcal{R}eglas$irán cambiando la forma en la que el $\mathcal{AA}$ se enfrenta a dicha situación, siendo el cambio más extremo el de comportamiento.

De esta manera, se puede ver un comportamiento como el medio por el cual el $\mathcal{AA}$ trata de conseguir un objetivo, y las condiciones de cambio de comportamiento como condiciones de cambio de objetivos. Formalmente:

$\forall i : \pi_i^{[t,t']} \rightarrow [b_i] g_i^{[t,t']}$, donde:

- b_i: comportamiento.
- $\pi_i^{[t,t']}$: condición de cambio de comportamiento, o de activación del comportamiento b_i, que es cierta en el intervalo de tiempo [t, t'].
- $g_i^{[t,t']}$: objetivo a lograr con el comportamiento b_i en el intervalo de tiempo [t, t'].

Propiedad 1 *Una condición de activación de un comportamiento será cierta en el intervalo de tiempo en el que los objetivos a los que está asociada (los objetivos que debe cumplir el $\mathcal{AA}$ mediante dicho comportamiento) sean los objetivos actuales del agente:*

$$\forall i : \pi_i^{[t,t']} \leftrightarrow (G^{[t,t']} = g_i^{[t,t']})$$

Propiedad 2 (No pueden existir dos comportamientos iguales)

$$\forall i : \pi_i^{[t,t']} \leftrightarrow (\mathcal{G}^{[t,t']} = g_i^{[t,t']}) \wedge \neg\exists j / ((j \neq i) \wedge (\mathcal{G}^{[t,t']} = g_j^{[t,t']}))$$

Propiedad 3 (Comportamiento actual único) *No puede haber dos comportamientos activos de forma simultánea.*

$$\forall i, j : i \neq j \leftrightarrow \pi_i^{[t,t']} \neq \pi_j^{[t,t']}$$

Definición 4 (Transición entre comportamientos) *Se dice que hay una transición del comportamiento b_i al b_j, y se representa $b_i \overset{[t_1,t_2]}{\rightsquigarrow} b_j$, cuando la condición de activación del comportamiento b_j se hace cierta justo en el instante en el que se hace falsa la condición de activación de b_i:*

$$\forall i,j:\ i \neq j | b_i \overset{[t_1,t_2]}{\rightsquigarrow} b_j \leftrightarrow \pi_i \,\mathcal{U}\, \pi_j$$

Un detalle a tener en cuenta de la definición anterior es el uso del operador temporal $\mathcal{U}$ propio de la lógica modal *RTAL* [Rebollo 2004] (extensión de la lógica RTCL para trabajar con las creencias de los agentes), que se define de la siguiente manera:

$$x \vDash p\,\mathcal{U}\,q \Leftrightarrow \exists i[x^i \vDash q \wedge (j < i \rightarrow x^j \vDash p)]$$

Otra forma de controlar el instante *t* en el que se debe producir una transición entre comportamientos es controlar cuando se alcanzan o dejan de ser válidos los objetivos de un comportamiento b_i dado y pasan a estar activos los objetivos asociados a otro comportamiento b_j:

$$\forall i,j:\ i \neq j / \ (b_i \overset{[t_1,t_2]}{\rightsquigarrow} b_j) \leftrightarrow (G = g_i^{[t_1,t-1]}) \wedge (G = g_j^{[t,t_2]}) / \ t_2 > t_1$$

La transición entre comportamientos define un *Autómata Finito No Determinista*, en el que un estado del autómata es un comportamiento, y desde un estado se puede ir a varios estados (de ahí el no determinismo).

Definición 5 (k-transición entre comportamientos) *Se dice que el comportamiento b_j se alcanza en k transiciones desde b_i, y se representa $b_i \overset{k}{\rightsquigarrow} b_j$, cuando existe al menos una secuencia de k transiciones que permiten alcanzar b_j a partir de b_i de la siguiente forma:*

Una ejecución de un $\mathcal{AA}$ puede verse como una secuencia de comportamientos

$$b_i \rightsquigarrow b_{i+1} \rightsquigarrow \cdots \rightsquigarrow b_{i+k} \rightsquigarrow b_j$$

Una ejecución de un $\mathcal{AA}$ puede verse como una secuencia de comportamientos.

Propiedad 6 *Tiene que existir una secuencia de transiciones entre comportamientos desde el comportamiento inicial hasta cada uno de los comportamientos existentes.*

$$\forall beh_i \in Beh, \exists k \in \mathbb{N} : \ beh_0 \overset{k}{\rightsquigarrow} beh_i$$

,donde beh_0 es el comportamiento inicial del AA.

El cumplimiento de esta propiedad asegura que todo comportamiento tiene que estar activo alguna vez.

El cumplimiento de los objetivos de diseño de un agente ARTIS se puede garantizar a partir del modelo de agente.

5.5. Definición de *in-agent*

Los *in-agents*contienen el conocimiento de resolución de problemas teniendo en cuenta las restricciones temporales que impone el entorno. Se definen de la siguiente forma:

$$a_i = \langle \Sigma_i, \mathcal{A}_i, B_i, r_i, d_i, D_i, T_i \rangle$$

donde:

- $\Sigma_i \subseteq \Sigma$ es el conjunto de percepciones que realiza el *in-agent* directamente sobre el entorno y que son empleadas para poder actualizar sus creencias. Para ello se define Σ como el conjunto de todas las posibles percepciones del entorno que puede tener el $\mathcal{AA}$.
- $\mathcal{A}_i$ es el conjunto de acciones[1] que el *in-agent* conoce. Dado que se define Act como el conjunto de todas las acciones que conoce el $\mathcal{AA}$, se cumple que $\mathcal{A}_i \subseteq Act$.

[1] Las acciones que puede realizar un *in-agent* son de tres tipos:

- cognitivas: modifican el estado interno;
- efectoras: actúan sobre el entorno, y
- comunicativas: responsables del envío y recepción de mensajes con otros agentes.

- $B_i \subseteq Belset$ es un conjunto de creencias que representan el estado del entorno así como el estado interno del *in-agent*. Estas creencias son parte del conjunto de creencias ($Belset$) del $\mathcal{AA}$ al que el *in-agent* pertenece.
- r_i es la función que contiene el conocimiento reflejo del *in-agent*.
- d_i es la función que contiene el conocimiento deliberativo del *in-agent*.
- $D_i \in \mathbb{R}^+$ es el *deadline* del *in-agent*, el cual indica el plazo máximo de ejecución en el cual el *in-agent* debe haber ejecutado alguna acción.
- $T_i \in \mathbb{R}^+$ es el periodo del *in-agent*, el cual determina la frecuencia de activación del *in-agent*. Este periodo es necesario debido a la propia dinámica del entorno en sistemas de tiempo real. A cada periodo, la ejecución del *in-agent* comienza con la lectura de los nuevos valores de los datos de entrada (percepciones) por medio de los sensores apropiados. Estos nuevos valores actualizarán el conjunto B_i del *in-agent*.

Se define $InAg = \{a_i | i \in \mathbb{N}\}$ como el conjunto de todos los *in-agents*de un agente ARTIS.

Como ya se ha comentado, los *in-agents*se agrupan en comportamientos que responden a escenarios concretos en los que se puede encontrar el agente. Cada comportamiento contiene los *in-agents*que se encuentran activos, permaneciendo el resto inactivos.

Dos comportamientos pueden ser diferentes porque el periodo o el *deadline* de alguno de los *in-agents*que lo forman varíe.

Definición 7 *Dos comportamientos beh y beh' se considera que son* diferentes *cuando se cumple:*

$$\exists a_i \in InAg | a_i \in beh \wedge a_i \notin beh'$$

No puede haber ningún *in-agent*que no pertenezca a un comportamiento

Propiedad 8 *No puede haber un* in-agent *que no pertenezca a ningún comportamiento. Es decir, el conjunto de comportamientos Beh que forman el $\mathcal{AA}$ debe formar un recubrimiento sobre el conjunto de* in-agents

$$\forall a_i \in InAg, \exists beh_j \in Beh / a_i \in beh_j$$

Definición 9 (Razonamiento reactivo de un $\mathcal{AA}$) *Dado beh_{act} comportamiento actual del $\mathcal{AA}$, el razonamiento reactivo de dicho $\mathcal{AA}$ viene dado por*

$$\bigotimes_{\forall_{i=1..n} a_i \in beh_{act}} r_i = r_1 \circ r_2 \circ \ldots \circ r_n$$

Definición 10 (Razonamiento deliberativo de un $\mathcal{AA}$) *Dado beh_{act} comportamiento actual del $\mathcal{AA}$, el razonamiento deliberativo de dicho $\mathcal{AA}$ viene dado por*

$$\bigotimes_{\forall_{i=1..n} a_i \in beh_{act}} d_i = d_1 \circ d_2 \circ \ldots \circ d_n$$

Las acciones de una $\mu\mathcal{R}egla$ pretenden modificar el proceso de razonamiento de un $\mathcal{AA}$. Esto supone la modificación del razonamiento reactivo y/o deliberativo de dicho agente. Estas modificaciones pueden suponer desde simples cambios en el orden de secuencia de ejecución (el orden en el que se componen, por tanto, las r_i y/o d_i), hasta el caso más extremo de cambio de comportamiento (que supone el cambio de las funciones que componen los razonamientos reactivo y deliberativo del $\mathcal{AA}$).

5.6. Conclusiones

En este capítulo se ha descrito las extensiones realizadas a la visión formal del $\mathcal{AA}$ para incorporar el concepto de comportamiento como base del conocimiento de resolución de problemas en la composición de un $\mathcal{AA}$. También se ha añadido el Meta-Control dentro de la definición de $\mathcal{AA}$.

Por tanto, en esta visión formal de la arquitectura de $\mathcal{AA}$ se puede observar cómo las extensiones comentadas suponen un cambio drástico en la visión del agente, así como lo van a suponer en la forma de diseñarlo (mediante las extensiones al modelo de usuario) y en la implementación interna del mismo (en este caso por medio de las extensiones al modelo de sistema).

6 Extensión al Modelo de Usuario del $\mathcal{AA}$

"To see the World in a Grain of Sand,
And a Heaven in a Wild Flower,
Hold Infinity in the palm of your hand,
And Eternity in an hour."
"Auguries of Innocence", I
– William Blake, (1757-1827)

6.1. Introducción

Tal y como se comentó en el capítulo 4, el trabajo aquí presentado amplía la visión que tiene el usuario de la arquitectura, para lo que se han extendido los dos tipos de conocimiento que puede definir el usuario.

Además, se ha añadido la posibilidad de incorporar y gestionar un nuevo tipo de conocimiento, meta-conocimiento, por medio de un conjunto de creencias propio de este tipo de conocimiento y de un *lenguaje de control*.

El resto del capítulo se estructura, por lo tanto, en tres secciones, la pri-

mera de las cuales se centra en la extensión del conocimiento de dominio, la segunda expone la extensión del conocimiento de resolución de problemas, y por último se presenta la visión de usuario del meta-conocimiento que se ha incorporado al $\mathcal{AA}$.

6.2. Extensión del Conocimiento del Dominio

El conocimiento del dominio o conocimiento factual está compuesto por la información del entorno que posee el agente. En el capítulo 3 se presentó la representación utilizada en el modelo de usuario de un $\mathcal{AA}$ para dicho conocimiento: una taxonomía de *frames* compuesta por un conjunto de clases con *slots* estáticos o temporales.

La extensión del conocimiento del dominio tiene como objetivo el permitir al diseñador de un $\mathcal{AA}$ identificar los indicios de situaciones significativas para así poder posteriormente establecer la forma de comprobar si de verdad se ha llegado a una situación significativa, y qué es lo que hay que hacer en dicho caso. La forma de identificar estos indicios es por medio de la definición de *eventos*. Un *evento* es un mensaje que tiene como objetivo el comunicarle al Módulo de Control algún suceso que ha acontecido.

6.2.1. Tipos de Eventos

Dentro de la arquitectura de $\mathcal{AA}$ existen dos clases principales de eventos:

1. Eventos asociados a las partes opcionales de los *in-agents* críticos (evento del tipo `CONSIDER_PERIODIC_AGENT`): Estos eventos van a generarse desde el $\mathcal{RS}$ para comunicarse con el $\mathcal{DS}$ en dos posibles ocasiones:

   a) Cuando se complete la ejecución de la parte inicial de un *in-agent* crítico: Para indicarle al $\mathcal{DS}$ que debe tener en cuenta la parte opcional de dicho *in-agent* para su posible ejecución, es decir, para que el Planificador de Segundo Nivel ($\mathcal{SLS}$) lo tenga el cuenta a la hora de planificar.

   b) Cuando se complete la ejecución de la parte final de un *in-agent* crítico: Para indicarle al $\mathcal{DS}$, en esta ocasión, que ya no debe tener en cuenta la parte opcional de dicho *in-agent*.

Debido a que, tal y como se acaba de ver, los eventos de este tipo se hacen necesarios para el correcto funcionamiento del $\mathcal{DS}$, se generan todos los eventos posibles de este tipo, no teniendo que especificar nada el usuario relativo a dicha generación.

2. Eventos asociados a cambios en el $\mathcal{KDM}$. Dentro de este último tipo de eventos se pueden a su vez distinguir dos subclases de eventos:

   Relacionados con cambios en valores actuales: Dentro de este tipo se distingue entre `MODIFICATION` para indicar que se desea un evento frente a cualquier modificación del valor actual del *slot*, y `DELETION` para indicar en este caso la generación de un evento cuando se borre el valor actual del *slot*.

   Relacionados con cambios en las predicciones: Estos eventos se asociarán a *slots* temporales, en concreto a los valores futuros (o predicciones) de dichos *slots*. Los diferentes eventos de este tipo son: `NEW_PREDICTION` (se generaría al asociar una nueva predicción al *slot*), `MATCHED_PREDICTION` (se generaría cuando se comprobase que una predicción se ha cumplido, y pasa a ser valor actual del *slot*) y `UNMATCHED_PREDICTION` (se generaría al comprobarse que una predicción no se ha cumplido).

   En el caso de los eventos de este tipo, del $\mathcal{KDM}$, se generarán bajo demanda, con lo que el diseñador del $\mathcal{AA}$ deberá especificar, por medio de la extensión realizada para ello al lenguaje de clases, los que quiera que se generen.

6.3. Extensión del Conocimiento de Resolución de Problemas

El conocimiento de resolución de problemas es el que engloba los distintos métodos para resolver problemas. Tal y como se vio en el capítulo 3 este conocimiento se implementa en la arquitectura de agente ARTIS por medio de una jerarquía de entidades. La extensión de este conocimiento realizada supone en primer lugar modificar la definición del concepto de *in-agent* para eliminar la obligatoriedad de incluir un período en la definición del mismo (lo que posibilita la definición de *in-agents* esporádicos o aperiódicos). En segundo lugar, la incorporación del concepto de *comportamiento* a esta jerarquía

de entidades del $\mathcal{AA}$, para lo cual se define el concepto de *comportamiento* y se debe incluir en el lenguaje de entidades.

Desde el punto de vista del modelo de usuario, un comportamiento va a estar formado por un conjunto de *in-agents* junto con las condiciones en las que se va a activar.

De esta manera, se extiende el lenguaje de entidades para incorporar un nuevo nivel dentro de la jerarquía de entidades, que será el correspondiente a comportamientos, tal y como refleja la figura 6.1.

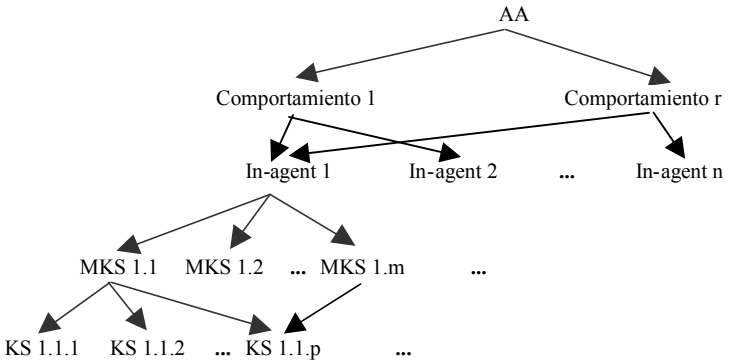

Figura 6.1: Jerarquía de entidades modificada

Cabe resaltar que por motivos de simplicidad (en una ya de por sí compleja jerarquía) se ha decidido presentar en este modelo el concepto de *comportamiento* como un todo, no distinguiendo entre comportamientos reflejos y deliberativos. Cada uno de los *comportamientos* definidos en este modelo corresponde a un comportamiento reflejo junto con el comportamiento deliberativo que lo refina. Los *in-agents* que compondrían los comportamientos deliberativos independientes de los comportamientos reflejos (*in-agents* aperiódicos acríticos) se activarán de forma individual mediante la acción correspondiente del lenguaje de control.

Esta incorporación a la jerarquía de entidades es algo más que una mera abstracción o entidad a rellenar por el diseñador, ya que supone un cambio drástico en la forma de diseñar un $\mathcal{AA}$.

Así, el usuario, cuando diseña un $\mathcal{AA}$, lo primero que deberá realizar es

estudiar las diferentes situaciones a las que se deberá enfrentar dicho agente, y diseñar los comportamientos adecuados para enfrentarse a ellas.

A partir de dichos comportamientos se construye, siempre en la fase de diseño del agente, un autómata finito no determinista en el que los estados son los distintos comportamientos del $\mathcal{AA}$, y los cambios entre estados vienen dados por las transiciones entre comportamientos (figura 6.2). El no-determinismo del autómata así construido viene dado porque no tiene por qué existir una secuencia de comportamientos única, sino que a partir de un comportamiento se puede pasar a varios diferentes dependiendo de la situación, ya que el agente se enfrentará a un entorno no determinista.

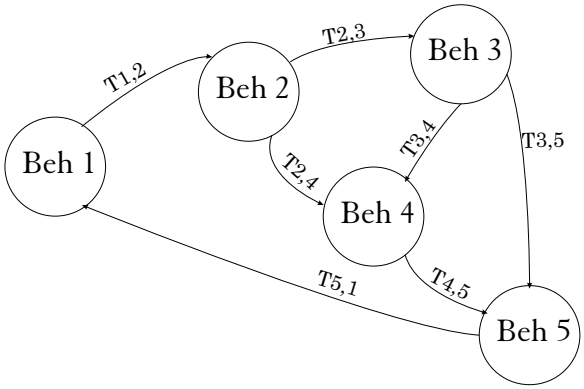

Figura 6.2: AFND de las transiciones de comportamientos de un $\mathcal{AA}$

Una vez se ha construido el autómata, se conocen todas las posibles transiciones entre comportamientos (además de tener en cuenta que no deben existir comportamientos que no sean alcanzables, ni ningún sumidero en el grafo, es decir, un comportamiento que una vez alcanzado no permite la transición a ningún otro comportamiento), y se realizará un análisis de la planificabilidad, no sólo de cada comportamiento por separado, sino también de las transiciones entre dichos comportamientos.

6.4. Meta-Conocimiento

El meta-conocimiento es conocimiento sobre el conocimiento. Uno de los objetivos del presente trabajo se traduce en la incorporación de este tipo de

conocimiento dentro de la arquitectura de agente ARTIS.

Al igual que ocurre con el conocimiento normal, el meta-conocimiento se puede clasificar en dos tipos:

- *Meta-conocimiento factual*: está compuesto por un conjunto de datos propios del meta-razonamiento, que se almacenan en un *blackboard de control* denominado *Meta-KDM* ($\mu\mathcal{KDM}$).

  Este $\mu\mathcal{KDM}$ se estructura de la misma manera que su contrapartida del conocimiento del dominio ($\mathcal{KDM}$).

- *Meta-conocimiento de resolución de problemas*: Este tipo de meta-conocimiento se especifica mediante un *lenguaje de control* que se ha desarrollado específicamente para dicho uso. Dicho meta-conocimiento se estructura en *focos de atención*.

  A la hora de planificar qué ejecutar, se ha dotado al *Servidor Deliberativo* de un mecanismo que le permite focalizar la atención dependiendo de ciertas condiciones (debidas o no a eventos), para priorizar a unos *in-agents* determinados, o a las importancias,... Un *foco de atención* estará formado por un conjunto de objetos del $\mathcal{KDM}$ (bien completos, bien tan sólo algunos *slots* determinados de dichos objetos). En un instante dado, pueden existir varios *focos de atención activos*, cada uno de ellos con un *grado de introspección* distinto.

  La definición de estos focos se realiza por medio del *lenguaje de control* como se verá al hablar de dicho lenguaje.

  Dentro del *lenguaje de control*, el usuario puede especificar reglas cuyas acciones varíen la forma en la que el $\mathcal{AA}$ va a realizar su proceso de razonamiento. Estas acciones permitirán desde cambiar el comportamiento del agente por otro completamente distinto hasta modificar los diversos parámetros que controlan el proceso de razonamiento del mismo (como el *grado de reactividad*, el *grado de introspección*, el *foco de atención*, el *grado de atención* de alguno de los focos, o la *política de planificación de segundo nivel*).

6.5. El Lenguaje de Control

El lenguaje de control permite al diseñador especificar qué eventos son de interés para el Módulo de Control y cuándo. Es decir, no siempre que se genere un evento éste va a ser de interés sino que deberán cumplirse ciertas condiciones a especificar por el diseñador. Este lenguaje proporciona un mecanismo de respuesta a dichos eventos de interés definiendo conjuntos de reglas por medio de la sintaxis comentada a continuación.

El lenguaje de control permite especificar qué eventos son de interés y cuándo

Así, un foco de atención vendrá determinado por los eventos de un subconjunto de *slots* del $\mathcal{KDM}$ a los que el proceso de meta-razonamiento del $\mathcal{AA}$ debe atender. Para atender dichos eventos habrá un conjunto de meta-reglas, una por cada evento. Así, un foco de atención de n *slots* tendrá n meta-reglas. Con el objetivo de simplificar todo el proceso, se define un foco de atención por el conjunto de meta-reglas que dan respuesta a los eventos de los *slots* que, en realidad, lo componen.

Un *Foco de Atención* se define de la siguiente manera:

```
( AttentionFocus <nombre>
          <lista_reglas>
        )
```

Además, este lenguaje permite utilizar los denominados *meta-slots* que forman el *Meta-conocimiento factual* comentado en la sección anterior. Este conjunto incluye los siguientes *meta-slots*:

SLS_Policy : Política de planificación utilizada por el $\mathcal{SLS}$ (Planificador de Segundo Nivel) de entre los definidos en el sistema (EDF, BIF, ELDF, HQF, HSF, MU, DBM, AEDF, AHUF, SSS).

ReactivityDegree : *Grado de Reactividad* del agente ARTIS ($\mathcal{AA}$). Indica el porcentaje del tiempo de holgura que se utiliza para la parte deliberativa del $\mathcal{AA}$.

AttentionDegree : *Grado de Introspección* del $\mathcal{DS}$. Permite especificar qué parte del tiempo de que dispone el $\mathcal{DS}$, va a utilizar en atender a los cambios del exterior (representados por medio de los eventos) en lugar de dedicarlo a la introspección, o deliberación.

EventRate : Tasa de eventos o número máximo de eventos que puede gestionar el $\mathcal{DS}$ por ciclo de control. Este dato se calcula a partir del grado de introspección del $\mathcal{DS}$ y del tiempo disponible para la ejecución del mismo.

Quality : Calidad actual de la ejecución de la capa deliberativa del $\mathcal{AA}$.

LevelsCompleted : Número de niveles acríticos ejecutados de forma completa hasta el momento.

LevelsInterrupted : Número de niveles acríticos cuya ejecución ha sido interrumpida hasta el momento.

6.5.1. Tipos de Reglas

Existen tres tipos distintos de reglas de acuerdo al tipo de evento que las dispara:

- *Reglas asociadas a eventos relacionados con cambios en el $\mathcal{KDM}$:* Además de indicar el evento, se puede especificar una condición adicional a cumplir para que se ejecuten las acciones especificadas. También se permite el priorizar unas reglas frente a otras por medio de la *importancia* de la regla.

```
( defMetaRule <nombre> <importancia>
    ( <Tipo_evento_slot> <clase>.<instancia>.<slot> )
    <condición>
    =>
    <acciones>
 )
```

- *Reglas asociadas a las partes opcionales de* in-agents *críticos:* El evento de disparo de estas reglas es generado por el $\mathcal{RS}$ cuando ha finalizado la ejecución de la parte crítica de un MKS crítico de un *in-agent* periódico, y que puede por tanto ejecutar, si lo considera oportuno, los niveles opcionales de dicha MKS, y las MKS opcionales que pueda tener dicho *in-agent*. Al igual que ocurre con el tipo de reglas anterior, se puede especificar, de forma opcional, una condición adicional a cumplir para

ejecutar las acciones asociadas a la regla.

```
( defMetaRule <nombre> <importancia>
            ( CONSIDER_PERIODIC_AGENT <In-Agent> )
            <condición>
            =>
            <acciones>
 )
```

- *Reglas sin Evento:* Estas reglas sin evento van a permitir especificar una serie de acciones a ejecutar siempre que entre en funcionamiento esta parte del $\mathcal{DS}$.

```
( defMetaRule <nombre> <importancia>
        =>
        <acciones>
 )
```

Las acciones a realizar como parte derecha de las reglas van a poder ser de dos tipos:

- *Modificación de valores de datos:* Estos valores pueden ser:
  - Del $\mathcal{KDM}$: Se puede modificar el valor de un *slot* del $\mathcal{KDM}$.
  - Del $\mu\mathcal{KDM}$: Para modificar estos *meta-slots* que contienen los datos internos propios del proceso de meta-razonamiento del Módulo de Control, se debe utilizar la función de interfaz adecuada. Además, la modificación de alguno de estos valores desencadena un cambio en la forma de actuar del Módulo de Control.
- *Activación de módulos a tener en cuenta por el planificador de segundo nivel.*

De esta manera, se dispone de los siguientes siete tipos distintos de acciones con la sintaxis que se indica a continuación:

- Modificar el valor de un *slot* del $\mathcal{KDM}$ o del $\mu\mathcal{KDM}$:

```
<clase>.<instancia>.<slot> := <expresión>
```

- Activar el *in-agent* aperiódico (o esporádico) acrítico *identificador* :

```
TriggerSporadicLevel ( <identificador> )
```

- Modificar el *Grado de Reactividad* del agente para que pase a valer *number*:

```
SetReactivityDegree ( <number> )
```

- Modificar la política de planificación del planificador de segundo nivel ($\mathcal{SLS}$) para que pase a ser *Policy_id* :

```
SetPolicy ( <Policy_id> )
```

- Activar/Desactivar el foco de atención *Focus_id*: Si el *Attention_Degree* que se establece es 0, se desactiva el foco de atención *Focus_id* (en el caso de que ya estuviese activo), si dicho valor es mayor que cero se activará este foco de atención con el grado de atención indicado en *Attention_Degree*:

```
SetFocus ( <Focus_id>, <Attention_Degree> )
```

- Modificar el grado de introspección del agente para que pase a ser *number* :

```
SetAttentionDegree ( <number> )
```

- Modificar el comportamiento actual del agente para que pase a ser *identifier*:

```
SetBehaviour ( <identifier> )
```

- Añadir un evento del tipo *Ev_Type* sobre el *slot* identificado por *cla-*

se.instancia.slot con la importancia indicada por *importance*:

```
KDM_Add_Event ( <clase>.<instancia>.<slot>,
                <Ev_Type>, <Importance> )
```

- Eliminar el evento de tipo *Ev_Type* del *slot* identificado por *clase.instancia.slot*:

```
KDM_Release_Event ( <clase>.<instancia>.<slot>, <Ev_Type> )
```

6.5.2. Ejemplo de uso de Reglas de Control

Con objeto de ilustrar la sintaxis presentada, a continuación se muestra el código en lenguaje de usuario de un sencillo ejemplo de un foco de atención (tanque), compuesto por una única regla (nivel) que modifica el comportamiento deliberativo del agente frente a un evento de modificación del valor del nivel. De esta manera, si el valor de dicha variable es mayor o igual que 200 y menor que 400, se realizan las siguientes acciones:

- Activar un *in-agent* esporádico acrítico (*SpNivel*) que será tenido en cuenta por el planificador de segundo nivel ($\mathcal{SLS}$) para su posible ejecución.

- Se cambia el grado de reactividad del agente a un 75 % del total posible, es decir, el $\mathcal{AA}$ tan sólo utilizará el 75 % del tiempo de *slack* u holgura disponible para la ejecución del $\mathcal{DS}$.

- Se modifica el grado de atención del $\mathcal{DS}$, es decir, el porcentaje del tiempo que dedica el $\mathcal{DS}$ a responder a eventos para que pase a ser un 25 % del total de que dispone.

```
(AttentionFocus tanque
    (defMetaRule nivel 3
        ( MODIFICATION tanque.tanqueA.nivel)
        ( tanque.tanqueA.nivel >= 200 )
        ( tanque.tanqueA.nivel < 400 )
        =>
        ( TriggerSporadicLevel ( SpNivel ) )
        ( SetReactivityDegree ( 0.75 ) )
        ( SetAttentionDegree ( 0.25 ) )
    )
)
```

En cuanto a la estructura y representación interna de las Meta-Reglas, se tratará en el capítulo posterior correspondiente.

6.6. Conclusiones

En este capítulo se han presentado las extensiones al modelo de usuario desarrolladas. Estas extensiones suponen por una parte la incorporación del concepto de comportamiento como base de la jerarquía de entidades de un $\mathcal{AA}$, y la incorporación del lenguaje de control como interfaz de usuario para la especificación del proceso de meta-razonamiento.

Estas fácilmente resumibles aportaciones al modelo de usuario suponen un cambio drástico en la forma en la que un $\mathcal{AA}$ debe ser diseñado.

7

Extensión al Modelo de Sistema del $\mathcal{AA}$

"El trabajo se alarga hasta llenar el tiempo disponible para llevarlo a cabo [...]
La tarea por hacer crece en importancia y complejidad
en relación directa con el tiempo que ha de emplearse."
"Las Leyes de Parkinson"
– Cyrill Northcote Parkinson, (1919)

7.1. Introducción

Tal y como se comentó en el capítulo 4, en el Modelo de Sistema del $\mathcal{AA}$, se ha extendido tanto el Módulo de Control como el modelo de tareas para incorporar tanto la utilización flexible del tiempo de holgura disponible (como única parte del tiempo de procesamiento cuya utilización puede ser modificada) como todo el soporte necesario al nuevo modelo de usuario visto en el capítulo 6).

De forma más concreta, para dar soporte en el modelo de sistema a los objetivos planteados en el capítulo 4 se ha realizado lo siguiente:

1. **Detección de situaciones significativas de cambio**: se ha diseñado un mecanismo de comunicación asíncrona que permite la generación de eventos significativos para detectar las situaciones significativas de cambio.

   Para gestionar estas situaciones se han añadido los módulos necesarios para que el Módulo de Control pueda gestionar *meta-reglas* (en concreto *focos de atención* definidos como conjuntos de *meta-reglas*) tal y como se definen en el modelo de usuario.

2. **Adaptar el proceso de razonamiento del agente**: por medio de la variación del *Grado de Reactividad*, *Grado de Introspección* y el *Foco de atención*, y siempre teniendo en cuenta las restricciones temporales del agente.

   Además, se da soporte desde el Módulo de Control a estos tres conceptos que fueron definidos en el modelo de usuario (capítulo 6), así como al resto de posibles acciones de las meta-reglas tales como la activación de la carga aperiódica acrítica, la modificación dinámica de la política de planificación de segundo nivel, ...

3. **Gestión de múltiples comportamientos**: Uno de los conceptos introducidos en el presente trabajo es el de la gestión de *comportamientos* que desde el punto de vista del Modelo de Sistema se va a distinguir entre *comportamientos deliberativos* y *comportamientos reflejos*. Al definir dicho modelo de gestión, se han tomado las siguientes decisiones:

   - Con respecto a los *Comportamientos Reflejos*:
     - Se ha extendido el modelo de tareas para incorporar la definición de modo de funcionamiento como traducción en este modelo del concepto de comportamiento presentado en el modelo de usuario.
     - Se han incorporado las extensiones necesarias en el $\mathcal{RS}$ que permiten cambiar el comportamiento reactivo del agente para adecuarse a las necesidades tanto internas (según su estado) como externas (producidas por eventos). Esto supone, entre otras cosas, la incorporación de un protocolo de cambio de modo que da soporte a la gestión de los cambios de comportamiento.

       En este caso cabe destacar que la aplicación de técnicas de los sistemas multi-modo a esta arquitectura tiene como objetivo

de nivel del sistema el minimizar el tiempo de cómputo dedicado a las tareas críticas, para así poder maximizar el tiempo dedicado a la mejora de la solución del agente $\mathcal{AA}$. En este sentido, cabe recordar que esta arquitectura trabaja a dos niveles, es una arquitectura con dos planificadores, utilizando el tiempo *sobrante* u holgura, después de asegurar la ejecución de todas las tareas con restricciones temporales críticas dentro de sus plazos, para tratar de mejorar la calidad de la respuesta del $\mathcal{AA}$. De esta manera, al utilizar las técnicas multi-modo, en un momento dado se ejecutarán tan sólo aquellas tareas críticas de un modo concreto, y no todas las tareas críticas del sistema, obteniendo así más tiempo de procesador para la mejora de la respuesta del agente.

  - Se ha dotado al Modelo de Sistema de la flexibilidad y estructuras necesarias para permitir *comportamientos reflejos adquiridos*, aunque el establecer un mecanismo de aprendizaje que permita la generación de dichos comportamientos queda fuera del contexto del presente trabajo.

- Con respecto a los *Comportamientos Deliberativos*: se ha extendido el $\mathcal{DS}$, incorporando las funcionalidades necesarias para gestionar la activación/desactivación de *in-agents* esporádicos acríticos.

Se han desarrollado, en las dos partes del Módulo de Control, nuevas funcionalidades y extendido algunas de las existentes:

- En el $\mathcal{DS}$ estas funcionalidades incluyen desde la gestión de la carga aperiódica acrítica, y la modificación dinámica de la política de planificación de segundo nivel, hasta la propia modificación del ciclo de control. Además, se ha tenido que dar soporte a la gestión del *Grado de Introspección* (según se definía en el capítulo 4).

- En el $\mathcal{RS}$ se ha flexibilizado el cálculo del tiempo de holgura disponible, para hacerlo variable en base a un *Grado de Reactividad*.

En base a todo esto, en el resto del capítulo se presentarán con mayor detalle estas extensiones al modelo de tareas y al Módulo de Control. Así, en primer lugar se relatan las extensiones al modelo de tareas, pasando a continuación a detallar los cambios producidos a nivel de $\mathcal{RS}$ y por último a nivel de $\mathcal{DS}$.

7.2. Extensión al Modelo de Tareas

El modelo de tareas de la arquitectura de $\mathcal{AA}$ se ha extendido para permitir la definición de distintos modos de funcionamiento, que es la traducción dentro del modelo de sistema del concepto de comportamiento que aparecía en el modelo de usuario.

Como se ha explicado en el capítulo 3, un *in-agent* se traduce en el modelo de sistema a una tarea de bajo nivel, con lo que un comportamiento se traducirá a un conjunto de tareas de bajo nivel.

Hay que recordar que, tal y como se presentó en el capítulo 2, un modo de funcionamiento se define por un conjunto de tareas. Además, los conjuntos de tareas que definen distintos comportamientos no tienen por qué ser disjuntos. De hecho, distintos comportamientos se pueden diferenciar en tan sólo una tarea, tal y como ocurre en el caso de querer activar un *in-agent* / tarea esporádica crítica, en el que, como ya se comentó en el capítulo 4, se va a implementar como un cambio de modo (en realidad dos, uno para incorporar la tarea, y otro para eliminarla una vez que se ha ejecutado, lo que ocurre es que este segundo cambio de modo tan sólo supone eliminar la tarea de la lista de tareas activas).

Desde un punto de vista del modelo de sistema, la ejecución de una tarea en un Agente ARTIS ($\mathcal{AA}$), según lo considerado hasta ahora, se dividía en dos partes:

Parte Crítica: formada a su vez por una parte inicial y una final, que tienen que ejecutarse para el correcto funcionamiento del sistema. Para ello, el usuario les define un plazo máximo de finalización o *deadline*, antes del cual debe haber finalizado su ejecución.

Parte Opcional: formada por aquellas partes de la tarea que no son imprescindibles para el funcionamiento del sistema, pero que podrían conseguir una mejor respuesta del sistema.

Según la clasificación de comportamientos presentada en la sección 4.4, el conjunto de todas las partes críticas de las tareas de los *in-agents* que conforman un comportamiento serán lo que formarán el *comportamiento reflejo innato* correspondiente.

Según esto, en la fase de diseño del agente se hace un análisis de la planificabilidad de las tareas de cada uno de los comportamientos por separado, de tal forma que todos los comportamientos reflejos innatos deberán ser planificables. Este análisis es el mismo análisis off-line de la planificabilidad que se presentaba en el capítulo 3, pues se puede ver, desde el punto de vista de los modos de funcionamiento, que en la versión previa a este trabajo de la arquitectura de $\mathcal{AA}$, ésta estaba formada por un sólo modo de funcionamiento.

7.3. Extensiones al $\mathcal{RS}$

7.3.1. Cambios de Modo

Un *Comport. Reflejo Innato* se puede ver como un *Modo de Funcionam.*

La gestión de los diferentes conjuntos de tareas que forman cada uno de los comportamientos reflejos innatos se traduce en lo que dentro de la literatura de sistemas de tiempo real se llama *cambios de modo* (tal y como se presentó en el capítulo 2). Así, cada *comportamiento reflejo innato* se puede ver como un *modo de funcionamiento*, y habrá que realizar un cambio de modo para pasar de un comportamiento reflejo innato a otro.

Todas las tareas (traducción en el modelo de sistema de un $\mathcal{AA}$ de los *in-agents* que componen los diferentes comportamientos reflejos innatos del $\mathcal{AA}$) van a estar activas desde el inicio, para evitar el coste de creación de una tarea en el período de transición entre comportamientos.

La forma de gestionar todas esas tareas para que tan sólo se ejecuten las del comportamiento actual es por medio de unas listas de tareas que indican qué tareas pertenecen a qué comportamientos. Estas listas se generan en la traducción del lenguaje de usuario introducido por medio de la herramienta *InSiDE*, y permiten que tan sólo las tareas del comportamiento activo sean tenidas en cuenta por los distintos cálculos del $\mathcal{FLS}$.

MCR generada por una $\mu\mathcal{R}egla$

La petición de cambio de modo (*MCR*) en un $\mathcal{AA}$ va a venir generada por la activación de una regla de control o $\mu\mathcal{R}egla$ que va a ser la que va a decidir que dicho cambio de comportamiento es necesario.

Según se presentó en la sección 6.3, en la fase de diseño del $\mathcal{AA}$, se construye un autómata finito no determinista donde los nodos representan los diferentes comportamientos definidos en el agente (cada uno de los cuales corresponde a un modo de funcionamiento en el modelo de sistema), mien-

tras que los arcos representan las diferentes transiciones entre comportamientos establecidas (y que se implementarán por $\mu\mathcal{R}eglas$ cuyas acciones darán lugar a los cambios de modo correspondientes). Este *Autómata de Transiciones* define, por tanto, las diferentes transiciones entre modos de funcionamiento que podrán darse en el $\mathcal{AA}$, y de las cuales hay que calcular que cumplen los requerimientos de transición entre modos que se presentaban en la sección 2.4.2: planificabilidad, periodicidad, consistencia y prontitud.

Para realizar estas transiciones entre dos modos de funcionamiento, siguiendo el estudio presentado por Real en el documento citado previamente en el capítulo 2 [Real 2000], la mejor opción para la arquitectura de $\mathcal{AA}$ es utilizar un *protocolo de cambio de modo asíncrono con continuidad*. Interesa un protocolo asíncrono porque favorece la rapidez en las transiciones, y que aporte continuidad porque así las tareas que pertenezcan al modo antiguo y al nuevo y no sufran ningún cambio en sus características (*tareas inalteradas*) no se verán afectadas por dicho cambio.

Debido a los buenos resultados obtenidos por Real [Real 2000] con su protocolo de cambio de modo (que es asíncrono con continuidad), se estudió su aplicabilidad para el problema aquí planteado, teniendo en cuenta las variaciones entre el modelo que él plantea y el propio del $\mathcal{AA}$, así como las restricciones que se pueden aplicar, ya que su modelo es más genérico que el que aquí se necesita.

Tal y como se requiere en dicho protocolo un modo de funcionamiento viene determinado por un conjunto de tareas con las siguientes características temporales definidas: tiempo de ejecución en el peor caso, período, plazo máximo de ejecución, prioridad (establecida en el modelo de sistema en base al plazo máximo de ejecución, a menor plazo mayor prioridad), relaciones de uso de los recursos compartidos (*slots* del $\mathcal{KDM}$).

De esta forma según la clasificación establecida de las tareas en una transición entre dos modos de funcionamiento y que se presentaba en el capítulo 2, se identifican los siguientes grupos de tareas:

Tareas Completadas: Serán aquellas tareas periódicas críticas del modo previo que estén ejecutando su parte crítica final.

Tareas Abortadas: Serán las tareas acríticas y aquellas tareas periódicas críticas que estén ejecutando su parte inicial o su parte opcional que pertenezcan al modo previo.

Tareas Cambiadas: Aquellas tareas que pertenezcan a ambos modos con cambios en sus características temporales.

Tareas Invariables: Serán aquellas tareas que pertenezcan a ambos modos sin ningún cambio en sus características temporales.

Tareas Nuevas: Serán las propias del nuevo modo.

Para asegurar que se cumplen los requerimientos de transición entre modos, se calculan los *retardos de incorporación* de las *tareas nuevas* en cada posible transición entre modos, adaptando el algoritmo de cálculo de los retardos que se presentaba en [Real 2000] y que se comentaba en la sección 2.4.2.1.1. Esta adaptación se ha basado en tener en cuenta la estructura del modelo de tareas de un $\mathcal{AA}$. Así, a la hora de calcular el retardo causado a una tarea del modo nuevo por las tareas previas de menor prioridad que ella, hay que utilizar el tiempo de respuesta en el peor caso. Estas tareas previas podrán ser (según la clasificación anterior) o bien tareas completadas, o bien tareas abortadas. En el primer caso, dicho tiempo de respuesta viene dado por el tiempo de ejecución en el peor caso de la parte final de dicha tarea, mientras que en el segundo se utiliza el tiempo de ejecución en el peor caso de las parte inicial de la tarea.

El cálculo de estos retardos se realizará en la fase de diseño del agente, después de que el diseñador ha especificado las posibles transiciones entre comportamientos y se construye el AFND correspondiente. En este momento, se realiza el análisis de la planificabilidad de cada comportamiento y se estudia la viabilidad de las transiciones especificadas, calculando los retardos de las tareas oportunas siempre que estas transiciones sean posibles. En el caso de existir transiciones no planificables, habría que revisar el diseño del $\mathcal{AA}$.

Una activación de un nuevo comportamiento y, por consiguiente una transición entre modos de funcionamiento, se hará por medio de la acción *SetBehaviour(int nBehav)* de las meta-reglas. Por lo tanto, y siguiendo la nomenclatura propia de los cambios de modo que se presentaba en la sección 2.4.2, la ejecución de esta acción genera una petición de cambio de modo (*MCR*). Esto disparará la siguiente secuencia de acciones por parte del Módulo de Control:

1. El *Grado de Introspección* del $\mathcal{DS}$ pasa a ser del 100 %, ejecutándose a continuación dicho módulo para que éste vacíe la cola de eventos que

pertenecen al comportamiento antiguo.

2. El *Grado de Reactividad* pasa a ser 0 durante la transición entre comportamientos (después se recuperará el grado antiguo).

3. Si el $\mathcal{DS}$ está en funcionamiento, entonces cede su control al $\mathcal{RS}$ cuando ya no existan más eventos, ya que un cambio de comportamiento supone un cambio del conjunto de tareas *críticas*, con lo que debe ser controlado por la parte del Módulo de Control encargado de gestionar ese tipo de tareas, es decir, el $\mathcal{RS}$.

4. El $\mathcal{RS}$ aborta todas aquellas tareas previas abortables que no hayan empezado a ejecutar aún su parte inicial (eliminándolas de la lista de tareas activas).

De esta manera, cuando se solicita un cambio de comportamiento, entra en un estado de transición entre modos en el cual se eliminan todas las tareas del modo antiguo *abortables*. El resto de tareas del modo antiguo que no pertenecen al nuevo se irán eliminando conforme acaben la ejecución de sus respectivas partes finales (*tareas completadas*).

7.3.2. Grado de Reactividad

La *reactividad* no es una característica absoluta sino gradual

En el capítulo 4 se describía una de las aportaciones del presente trabajo que supone una forma de replantearse una de las características básicas de un agente, la *reactividad*. Así, se plantea que la reactividad no es una característica absoluta sino gradual. No sólo eso, sino que además dicho grado es dinámico y variará de acuerdo a la situación a la que deba enfrentarse el agente.

La forma en la que se gestiona el concepto de *Grado de Reactividad* en el modelo de sistema del $\mathcal{AA}$ viene fundamentada por la gestión del tiempo de procesamiento en un $\mathcal{AA}$ en este modelo.

En un $\mathcal{AA}$ ejecutable, después de ejecutar una parte inicial de una tarea, se calcula la holgura del sistema en ese momento, y si hay suficiente *slack* disponible, se lanza el *Servidor Deliberativo* ($\mathcal{DS}$) comunicándole cuanto tiempo tiene disponible para su ejecución, es decir, el *slack* actual.

En el caso en que el $\mathcal{DS}$ acabe sus tareas antes de consumir todo el *slack* disponible, devuelve el control de la ejecución al $\mathcal{RS}$, que puede así reaprove-

char ese tiempo disponible para adelantar parte de la ejecución de las tareas críticas[1].

Se define el *Grado de Reactividad* de un $\mathcal{AA}$, desde el nivel de abstracción de su modelo de sistema, como un valor real entre 0 y 1 que representa el porcentaje máximo (en tanto por 1) de la holgura (*slack*) disponible en cada momento que se le deja al agente para refinar su respuesta.

Además, tal y como se explicaba en el capítulo 4, el *Grado de Reactividad* es un valor dinámico, ya que va a variar para permitirle al agente adaptarse a un cambio significativo en la situación actual. Este grado permite ver el *slack* como un valor en el intervalo [0, *slack* máximo]. Por tanto, se tienen dos modos de ejecución[2] extremos (el de *slack* 0 –*Grado de Reactividad* 0– que corresponde al modo de emergencia o *Modo Reflejo* , y el de *slack* original o máximo –*Grado de Reactividad* 1– que corresponde al *modo cognitivo* o *Deliberativo en Tiempo Real*) con un número indeterminado de modos de ejecución intermedios (tal y como queda reflejado en la figura 4.1).

El uso de estos modos de ejecución no transgrede la planificabilidad del sistema asegurada en el diseño del mismo, ya que como se puede observar estos modos de ejecución tan sólo se diferencian en la utilización del tiempo de holgura o *slack* que se hace, sin poder ocasionar ningún obstáculo para el cumplimiento de los plazos máximos de ejecución de las partes críticas del sistema.

7.4. Extensiones al $\mathcal{DS}$

El $\mathcal{DS}$ ha sido extendido para dar cabida tanto al soporte de las extensiones del modelo de usuario como a un objetivo propio intrínseco de incrementar la flexibilidad del aprovechamiento del tiempo de holgura que hace este módulo.

Para lograr este último objetivo se ha extendido el ciclo de control, a la vez que se ha desarrollado el concepto de *Grado de Introspección* tal cual se

[1] Este método es utilizado incluso por la política de planificación de segundo nivel *SSS* para poder juntar distintos intervalos de *slack* y conseguir así aumentar las posibilidades de ejecución de tareas.

[2] No confundir *modo de ejecución* con *modo de funcionamiento*, ya que en el primer caso no varía el conjunto de tareas que se ejecutan, sino tan sólo las condiciones de ejecución, como puede ser el *slack* u holgura del sistema

definió en el capítulo 4.

Además, se ha desarrollado un mecanismo de gestión dinámica de eventos que posibilita el desarrollo de la gestión de *Focos de Atención*, así como el de la gestión de niveles interrumpidos.

Por último se ha extendido el $\mathcal{DS}$ para dar soporte al resto de acciones de las $\mu\mathcal{R}eglas$ definidas en el modelo de usuario (y que se describieron en el capítulo 6). Estas funcionalidades incluyen la gestión de la carga aperiódica acrítica y la modificación dinámica de la política de planificación de segundo nivel.

Con respecto a la política de planificación de segundo nivel, previo a este trabajo, cuando el *Servidor Inteligente* establecía (en su fase de inicialización) una estrategia de planificación de segundo nivel, se mantenía a lo largo de toda la ejecución del sistema. La mejora desarrollada en este caso es que esta característica no sea tan estática, sino que se pueda cambiar esta estrategia, permitiendo que este cambio se haga bien en respuesta a un evento, bien por medio de alguna condición interna como puede ser controlar si la calidad media conseguida hasta el momento no llega a un determinado umbral o si el número de niveles interrumpidos es mucho menor que el número de niveles completados, ...

7.4.1. Nuevo Ciclo de Control

Dadas las extensiones propuestas, las funciones principales del $\mathcal{DS}$ son:

- Recepción y respuesta a eventos. Esta respuesta se gestionará por medio de $\mu\mathcal{R}eglas$, cuyas acciones pueden suponer desde el cambio del valor de un *slot*, a la activación de un *in-agent* opcional para que pueda ser tenido en cuenta en la planificación de tareas a ejecutar.
- Control de cuáles de las siguientes entidades (de ejecución no obligatoria para el sistema) pueden ser elegibles para su ejecución en este ciclo:
  - *In-agents* opcionales esporádicos activos, que podrán ser activados como respuesta a eventos (es una de las posibles acciones de las $\mu\mathcal{R}eglas$).
  - Niveles de las distintas MKSs que puedan estar activas, teniendo en cuenta que si la MKS implementa métodos múltiples, todos los ni-

veles serán elegibles, mientras que si no es así, tan sólo el siguiente nivel al último ejecutado con éxito será elegible. Además hay que tener en cuenta, en este último caso, que en cuanto un nivel no se ejecute con éxito, ya no se seguirán ejecutando más niveles de esa MKS.

Hay que tener en cuenta que, aunque no gestiona tareas con restricciones temporales críticas, el $\mathcal{DS}$ debe controlar el tiempo que se ejecuta, pues su ejecución debe restringirse a los intervalos de tiempo ociosos que existen entre las ejecuciones de las tareas con restricciones críticas. Además, para salvaguardar la integridad del $\mathcal{DS}$ y de la información que gestiona, deberá ser él quien ceda el control al $\mathcal{RS}$ cuando detecte que el tiempo de que dispone no es suficiente para poder completar con garantías ninguna acción de interés. Cada vez que se invoca se debe empezar por un ciclo de ejecución nuevo, para que el $\mathcal{DS}$ no esté trabajando con información no actualizada. Es por esto que resulta tan importante el control de este tiempo de holgura, y el mejor aprovechamiento del mismo. Así, para poder dar cobertura al objetivo de flexibilizar el aprovechamiento que del tiempo de holgura se hace en el $\mathcal{AA}$, se define un nuevo ciclo de control para el $\mathcal{DS}$.

Este nuevo ciclo de control del $\mathcal{DS}$ se divide en dos partes que corresponden lógicamente a dos módulos, el *Gestor de Eventos* (*Event Manager* —$\mathcal{EM}$—) y el *Planificador de Segundo Nivel* (*Second Level Scheduler* —$\mathcal{SLS}$—). El primero recibe los eventos significativos y reacciona a ellos, estando formado por las fases de *Trigger* (que como consecuencia de la incorporación de eventos pasa a ser más compleja encargándose ahora de la recepción de eventos, y gestión de Niveles / *in-agents* activos) y *Condition Testing* del *satisfacing cycle*. El segundo módulo está al cargo de la planificación del uso del tiempo de *slack* y comprende las fases de *Rating*, *Schedule* e *Interpret*. Esta correspondencia entre estos módulos y las fases del ciclo de control se pueden observar en la figura 7.1.

Grado de Introspección variable

Hay que tener en cuenta que el que un $\mathcal{AA}$ varíe su *grado de reactividad* en respuesta a cambios en el entorno supone una variación en el tiempo del que va a disponer el $\mathcal{DS}$ para poder ejecutarse, y, por tanto, una variación del tiempo asignado a cada una de las dos partes que lo conforman. Esta asignación no tiene por qué ser equitativa. Por ejemplo, al *Gestor de Eventos* ($\mathcal{EM}$) se le puede dejar tanto tiempo como necesite. Sin embargo, ésta no tiene por qué ser la mejor opción siempre. A veces, puede que la mejor opción sea con-

trolar qué porcentaje de tiempo del que dispone el $\mathcal{DS}$ se le va a conceder al $\mathcal{EM}$. Este porcentaje, que correspondería al concepto del modelo de usuario denominado *Grado de Introspección*, puede ser modificado mediante reglas de Control.

Uno de los mecanismos para modificar el comportamiento del $\mathcal{DS}$, por tanto, es variar su interés en los cambios en el entorno (modificar su *grado de introspección*). Es decir, si se le va a dejar que reciba todos los eventos ocurridos o sólo alguno. Se debe tener en cuenta que esta variación va a ser gradual, siendo los comportamientos extremos que atienda a todos los eventos (el $\mathcal{DS}$ está centrado en el entorno) o que no atienda a ninguno (el $\mathcal{DS}$ está totalmente centrado en sus propios objetivos). Además, para poder responder a los eventos más adecuados en cada momento (sobre todo cuando no vaya a disponer de tiempo para responder a todos) los eventos estarán ordenados por su importancia.

Por todo esto, el *ciclo de control* pasa a ser el siguiente:

```
Mientras hay trabajo pendiente y suficiente tiempo {
  Mientras hay eventos
            y lo permite el grado de introspección {
       Trigger(TRIGGER * fT);}
  Si (hay bastante tiempo para (COND_TEST)) {
       Condition_Testing();
       Si (hay bastante tiempo para (RATING)) {
         Rating();
         Si (hay bastante tiempo para (SCHEDULE)) {
           scheduledAG = Schedule();
           Si (hay bastante tiempo para (INTERPRET))
             Interpret( scheduledAG );
} }    } }
```

7.4.2. Gestión Dinámica de Eventos

Tal y como se presentaba al comienzo de esta sección 7.4, una de las extensiones realizadas al $\mathcal{DS}$ se ha habilitado un mecanismo de gestión dinámica de eventos que va a permitirle al Módulo de Control llevar a cabo dentro del modelo de sistema el concepto de *Foco de Atención*, así como llevar a cabo la gestión de niveles interrumpidos (funcionalidad propia del modelo de sistema del $\mathcal{DS}$).

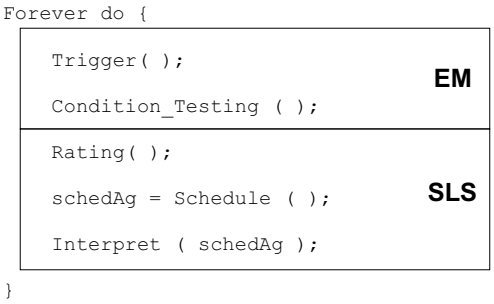

Figura 7.1: Correspondencia entre el ciclo de control y los módulos del $\mathcal{DS}$

Este mecanismo va a permitirle al Módulo de Control comunicar al $\mathcal{KDM}$ en qué eventos está interesado o en cuáles ya no lo está.

Esta gestión es realizada por el Módulo de Control utilizando dos funciones de interfaz con el $\mathcal{KDM}$que corresponden en el modelo de sistema a las acciones correspondientes de las $\mu\mathcal{R}eglas$que se presentaron en el capítulo 6:

```
KDM_Add_Event ( <class.instance.slot>, <EventType>,
                <Importance> );
```

Esta función permite indicar al $\mathcal{KDM}$ que debe informar al Módulo de Control cuando se produzca un evento del tipo indicado en el segundo parámetro (puede recoger más de un tipo) sobre el *slot* indicado en el primer parámetro. Además este evento tendrá la importancia indicada como tercer parámetro.

```
KDM_Release_Event ( <class.instance.slot>, <EventType> );
```

Esta función permite indicar al $\mathcal{KDM}$ que ya no tiene que informar al Módulo de Control sobre los eventos del tipo $< EventType >$ relacionados con el *slot* indicado como parámetro.

7.4.3. Focalización del $\mathcal{DS}$

La focalización de un agente, del $\mathcal{AA}$ en este caso, se puede ver a diferentes niveles. En un nivel conceptual, el agente focaliza su atención concentrando su capacidad de razonamiento en un subconjunto concreto de su co-

nocimiento y / o percepción al que se denomina *Foco de Atención*. Desde otro punto de vista más cercano al nivel de usuario del $\mathcal{AA}$, un *foco de atención* lo componen un conjunto de *slots* del $\mathcal{KDM}$, y cuando ese foco de atención está activo en el $\mathcal{AA}$, se priorizará cualquier razonamiento en el que dichos datos formen parte de acuerdo a la importancia relativa de ese foco de atención con respecto al resto de focos de atención activos del $\mathcal{AA}$.

De esta manera, este mismo mecanismo de generación dinámica de eventos comentado en el apartado previo, sirve para focalizar la atención del $\mathcal{DS}$. De este modo, incluyendo dentro de la parte derecha de reglas de control invocaciones a las funciones *KDM_Add_Event()* y *KDM_Release_Event()*, y habilitando/deshabilitando las reglas de control oportunas, se consigue que el $\mathcal{DS}$ varíe la focalización de su atención.

Cabe destacar en este punto que la focalización del $\mathcal{DS}$ no sólo implica la anteriormente comentada selección de los datos del $\mathcal{KDM}$ en los que más está interesado el $\mathcal{DS}$, sino que también incluye las respuestas preestablecidas a cambios significativos de los mismos, es decir, los conjuntos de *Meta-Reglas* (llamados por este motivo *Focos de Atención*). Hay que tener en cuenta, por tanto, que frente a un mismo conjunto de *slots* del $\mathcal{KDM}$ se pueden tener distintas focalizaciones del $\mathcal{DS}$ cambiando los conjuntos de $\mu\mathcal{R}eglas$ que se utilizan. Además utilizando la siguiente función (o acción de las $\mu\mathcal{R}eglas$) de interfaz se pueden activar / desactivar *Focos de Activación* (conjuntos de $\mu\mathcal{R}eglas$) permitiendo focalizar el $\mathcal{DS}$:

```
SetFocus ( <Attention Focus id.>, <Attention_Degree> );
```

Así, esta acción de las $\mu\mathcal{R}eglas$ activa el foco de atención con identificador *Attention Focus id.* con el grado de atención indicado en *Attention Degree*.

Además, la función anterior permite desactivar el foco de atención con identificador *Attention Focus id.* si el grado de atención que se le pasa como parámetro en *Attention Degree* es 0.

Estos dos tipos de modificaciones de la focalización del $\mathcal{DS}$ (variar los eventos y variar los focos de atención activos) se pueden llevar a cabo gracias a que en el $\mathcal{KDM}$ se encuentran almacenados, para cada KS, qué *slots* contienen la información que necesita para realizar sus cálculos.

Utilizando esta información, y teniendo en cuenta que un foco de atención

del agente va a venir determinado por un conjunto de *slots*, el $\mathcal{DS}$ puede activar / desactivar *in-agents* de acuerdo al foco de atención del agente en un instante dado.

Para ello, en la fase de *Condition Testing* del ciclo de control, se comprueba, para cada elemento activo de la *triggering subagenda* si está interesado en la información de alguno de los focos activos. Si no es así, no será copiado a la *condition-testing subagenda*. En caso contrario, se copiará, modificando la importancia de dicho elemento de acuerdo al grado de atención que tenga el foco en el cual está interesado.

7.4.4. Gestión de Niveles Interrumpidos

Dentro de las modificaciones que se han realizado al $\mathcal{DS}$, hay algunas relativas a su funcionamiento interno que no tienen correspondencia con el modelo de usuario. Una de estas modificaciones es la relativa a la gestión de niveles interrumpidos.

Si consideramos que un nivel puede ser interrumpido, para su posterior reanudación, el $\mathcal{DS}$ debe controlar los casos en los que dicha reanudación tendrá sentido. Es decir, si un nivel interrumpido utiliza para sus cálculos información de una serie de *slots* que han modificado su valor desde que el nivel fue interrumpido, ese nivel debería comenzar su ejecución de cero si se decide que hay que ejecutarlo.

Así, al interrumpir la ejecución de un nivel, el $\mathcal{DS}$ debe comunicar al $\mathcal{KDM}$ que ahora quiere ser informado de la modificación de los *slots* en los que está interesado dicho nivel. Para ello, se hace uso de la función de interfaz anteriormente comentada *KDM_Add_Event()*.

Esta función es invocada una vez por cada *slot* del $\mathcal{KDM}$ en el que esté interesado el nivel interrumpido, indicando que el tipo de evento es de modificación (sólo en aquellos casos en los que el $\mathcal{DS}$ no reciba ya dicho evento de ese *slot*).

Además, hay que generar, también de forma dinámica, un conjunto de reglas de control para gestionar dichos eventos.

La llegada de uno de estos eventos supone, por tanto, que se ha modificado uno de los *slots* en los que está interesado el nivel interrumpido, por lo que la información con la que estaba trabajando dicho nivel cuando se le in-

terrumpió ya no es válida. Por tanto, ese nivel no debe reanudar su ejecución a partir de donde se quedó, sino que si se decide que debe ejecutarse, se debe comenzar su ejecución desde el principio.

De esta manera, una de las acciones que se deben llevar a cabo en estas $\mu\mathcal{R}eglas$ es eliminar de la *rating subagenda* la instancia del nivel (como no se elimina de las otras subagendas, puede ser elegido para su ejecución en otro momento si aún continúa activo).

Debido a que ya han cumplido su función, y con tal de no sobrecargar al $\mathcal{KDM}$ ni al $\mathcal{DS}$, habría que deshabilitar las reglas de control que se habilitaron para controlar el nivel interrumpido, e informar al $\mathcal{KDM}$ para que no genere los eventos de modificación de los *slots* en los que estaba interesado dicho nivel (esto último siempre que dicho evento no le siga interesando por otra razón). Para informar al $\mathcal{KDM}$ haremos uso, en esta ocasión, de la función *KDM_Release_Event()*.

Además, haciendo uso también de esta información, si al activar la parte opcional de un *in-agent* crítico los *slots* en los que está interesado no han modificado su valor desde la última vez que se activó, se puede tratar de continuar su ejecución desde donde se quedó. Esto es relativamente fácil si estaba ejecutándose un MKS con métodos múltiples, donde se pasaría a ejecutar el siguiente nivel al último completado con éxito, pero si el MKS es de refinamiento, lo único que se puede hacer es tomar como primera solución la última que se calculó.

7.5. Otras Extensiones

Además de las extensiones presentadas a lo largo del presente capítulo, el modelo de sistema de un $\mathcal{AA}$ ha sido extendido para permitir la gestión de Eventos y la de $\mu\mathcal{R}eglas$ o focos de atención.

Esta gestión de Eventos supone la incorporación de un mecanismo de comunicación asíncrono que permite comunicar las dos partes o submódulos del Módulo de Control, el $\mathcal{RS}$ y el $\mathcal{DS}$ entre sí y con el $\mathcal{KDM}$.

En cuanto a la gestión de focos de atención o $\mu\mathcal{R}eglas$ el mecanismo incorporado permite la creación, activación, interpretación y borrado dinámicos de focos de atención, entendidos estos como conjuntos de $\mu\mathcal{R}eglas$. Todas

estas operaciones están accesibles tanto desde el $\mathcal{RS}$ como desde el $\mathcal{DS}$. El diseño de este mecanismo está realizado pensando en la futura incorporación de un algoritmo de aprendizaje que permita aprender/olvidar $\mu\mathcal{R}eglas$.

Estas dos extensiones son detalladas en el anexo A y en el capítulo 8, respectivamente.

7.6. Conclusiones

En este capítulo se han presentado las extensiones realizadas en el presente trabajo al modelo de sistema de la arquitectura de agente ARTIS. Estas extensiones serán completadas con lo expuesto en los dos capítulos siguientes que se centran en la gestión de eventos y la de meta-reglas, respectivamente.

Las extensiones al modelo de sistema se han centrado sobre todo en el Módulo de Control del $\mathcal{AA}$, tanto para incorporar las funcionalidades propias del meta-razonamiento que se ofrecen al modelo de usuario, como para optimizar el proceso realizado por dicho módulo, haciendo más flexible y adecuado el uso del tiempo de procesador que se dedica al propio Módulo de Control, o incorporando como parte del mecanismo de gestión de eventos una forma de comunicar los dos submódulos del Módulo de Control permitiéndoles así comunicarse el tiempo disponible (del $\mathcal{RS}$ al $\mathcal{DS}$) o indicar que no desea realizar más operaciones (del $\mathcal{DS}$ al $\mathcal{RS}$). Además, se ha incorporado un algoritmo que permite gestionar diferentes modos de funcionamiento, traducción de los diferentes comportamientos especificados en el modelo de usuario.

Además, cabe destacar la definición del concepto de *Grado de Reactividad* variable. Esta nueva aproximación a la reactividad de una agente como un grado en lugar de una característica que permite definir agentes con una reactividad ajustable a la situación a la que se enfrente, en contraposición a las características más rígidas de las arquitecturas de agente tradicionales. Hay que tener en cuenta que este grado define dos situaciones extremas (de máxima reactividad y de máxima deliberación), pero con infinitos estados intermedios. Por otra parte, se ha implementado este grado como parte del modelo de sistema del $\mathcal{AA}$.

8

Gestión de Meta-Reglas ($\mu Reglas$)

"Hell, there are no rules here–
we're trying to accomplish something."
– Thomas A. Edison, (1847-1931)

8.1. Introducción

El presente capítulo se centra en la definición del concepto de meta-regla ($\mu Regla$), y de cómo se ha introducido este concepto dentro de la arquitectura de $\mathcal{AA}$ en todos los niveles del mismo, prestando una especial atención al nivel de sistema, debido a que la dificultad entrañada por el hecho de que la zona de memoria donde se almacenan las $\mu Reglas$ debe ser accesible tanto desde Linux como desde RT-Linux, y a que aunque actualmente no se ha incorporado aún ningún método de aprendizaje, el mecanismo de gestión está preparado para añadir y borrar dinámicamente $\mu Reglas$.

De esta manera, el capítulo se estructura en tres secciones. La primera presenta la estructura abstracta del concepto de $\mu Regla$ y de *foco de atención*.

En la siguiente sección se retoma el tema de la estructura de usuario, es decir, del denominado lenguaje de control (que ya se comentó en el capítulo 6 y cuya sintaxis completa se recoge en el anexo B), aunque en este caso se presta un mayor detalle a la presentación de las diferentes acciones que se pueden realizar en una $\mu\mathcal{R}egla$ (debido a que la implementación de los diferentes mecanismos que las soportan se presenta como parte del capítulo 7 y del anexo A). La tercera y última de estas secciones se centra en la estructura interna del denominado *almacén de meta-reglas*, presentando la solución adoptada para dar respuesta a las necesidades que se planteaban de variabilidad en el conjunto de $\mu\mathcal{R}eglas$ y de acceso desde diferentes puntos.

8.2. Estructura Abstracta

De forma general, una meta-regla ($\mu\mathcal{R}egla$) se define como una tupla de la siguiente manera:

$$\mu\mathcal{R}egla_i = (\varepsilon_i, \Gamma_i, \Lambda_i), \text{ donde:}$$

- ε_i *(Tipo de Evento):* Indica el tipo de evento al que debe responder la meta-regla ($\mu\mathcal{R}egla$) *i*. Como se presenta en el capítulo 6 y en el anexo A, tan sólo los eventos $\mathcal{KDM}$ – Módulo de Control tendrán asociados un *slot* del $\mathcal{KDM}$ (que será el que generará el evento en cuestión).

- Γ_i *(Condición):* Conjunto de 0 ó más elementos que reflejan qué situación se debe dar para que se ejecute la meta-regla ($\mu\mathcal{R}egla$) *i*. Dicho de otra forma, qué se tiene que cumplir para que se ejecute la acción o acciones asociada.

  La forma en la que se expresarán los distintos elementos que conforman la condición será mediante una expresión de comparación, teniendo que ser todas las comparaciones ciertas para que la condición de la meta-regla se cumpla, es decir, existe una *y lógica* implícita que une todas las subcondiciones de la meta-regla ($\mu\mathcal{R}egla$).

- Λ_i *(Acción):* Conjunto de 1 ó más elementos que indican los efectos a llevar a cabo si se ejecuta la meta-regla ($\mu\mathcal{R}egla$) y se cumple su condición.

La definición de una meta-regla ($\mu\mathcal{R}egla$) está relacionada con la definición formal de comportamiento y de la transición entre comportamientos

vista anteriormente. Esto es debido a que uno de los posibles tipos de acciones que se pueden realizar en una meta-regla ($\mu\mathcal{R}egla$) es el de cambio de comportamiento, aunque no el único. Es decir, la definición de cambio de comportamiento $\forall i : \pi_i^{[t,t']} \rightarrow [b_i]g_i^{[t,t']}$ está definiendo un tipo particular de meta-regla ($\mu\mathcal{R}egla$), en el que Λ_i es un conjunto formado por tan sólo un elemento, el del cambio al comportamiento b_i.

Por otra parte, para que se cumpla la condición de una meta-regla *i* en el intervalo dado, se deben cumplir básicamente dos partes, que se dé el evento asociado y que se cumplan todos los elementos de Γ_i, con lo que la condición de activación π_i se redefine de la siguiente manera:

$$\pi_i = \varepsilon_i(\kappa_i) \wedge (\bigwedge_{\forall j} \Gamma_{ij})$$

Con todo esto se define de manera formal una meta-regla como la 4-tupla arriba mencionada tal que:

$$\forall\ \mu\mathcal{R}egla_i : (\varepsilon_i(\kappa_i) \wedge (\bigwedge_{\forall j} \Gamma_{ij})) \rightarrow [\Lambda_i]$$

Las meta-reglas se agrupan en *focos de atención* de forma que, en un instante dado, tan sólo un subconjunto de los focos de atención existentes estará activo, teniendo cada uno de ellos un grado de atención. Tan sólo las meta-reglas que pertenezcan a focos de atención activos son tenidas en cuenta a la hora de responder a eventos. La probabilidad de que una meta-regla sea la elegida para responder a un evento, de entre todas las meta-reglas de los focos de atención activos que podrían responder a dicho evento, es proporcional al grado de atención del foco activo al que pertenece.

El objetivo de esta representación es permitir el aprendizaje y/o olvido de meta-reglas, de forma que aprender una nueva meta-regla supone incorporarla al foco de atención activo con mayor grado de atención. En cuanto a olvidar meta-reglas, sólo se podrán olvidar aquellas meta-reglas que se hayan incorporado fruto del aprendizaje.

Hay que resaltar que sólo se permite el aprendizaje de meta-reglas, pero no el de focos de atención.

8.3. Estructura de Usuario

Tal y como se veía en el capítulo 6, existe un lenguaje de usuario, con una sintaxis de tipo CLIPS, que permite la definición de $\mu\mathcal{R}eglas$, o de forma más concreta, permite la definición de *focos de atención* que estarán compuestos por $\mu\mathcal{R}eglas$. Tal y como se puede ver en el ya mencionado capítulo 6 (y en el anexo B donde se recoge el resumen de este lenguaje de usuario) una $\mu\mathcal{R}egla$ se compone de las tres partes correspondientes a las presentadas en el apartado anterior:

- Evento de disparo de la regla: En esta parte el usuario especifica el evento frente al cual se podrá activar la $\mu\mathcal{R}egla$ que está definiendo. Si este evento es del tipo $\mathcal{KDM}$ – Módulo de Control, además deberá especificar el *slot* que generará dicho evento.

- Condición: Además del evento de disparo de la regla, para la ejecución de la acción o acciones asociadas a la misma se debe cumplir una condición que especifique el usuario. Esta condición está compuesta por una o más expresiones relacionales de obligado cumplimiento. El objetivo es comprobar que se cumple la situación prevista por el usuario en la fase de diseño frente a la cual debe actuar la $\mu\mathcal{R}egla$. Para ello, se pueden comprobar los valores de los *slots* que se consideren oportuno.

- Acción: Esta parte está compuesta por una secuencia de acciones a llevar a cabo cuando se da el evento de disparo y se cumple la condición de la $\mu\mathcal{R}egla$.

En el siguiente apartado se comentan con más detalle las diferentes acciones que se pueden realizar en una $\mu\mathcal{R}egla$.

8.3.1. Acciones de las Meta-Reglas

Las diferentes acciones que se pueden realizar en una $\mu\mathcal{R}egla$ se pueden dividir en tres grupos de acuerdo a la parte que afectan.

8.3.1.1. Acciones sobre la Gestión del Tiempo

Este tipo de acciones permiten variar las cantidades de tiempo de procesador que se utilizan para las distintas operaciones en el $\mathcal{AA}$. Así, en el capítulo 7 se han definido los conceptos de *Grado de Reactividad* y *Grado de Introspección*, como indicadores respectivamente del porcentaje del tiempo de holgura dedicado al proceso deliberativo, y del porcentaje del tiempo dedicado al proceso deliberativo que se dedica a atender a cambios en lugar de dedicarlo a la deliberación pura.

La sintaxis de estas dos acciones es, para el caso de cambio del grado de reactividad:

```
SetReactivityDegree ( <number> )
```

Mientras que para el cambio del grado de introspección es la siguiente:

```
SetAttentionDegree ( <number> )
```

En ambos casos, el argumento es un número entre 0 y 1 que indica el nuevo valor a establecer para el grado correspondiente.

8.3.1.2. Acciones sobre las Tareas

Estas acciones permiten modificar directamente las operaciones a ejecutar en el proceso de razonamiento, bien porque se cambia el algoritmo de planificación utilizado (*SetPolicy*), bien porque se añaden nuevas operaciones (niveles esporádicos), o porque se cambia completamente el proceso de razonamiento, es decir, se hace un cambio de comportamiento.

De esta manera, la sintaxis de estas acciones es la siguiente. Para el cambio de política de planificación (donde el número que se utiliza en realidad se puede indicar mediante el nombre de la política de planificación, gracias a un conjunto de constantes predefinidas):

```
SetPolicy ( <number> )
```

Para añadir un nivel esporádico al conjunto de componentes deliberativos activos:

```
TriggerSporadicLevel ( <identificador> )
```

Por último, para poder cambiar el comportamiento activo actual (que dispararía en el modelo de sistema una petición de cambio de modo o MCR) existe la siguiente acción:

```
SetBehaviour ( <identifier> )
```

8.3.1.3. Acciones de cambio de Foco de Atención

Por último, aunque no por ello menos importante, está el conjunto de acciones que permiten el ajuste dinámico del proceso de meta-razonamiento, evitando que se le informe, o que tenga que comprobar cosas que en un instante dado no son de interés.

Para ello, se puede activar un foco de atención por medio de la siguiente acción, indicando además el *grado de atención* que tendrá dicho foco, es decir, lo atento que debe estar el agente a ese foco de atención, es decir, la importancia de dicho foco con respecto al resto de focos de atención activos. Cuando se establece que un foco de atención tiene como grado de atención 0, se desactiva.

```
SetFocus ( <identifier>, <Attention_Degree> )
```

Además de que el agente no disponga de un conjunto desmesurado de $\mu\mathcal{R}eglas$ utilizables en un momento dado, también se debe controlar que el número y tipo de eventos que se vaya generando varíe, para no saturar al agente con información no relevante en un momento dado. Para ello, se pueden añadir y eliminar eventos utilizando las dos acciones siguientes (donde el número que aparece como segundo argumento es el tipo de evento declarado por medio de un conjunto de constantes predefinidas, mientras que el

primer argumento será el *slot* del $\mathcal{KDM}$ si es un evento de este tipo):

```
KDM_Add_Event ( <clase>.<instancia>.<slot>,
                <number>, <number> )            |

KDM_Release_Event ( <clase>.<instancia>.<slot>,
                    <number> )
```

Cabe resaltar que al añadir un evento, se incluye como último argumento la importancia relativa de dicho evento con respecto a los demás.

8.4. Estructura Interna

Una de las dificultades encontradas en el diseño del presente trabajo fue decidir cómo representar internamente las reglas de control o meta-reglas, pues se quería tener en cuenta que la representación utilizada fuese lo suficientemente flexible para permitir el aprendizaje/olvido dinámico de las mismas de forma simple y eficiente, así como el acceso eficiente a la información de las mismas. No hay que olvidar que esta eficiencia que se pide en todas las operaciones es debida a que se está trabajando en sistemas que van a tener un tiempo limitado de ejecución.

Gran parte del estudio que se ha realizado con respecto a las meta-reglas está relacionado con la forma en la que se representan internamente. Las opciones que se barajaban colocaban las reglas de control o bien en el $\mathcal{KDM}$ o bien sólo en la memoria local (del $\mathcal{DS}$ y del $\mathcal{RS}$). Una de las ideas que se buscaba con las reglas de control es que su representación interna fuese lo suficientemente flexible para permitir modificar algunos argumentos de las mismas (bien en las condiciones, bien en las acciones) e incluso poder añadir nuevas reglas fruto de los resultados de algún algoritmo de aprendizaje que se incorporase al $\mathcal{DS}$. Para poder realizar esto, se buscó una representación interna de las reglas que utilizase listas (o vectores debido a las restricciones existentes, si se decidía que debían estar almacenadas en el $\mathcal{KDM}$ para poder ser modificadas por alguna otra tarea que tenga acceso al mismo). Además, había que tratar de indexar la tabla de forma que se pudiera gestionar de la forma más eficiente posible.

De esta manera se va a tener un almacén de un tamaño pre-establecido

en la fase de diseño del sistema (para evitar tener que trabajar con memoria dinámica) con capacidad para almacenar todas las meta-reglas que conozca el $\mathcal{AA}$ en un instante dado. En este almacén se codificarán las meta-reglas que serán interpretadas en ejecución.

Dentro de las meta-reglas que conoce el $\mathcal{AA}$ hay que diferenciar entre las que son *innatas* (especificadas por el diseñador) y las que son aprendidas. Estas últimas van a poderse tanto aprender como olvidar (borrar definitivamente).

Así, como el número de meta-reglas conocidas por el $\mathcal{AA}$ en un instante dado puede variar, el control de la memoria que va a tener disponible para la representación de dichas reglas se ha realizado de forma similar al utilizado para la gestión de eventos. De esta manera, el almacén de reglas estará formado por dos partes, una primera donde estarán el conjunto de reglas conocidas por el $\mathcal{AA}$ y otra donde estarán un conjunto de *reglas vacías* que gestionarán la memoria disponible para aprender nuevas meta-reglas.

Además las meta-reglas se agruparán en *focos de atención* de entre los cuales hay que distinguir entre los que están activos en un instante dado y los que no lo están. Adicionalmente, las meta-reglas están indexadas para su acceso por el tipo de evento y el *slot* generador del evento.

De esta manera, con el objetivo en mente de dar soporte a un futuro mecanismo de aprendizaje capaz de aprender/olvidar meta-reglas se diseñó una solución similar a la adoptada para la gestión de eventos, utilizando un buffer de memoria accesible desde cualquiera de los dos submódulos del Módulo de Control.

8.4.1. Gestión Interna

El mecanismo desarrollado que permite la gestión del almacén de meta-reglas es similar al de la lista de nodos vacíos del almacén de eventos que se presenta en el anexo A, con la particularidad de que en este caso se dispone de diversas listas de nodos vacíos que hay que combinar para aprender/olvidar meta-reglas. Así existirán diferentes tipos de nodos para cada una de las partes de una meta-regla, para la propia meta-regla, para definir un foco de atención, y para definir un foco de atención activo.

Podemos ver su resumen en la figura 8.1

De esta manera, existen las siguientes listas de nodos vacíos:

- Punteros a *slot* (registros de tipo *regpSlot*).
- Registros de expresiones (*regExprNode*).
- Registros de acciones (*regActNode*).
- Punteros a meta-reglas (*regpMetaRuleNode*). Estos punteros establecen una lista entre las distintas meta-reglas (*regMetaRule*) vacías.

Además de las listas anteriores hay:

- NMAXMETARULES[1] registros de tipo *regMetaRule*.

Las listas de nodos vacíos (*aregEmptyCondNode* y *aregEmptyActNode*) sirven para gestionar el aprendizaje/olvido de nuevas reglas (siempre dentro de los máximos permitidos por NMAXMETARULES).

Dentro de las meta-reglas hay dos formas distintas de gestionar su olvido. Así, si la meta-regla fue añadida mediante el proceso de aprendizaje se puede olvidar definitivamente, mientras que si la meta-regla es innata no se puede borrar definitivamente (la única forma de olvidarla es desactivar el foco al que pertenezca).

8.4.2. Requerimientos de Memoria

Hay que reservar memoria de tipo *mbuff* (con un sistema similar del almacén de eventos) para almacenar las meta-reglas y los focos de atención en los que se estructuran.

Para ello, teniendo en cuenta que el número máximo de meta-reglas viene determinado por la constante *NMAXMETARULES*, se reservará espacio para (ver figura 8.1):

- Un registro de tipo *aregEventType* para implementar el vector/lista que permite la indexación por tipo de evento.

[1] Este valor constante debería generarse en el proceso de compilación, por tanto habrá un fichero de cabecera que se generará como resultado de dicho proceso

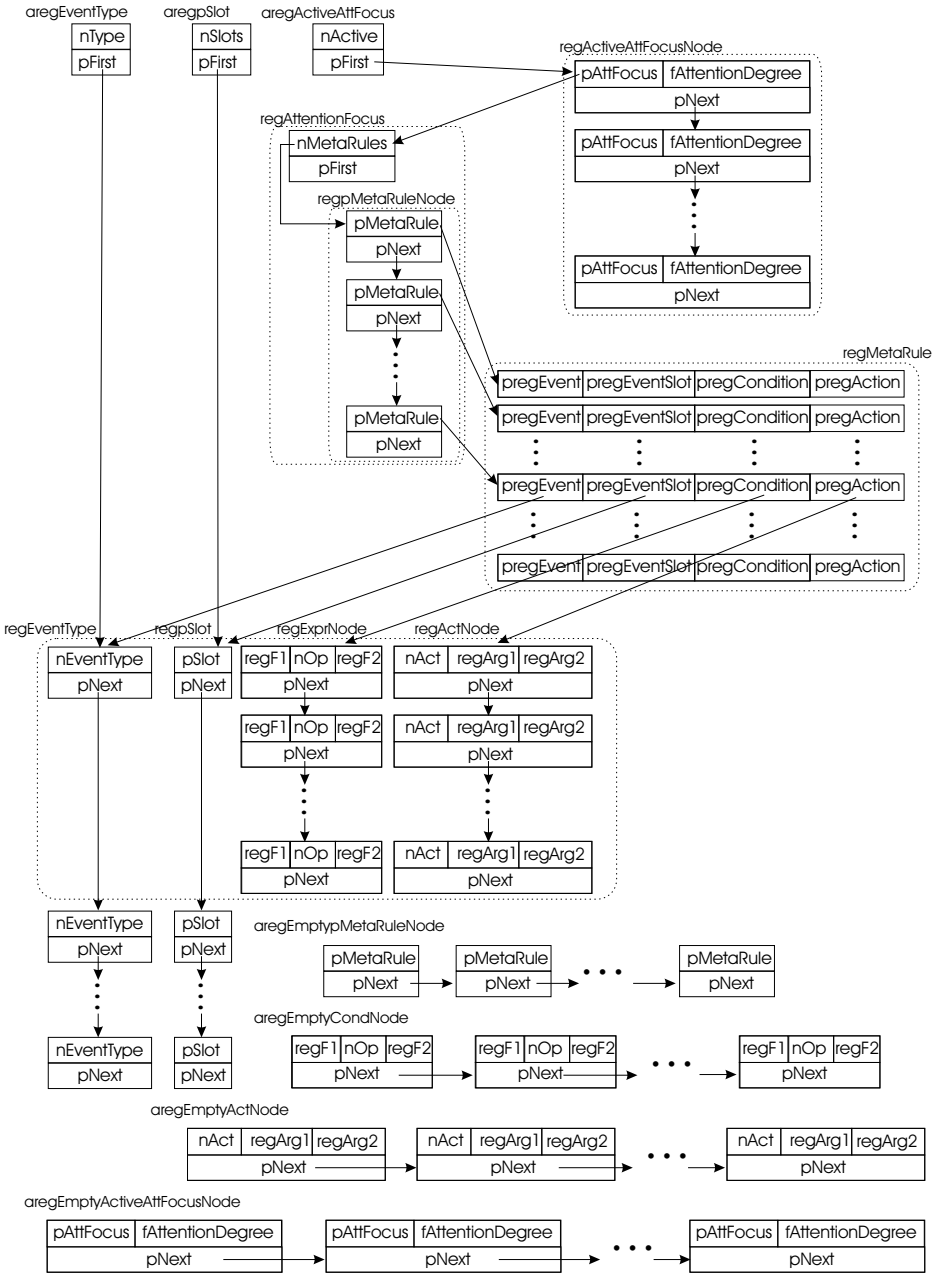

Figura 8.1: Estructura Interna del Almacén de Meta-Reglas

- Un registro de tipo *aregpSlot* para implementar el vector/lista que permite la indexación por *slot*.

- Un registro de tipo *aregActiveAttFocus* para implementar el vector/lista que permite la indexación por foco de atención activo.

- *NMAXMETARULES* registros de tipo *regMetaRule* (cada uno permitirá definir una meta-regla).
- *NMAXMETARULES* registros de tipo *regEventType* y otros tantos de tipo *regpSlot*, ya que cada meta-regla puede tener como máximo una de estas estructuras asociadas (*regEventType*para indicar el tipo de evento, y *regpSlot*para indicar el *slot* generador del evento).
- Un número de registros de tipo *regExprNode* que vendrá dado por el máximo entre estos dos valores:
  - El doble de *NMAXMETARULES*.
  - El número de condiciones existentes en las meta-reglas definidas por el usuario (el número de *regExprNode* usados) más el doble del número de meta-reglas que aún se podrían usar (*NMAXMETARULES* menos el número de meta-reglas definidas por el usuario).
- Un número de registros de tipo *regActNode* que vendrá dado por el máximo entre estos dos valores:
  - El doble de *NMAXMETARULES*.
  - El número de acciones existentes en las meta-reglas definidas por el usuario (el número de *regActNode* usados) más el doble del número de meta-reglas que aún se podrían usar (*NMAXMETARULES* menos el número de meta-reglas definidas por el usuario).

8.5. Conclusiones

En este capítulo se ha presentado el concepto de meta-regla ($\mu\mathcal{R}egla$) (junto con el de *foco de atención* como conjunto de $\mu\mathcal{R}eglas$), mecanismo fundamental para el incremento de la adaptatividad y de la eficiencia del agente. Se ha presentado dichos conceptos tanto a nivel abstracto como a nivel de usuario, y a nivel de sistema. Este último caso ha sido mostrado con mayor detalle para resaltar la forma en la que se ha logrado tener un almacén de $\mu\mathcal{R}eglas$ accesible desde Linux y RT-Linux que está preparado para gestionar el aprendizaje y olvido de estas $\mu\mathcal{R}eglas$ en cuanto se incorpore un algoritmo de aprendizaje adecuado.

Parte IV

Pruebas

9 Sistema de Ejecución y Pruebas

"No pretendo haber controlado los eventos, confieso,
por el contrario, que ellos me han controlado a mí."
Carta,4-IV-1864
– Abraham Lincoln, (1809-1865)

9.1. Introducción

Dado que se ha extendido el $\mathcal{AA}$, podría pensarse en realizar pruebas comparativas con y sin extensiones para demostrar la mejora conseguida o el incremento de la calidad de las soluciones o respuestas calculadas. Sin embargo, debido al carácter de las extensiones a la arquitectura de $\mathcal{AA}$ desarrolladas, no tiene sentido el realizar dichas pruebas de comparación con versiones previas de la arquitectura, pues estas extensiones amplian el ámbito de aplicación de la misma, así como flexibilizan la gestión del tiempo dentro de la misma. Es por ello, que aunque se presenta aquí alguna prueba mostrando cual ha sido el resultado con y sin las extensiones de meta-razonamiento desarrolladas, no se intenta realizar una demostración comparativa de la bon-

dad de las extensiones, pues no tiene sentido realizarla. El objetivo de las presentes pruebas es, por tanto, el reflejar el correcto funcionamiento de las extensiones realizadas.

Este capítulo presenta, en primer lugar, una descripción del sistema de ejecución utilizado, detallando no sólo el hardware y el software usado, sino también la forma en la que se ha conseguido una gestión uniforme del tiempo que posibilita la construcción de cronogramas para la depuración de los agentes. A continuación se muestran tres ejemplos de aplicación de la arquitectura de $\mathcal{AA}$. El primero de ellos, denominado *"ereMeta"*, es un *ejemplo de juguete* sin salida con el exterior que sirve para validar el funcionamiento de la implementación realizada, abstrayendo las pruebas de la variabilidad producida al tener que interactuar con un entorno variable, junto con la interfaz con ese entorno. El segundo de ellos, es un ejemplo que interactúa con un simulador de un proceso físico (tres tanques de aguas residuales junto con los grifos y válvulas adecuados para comunicarlos entre sí y suministrarlos de agua) y que se ha usado para mostrar el uso del concepto de *Grado de Reactividad*. El tercer y último ejemplo presentado, se usa para mostrar la forma de abordar un problema complejo, identificando los distintos comportamientos que debe presentar el agente para hacer frente a las diferentes situaciones, y diseñando las posibles transiciones entre comportamientos existentes.

9.2. Descripción del Sistema de Ejecución y sus Restricciones

9.2.1. Hardware/Software de Base

El modelo de ejecución del $\mathcal{AA}$ que se ha utilizado para implementar las extensiones propuestas está basado en:

- RT-Linux 3.2-pre1
- Linux kernel 2.4.18

sobre un ordenador Pentium III a 600 MHz con 128 Mb de RAM.

Hay que tener en cuenta que parte de la implementación del modelo de sistema de un $\mathcal{AA}$ supone modificaciones a los módulos a utilizar de RT-

Linux, ya que entre otras cosas no se utiliza el planificador de tareas de RT-Linux, sino uno propio (que como se ha mencionado previamente se ha denominado $\mathcal{FLS}$).

9.2.2. Sistema de Ejecución

Tal y como se ha descrito en el capítulo 3 existe una conversión entre los distintos modelos de abstracción de un $\mathcal{AA}$ que tiene como punto final un agente ejecutable. En esta sección se va a detallar los diversos módulos ejecutables de que se compone un $\mathcal{AA}$ y que se pueden observar en la figura 9.1.

Los módulos de los que se compone el sistema de ejecución de un $\mathcal{AA}$ se estructuran en dos conjuntos, los que corresponden a la capa refleja del agente y los que corresponden a la capa deliberativa en tiempo real del mismo.

9.2.2.1. Capa Refleja

Tal y como se describía en el capítulo 3, está formada por todos los módulos que gestionan las restricciones temporales críticas del agente, y que, por lo tanto, se cargarán como módulos de RT-Linux. Estos módulos se clasifican en tres grandes grupos o niveles: nivel del $\mathcal{FRTL}$, nivel del $\mathcal{RS}$ y nivel de aplicación.

Nivel del $\mathcal{FRTL}$: Este nivel está formado por los módulos que componen el sistema basado en RT-Linux que se utiliza como sistema operativo de tiempo real. Este conjunto comprende los módulos *CORE* y *HAL* que ofrecen los servicios y modelo de tareas necesarios para el resto del sistema de ejecución. Dentro de este grupo de módulos básicos se encuentra también *RT-TRACE* que ofrece la interfaz necesaria para establecer desde la capa refleja los *eventos de traza* que se utilizarán posteriormente en el modo de depuración.

Nivel del $\mathcal{RS}$: El conjunto de módulos que se encuentran en este nivel son el resto de módulos de RT-Linux que forman parte del agente y son independientes de la aplicación. Estos módulos forman la implementación del Servidor Reflejo (de ahí el nombre de este nivel), así como los

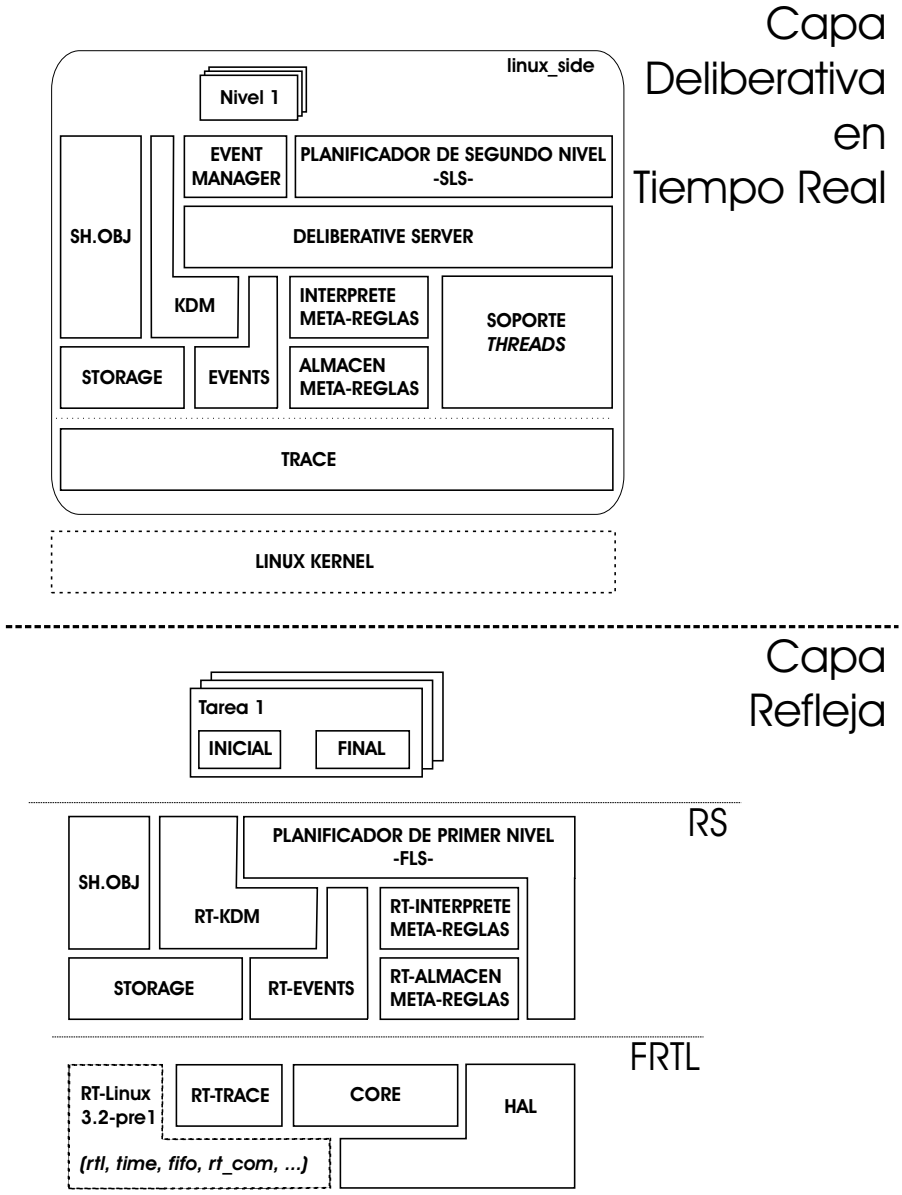

Figura 9.1: Estructura de módulos del sistema de ejecución de un $\mathcal{AA}$

módulos de gestión de memoria común a Linux y RT-Linux. De forma detallada, estos módulos son:

- *STORAGE*: Este módulo contiene la interfaz de funciones que permite crear, gestionar y liberar un trozo de memoria para que esté dis-

ponible tanto a tareas críticas del modelo de $\mathcal{FRTL}$, como a procesos Linux.

- *SHARED OBJECTS*: Ofrece una interfaz para crear *objetos protegidos*, es decir, zonas de memoria (de la ofrecida por el módulo *STORAGE*) de acceso en exclusión mutua.
- *RT-EVENTS*: Es el módulo que permite el acceso al almacén de eventos (que será otra zona de memoria común a las dos capas) desde esta capa del agente.
- *RT-KDM*: Utilizando la zona de memoria gestionada por el módulo *STORAGE* permite interactuar con las creencias del agente. Utiliza también el módulo *RT-EVENTS* para indicar cuando se genera un evento de los que están asociados a cambios en el $\mathcal{KDM}$ (según lo expuesto en la sección 6.2.1).
- *RT-ALMACEN META-REGLAS*: Este módulo gestiona otra zona de memoria común a las dos capas, que en esta ocasión contiene los elementos básicos de los que se componen las meta-reglas.
- *RT-INTERPRETE META-REGLAS*: Este módulo permite interpretar las meta-reglas que están almacenadas en la zona de memoria gestionada por el módulo *RT-ALMACEN META-REGLAS*.
- *PLANIFICADOR DE PRIMER NIVEL –FLS–*: Tal y como indica su nombre es el módulo que implementa el planificador de primer nivel, ofreciendo interfaces para la gestión de tareas críticas según el modelo de $\mathcal{FRTL}$. Utiliza los módulos *RT-EVENTS*, *RT-KDM* y *RT-INTERPRETE META-REGLAS*.

Nivel de Aplicación: Está formado por un único módulo que implementa las partes críticas (inicial y final) de todas las tareas del agente. Este módulo utiliza la interfaz de *RT-KDM* para el acceso a las creencias del agente y la del *PLANIFICADOR DE PRIMER NIVEL –FLS–* para la gestión de tareas.

9.2.2.2. Capa Deliberativa en Tiempo Real

Todos los módulos que componen la capa deliberativa en tiempo real de un $\mathcal{AA}$ se compilan de forma que compongan un único proceso denominado *linux_side* que se ejecuta a máxima prioridad sobre el *kernel de linux*. De

entre estos módulos, se pueden identificar primero aquellos que tienen su contrapartida en la capa refleja y que permiten el acceso a las mismas funcionalidades o zonas de memoria pero en distintas capas. La tabla siguiente recoge estas correspondencias:

Capa Refleja	Capa Deliberativa en Tiempo Real
RT-TRACE	TRACE
STORAGE	STORAGE
SHARED OBJECTS	SHARED OBJECTS
RT-EVENTS	EVENTS
RT-KDM	KDM
RT-ALMACEN META-REGLAS	ALMACEN META-REGLAS
RT-INTERPRETE META-REGLAS	INTERPRETE META-REGLAS

De forma similar a la organización de la capa refleja, los módulos que componen la capa deliberativa en tiempo real también se pueden clasificar en tres niveles:

Nivel del Operativo : Formado por aquellos módulos que complementan el *kernel* de Linux para dar el soporte básico de ejecución a esta capa. En este caso lo componen el módulo *TRACE* y el módulo *SOPORTE THREADS*. Este último módulo implementa la gestión de procesos Linux ligeros con las peculiaridades necesarias para la capa deliberativa en tiempo real del agente.

Nivel del $\mathcal{DS}$: Los módulos que forman este nivel, junto con los del nivel anterior, son independientes de la aplicación. Estos módulos componen la implementación del Servidor Deliberativo, junto a la gestión de memoria común a las dos capas. En concreto, estos módulos son: *STORAGE*, *SHARED OBJECTS*, mphEVENTS, *KDM*, *ALMACEN META-REGLAS*, e *INTERPRETE META-REGLAS*, que como se comentado ofrecen los mismos servicios que sus contrapartidas en la capa refleja, y los módulos correspondientes al *$\mathcal{DS}$ DELIBERATIVE SERVER*, *EVENT MANAGER* y *PLANIFICADOR DE SEGUNDO NIVEL –SLS–*.

Tal y como indica su nombre, el módulo *DELIBERATIVE SERVER* implementa el *Servidor Deliberativo*, para lo que comprende dos módulos, *EVENT MANAGER* y *PLANIFICADOR DE SEGUNDO NIVEL –SLS–*

que, tal y como se vio en la sección 7.4.1, corresponden a las dos partes del $\mathcal{DS}$. Este módulo utiliza los servicios ofrecidos por *EVENTS*, *KDM*, *INTERPRETE META-REGLAS* y *SOPORTE THREADS*.

Nivel de Aplicación : Es la parte dependiente de la aplicación de esta capa, y la forman los niveles no críticos. Cada uno de estos niveles es gestionado como un proceso ligero por el *PLANIFICADOR DE SEGUNDO NIVEL* por medio de los servicios ofrecidos por el módulo *SOPORTE THREADS*.

Cabe destacar que existen varios métodos de comunicación entre las dos capas que componen un $\mathcal{AA}$, cada uno de ellos utilizado para diferentes funciones:

- Comunicación Asíncrona: Es implementada por el mecanismo de eventos, y se utiliza para que el $\mathcal{RS}$ le proporcione al $\mathcal{DS}$ el conocimiento necesario para saber qué niveles opcionales de tareas críticas están activos.
- Comunicación Síncrona: En este caso se implementa por medio de dos colas de mensajes del tipo *RT-FIFO* (tal y como se definen en RT-Linux), y se utilizan para sincronizar la ejecución de las dos capas.

9.2.3. El Modo de Depuración

Para poder extraer información relativa a la ejecución del agente, y evaluar el funcionamiento de las mencionadas pruebas, se ha introducido en todos los módulos del modelo de sistema del $\mathcal{AA}$ un *Modo de Depuración*. Este modo se implementa, por un lado, mediante un mecanismo de extracción de *eventos de traza* en Linux y RT-Linux, y, por otra parte, mediante la instrumentación de buena parte de los módulos software de la arquitectura, de forma que la ejecución en modo depuración genera una traza que después se puede procesar para analizar la ejecución del $\mathcal{AA}$.

Así, en este modo, un prototipo de $\mathcal{AA}$ se va a ejecutar mediante un *script* de Linux denominado *rtload* que recibe en su invocación dos parámetros: el tiempo que se quiere que se ejecute el prototipo (en segundos) y el nombre genérico para todos los archivos de traza a generar.

Tal y como se mencionó en el apartado 2.2.1.1, la *continuidad temporal* es uno de los atributos más importantes de un agente, y por descontado, uno de los atributos del $\mathcal{AA}$. Sin embargo, en su ejecución en modo depuración, se requiere que dicha ejecución esté limitada en el tiempo para poder consultar la traza de dicha ejecución y poder depurar el agente.

De esta manera, mediante el *script* mencionado, desde Linux se activa el prototipo de $\mathcal{AA}$, insertando los módulos de RT-Linux necesarios, y poniendo en marcha la parte correspondiente de Linux, a la vez que se programa una señal para que después del tiempo indicado se finalize la ejecución (extrayendo los módulos de RT-Linux, y esperando a que acabe la parte de Linux correspondiente).

Durante la ejecución del prototipo de $\mathcal{AA}$, la parte Linux del mismo va escribiendo su traza en dos ficheros Linux, uno para los eventos de traza que acabarán en la traza *kiwi*, y otro para mensajes genéricos de depuración que el programador haya querido indicar. Por otro lado, la parte RT-Linux escribirá los eventos de traza en una zona de memoria común (del tipo *RT-FIFO*), cuando se detiene el prototipo, se activa un proceso Linux que vuelcan toda la información de la mencionada *RT-FIFO* en un fichero de traza *kiwi*. Al finalizar, se fusionan los dos ficheros de traza *kiwi*, para obtener un fichero *kiwi* con el nombre indicado como segundo parámetro de la invocación del *rtload*. Cualquier mensaje que el programador quisiese indicar en el modo depuración de la parte RT-Linux se mostrará por pantalla.

Esta herramienta de depuración aquí explicada ha demostrado ser valiosísima a la hora de depurar el código en la fase de desarrollo, e imprescindible para poder extraer la información que se presenta en este capítulo.

9.2.3.1. La Gestión Uniforme del Tiempo

Uno de los problemas planteados a la hora de establecer un modo de depuración del $\mathcal{AA}$ es que la noción del tiempo en Linux y RT-Linux es distinta, pues utilizan distintas precisiones (Linux trabaja en microsegundos, mientras que RT-Linux lo hace en nanosegundos) e incluso los orígenes de tiempo son distintos. Es por esto, que a la hora de construir el modo de depuración se hacía necesario unificar la línea temporal entre ambos mundos para poder seguir el orden en el que se producían las operaciones en dicha depuración.

Para ello se optó por marcar todos los *eventos de traza* en el momento en

que se generan mediante la instrucción en ensamblador *RDTSC* (*ReaD-Time Stamp Counter*) [Intel 1997]. Esta instrucción consulta el valor del *Time Stamp Counter* (*TSC*), que es un registro de 64 bits[1] (disponible desde el Pentium®) cuyo valor se va incrementando a cada señal del reloj hardware desde el arranque de la máquina, con lo que es un contador de ciclos de reloj de la CPU que funciona a la velocidad del propio reloj interno de la CPU. Así, para transformar esta cantidad de ciclos a unidades de tiempo, hay que tener en cuenta la velocidad del procesador utilizado, ya que, por ejemplo, dos cientos millones de ciclos en un procesador a 200MHz es equivalente a un segundo de tiempo real, mientras que ese mismo número de ciclos en un procesador a 400 MHz corresponde sólo a medio segundo de tiempo real. Para realizar esta conversión, se divide el número de ciclos por la frecuencia (en Hz).

Precisamente, la transformación del número de ciclos almacenados en TSC a unidades de tiempo es el mecanismo que utiliza RT-Linux para medir el tiempo, con lo que el origen de tiempo que se tiene es el arranque del ordenador y la unidad mínima de medida es el nanosegundo. Por otro lado, en Linux [2], la unidad mínima de medida es el microsegundo y el origen de tiempos es el denominado *Epoch* , 00:00:00 horas del 1 de enero de 1970.

9.2.3.2. Un *Cronograma* como salida del modo depuración

Una ejecución del $\mathcal{AA}$ en modo depuración genera dos ficheros de traza que se convierten en un *cronograma* que describe la ejecución del modelo de sistema del agente. Este cronograma se visualiza por medio de la herramienta *kiwi*[3], y produce la visualización que se presenta en los siguientes ejemplos. En la figura 9.2 se puede observar el significado de gran parte de los símbolos que aparecen en este tipo de gráficas.

De esta manera, en la figura 9.3 se puede ver un detalle de una representación en *kiwi* de un cronograma. En uno de estos cronogramas, cada tarea va a venir representada por una posición dentro de las coordenadas verticales del cronograma, indicando además la posición la prioridad de esa tarea con respecto a las demás, estando la más prioritaria en la parte superior del

[1] La instrucción *RDTSC* carga los 32 bits más altos del registro en *EDX*, y los 32 más bajos en *EAX*.

[2] Como puede consultarse en las páginas de *man* de Linux, gettimeofday(2) y time(2).

[3] Herramienta para visualizar trazas, desarrollada en Tcl/Tk por el Doctor Agustín Espinosa y que se encuentra disponible en *http://rtportal.upv.es/rtportal/apps/kiwi/*

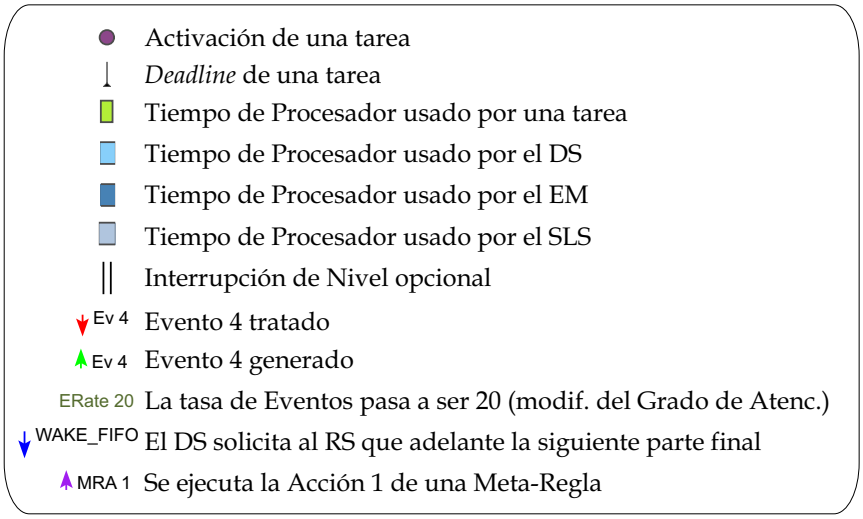

Figura 9.2: Leyendas de los símbolos usados en *kiwi*

cronograma y la menos en la parte inferior. Así, en todos los cronogramas aparecerán, de arriba hacia abajo las siguientes partes:

- En la parte superior, como una tarea más aunque no lo sea, y etiquetado como *Kernel* el $\mathcal{RS}$.
- A continuación, y siempre en estricto orden de prioridad descendente, aparecerán las tareas críticas.
- Después aparecerá una línea etiquetada como *Linux* que representa la parte de Linux que no pertenecen ni al $\mathcal{DS}$ ni a ninguna de las tareas no críticas del agente.
- La siguiente línea es la correspondiente al $\mathcal{DS}$, donde además si existe meta-razonamiento, se distinguirá entre el $\mathcal{EM}$ y el $\mathcal{SLS}$.
- La última parte del cronograma la componen las líneas que corresponden a las tareas opcionales del $\mathcal{AA}$, ordenadas, claro está, por prioridad.

Para acabar de ver la forma en la que se representan las tareas en un cronograma, basta indicar las partes que se han resaltado de la figura 9.3:

1. Activación de la tarea *In_Agent_B*. A partir de este instante, y hasta el *deadline*, esta tarea puede ejecutarse.

2. *Deadline* de la tarea *In_Agent_B*.

3. Ejecución de la tarea *In_Agent_C*. Estos rectángulos indicarán el tiempo en el que se estén ejecutando las diferentes tareas.

4. La ejecución de la tarea opcional *T8* es interrumpida. Posteriormente, si el $\mathcal{SLS}$ lo estima oportuno podrá reanudarse su ejecución si hay tiempo disponible antes de que se cumpla su *deadline*)

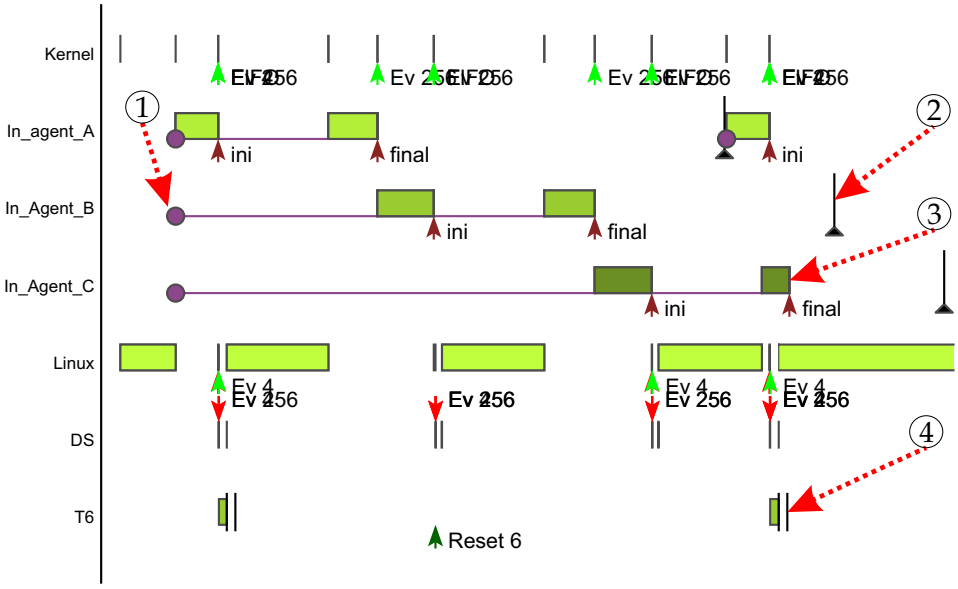

Figura 9.3: Ejemplo de cronograma con kiwi

Tal y como se ha mencionado previamente, hay que tener en cuenta que, dentro del Módulo de Control, $\mathcal{RS}$ y $\mathcal{DS}$ van a trabajar con magnitudes temporales distintas, debido a que RT-Linux (y, por tanto, el $\mathcal{RS}$) trabaja en nanosegundos, mientras que Linux (y, de esta manera, el $\mathcal{DS}$) trabaja en microsegundos. Así, cada vez que se diseña un ejemplo todas las magnitudes temporales que se le definen se hacen en microsegundos, aunque internamente se les aplicará el factor corrector correspondiente a aquellas que deban pasarse al $\mathcal{RS}$.

9.2.4. Restricciones

En algunos casos, la implementación de las extensiones propuestas en el sistema de ejecución con el que se han desarrollado las pruebas es ligeramente distinta de la definición teórica, tal como se ha planteado en capítulos anteriores. El motivo de estos cambios es, en todos los casos, la *falta de funcionalidades* en otros módulos del $\mathcal{AA}$ que quedan fuera del ámbito de este trabajo.

La arquitectura de $\mathcal{AA}$ es un trabajo en desarrollo, y por tanto hay que tener en cuenta que durante la realización del presente trabajo, e incluso después de finalizado el mismo, se están desarrollando nuevos componentes, depurando los existentes y/o extendiéndolos.

En concreto, el más importante de los cambios es que debido a que en la implementación actual existe un único mútex para la protección del acceso a los datos del $\mathcal{KDM}$, el protocolo de cambio de modo implementado ha pasado a ser un protocolo síncrono en lugar del asíncrono que se planteaba. Protocolo de cambio de modo síncrono, por limitaciones en la protección de datos del $\mathcal{KDM}$ Esto es así, porque al tener un único mútex para proteger el acceso a la memoria, cualquier tarea del modo previo puede interferir en la ejecución de cualquier tarea del modo nuevo, con lo cual se debe retrasar la activación de todas las tareas del modo nuevo (que no estuviesen ya en el modo previo) hasta que acaben las tareas del modo anterior, o lo que es lo mismo, se utiliza un cambio de modo síncrono.

En los ejemplos planteados, se han realizado pruebas de ejecución con y sin el proceso de meta-razonamiento, teniendo en cuenta que para que el $\mathcal{AA}$ funcione de forma similar en ambos casos, si existen varios comportamientos, al ejecutarlo sin meta-razonamiento deben estar siempre activos todos los *in-agents* de todos los comportamientos.

A la hora de hacer estas pruebas, se ha utilizado la nomenclatura apropiada a cada prueba. Así, cabe recordar que el Módulo de Control se divide a su vez en dos módulos, y que estos se llaman $\mathcal{RTOS}$ e $\mathcal{IS}$ si no se usa meta-razonamiento (como se define en el punto 3.4.2.4 de este documento), o $\mathcal{RS}$ y $\mathcal{DS}$ si sí que se incluyen las capacidades de meta-razonamiento (definido en este caso en el capítulo 7).

9.3. El *"ereMeta"*: Un ejemplo abstracto

9.3.1. Descripción del Problema

Uno de los principales problemas encontrados en la fase de implementación y pruebas del presente trabajo es la integración de módulos software desarrollados por múltiples miembros del grupo de investigación, sumado al hecho de que las aplicaciones a las que típicamente se aplica la arquitectura de $\mathcal{AA}$ están orientadas a controlar un proceso físico (robot, planta de depuración de agua, ...), donde a la complejidad típica de integrar un software en desarrollo hay que añadirle el control real de un hardware específico.

Por ello se decidió que como primer banco de pruebas de la implementación planteada en este documento, se utilizase el ejemplo abstracto detallado a continuación con el único objetivo de validar la implementación independizándola de los dispositivos de entrada/salida utilizados.

9.3.2. Diseño del $\mathcal{AA}$

El $\mathcal{AA}$ está compuesto por los comportamientos recogidos en la siguiente tabla, donde se indican los *in-agents* críticos que los componen, con sus características principales:

Comp.	*in-agents*	*Deadline*	Período	Opcionales	*WCET* Ini	*WCET* Fin
1	1	900000	1000000	T9 y T10	51	0
	2	1900000	2000000	T11	1	0
2	3	350000	350000	—	10100	1000
	4	340000	340000	—	20100	1000
	5	490000	490000	T12	1000	1000
	6	500000	500000	T13	1000	1000

WCET Ini y *WCET Fin* corresponden a los tiempos de ejecución en el peor caso de las partes inicial y final de los distintos *in-agents*.

Utilizando las características temporales de las tareas críticas del modelo de sistema del $\mathcal{AA}$ se analizaría la planificabilidad de los distintos comportamientos del $\mathcal{AA}$.

Cabe destacar en este ejemplo que la parte opcional del *in-agent* 2 la compone un nivel en *CLIPS* (T11), mientras que el resto de operaciones que realizan las diferentes partes de las tareas que conforman los *in-agents* de los comportamientos del *ereMeta* son bucles de cálculo con diferentes accesos al $\mathcal{KDM}$, ya que el único objetivo es que consuman tiempo.

9.3.3. Ejemplo de Ejecución

En este caso se ha aprovechado un ejemplo *vacío* para mostrar la diferencia en la ejecución $\mathcal{AA}$ con y sin meta-razonamiento. Así, en la figura 9.4 se puede ver un detalle de un cronograma fruto de ejecutar el presente ejemplo sin Meta-Razonamiento, mientras que en la figura 9.5 se presenta un detalle del cronograma correspondiente a la ejecución del mismo ejemplo pero, en este caso, activando el Meta-Razonamiento.

Al ejecutar el *"ereMeta"* sin meta-razonamiento, se deben incluir todas las tareas correspondientes a todos los *in-agents* críticos del $\mathcal{AA}$, por lo que se puede observar como están activas las 6 tareas en la figura 9.4. Además, también se pueden apreciar algunas de las deficiencias que se han superado con las extensiones planteadas en este documento:

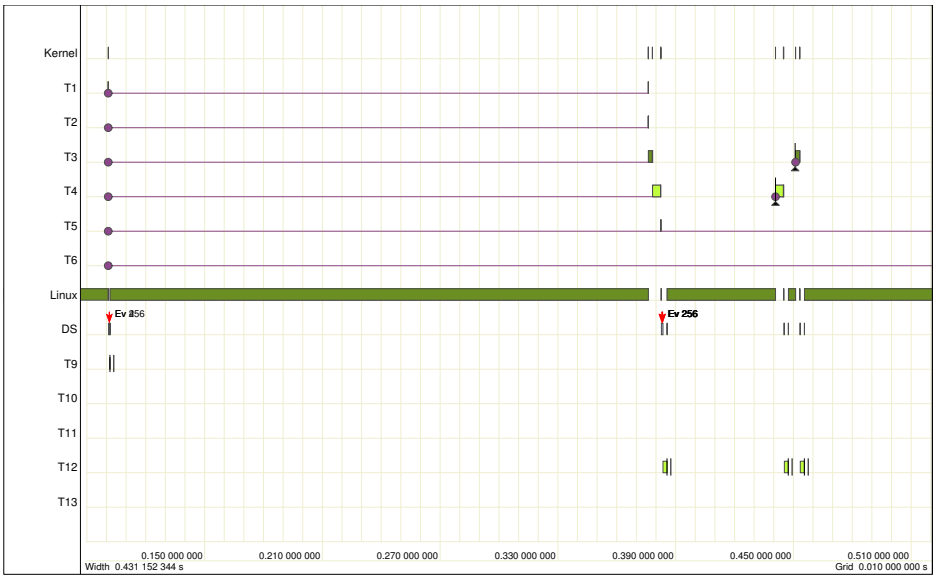

Figura 9.4: Ejecución del *ereMeta* sin Meta-Razonamiento

- La falta de comunicación entre las dos partes del Módulo de Control (en el gráfico etiquetados como Kernel y $\mathcal{DS}$, que corresponderían al $\mathcal{RTOS}$ y al $\mathcal{IS}$ respectivamente) hace que cuando el $\mathcal{IS}$ decide no hacer nada más en su primera ejecución se tenga que consumir sin embargo todo el *slack* u holgura disponible hasta que se decida ejecutar la parte final de la primera tarea (la correspondiente al primer *in-agent*). Este consumo innecesario de holgura hace que el resto de tareas no puedan utilizarla, con lo que por ejemplo, la tarea T11 (parte opcional del segundo *in-agent*) no llega a lanzarse a ejecución.

- En este caso, al tener mal calculados los tiempos de ejecución en el peor caso de las tareas opcionales, éstas no logran acabar su ejecución con el tiempo disponible, y el $\mathcal{IS}$ no es capaz de reanudar su ejecución, con lo que todas las partes opcionales que se tratan de ejecutar, no sirven para nada.

- Se interrumpe la ejecución de T9 (parte opcional del *in-agent* correspondiente a la tarea T1) y no se reanuda su ejecución, ni se trata de ejecutar T10 (la otra parte opcional de dicho *in-agent*), debido a la mala gestión del tiempo del $\mathcal{IS}$.

Esto no ocurre, sin embargo, en la ejecución de la figura 9.5, ya que al usar Meta-Razonamiento se palían las deficiencias comentadas en el caso anterior:

- Como se ha comentado al describir el ejemplo, existen dos comportamientos, lo que hace que sólo se dedique tiempo a las tareas del comportamiento actual, no desperdiciando tiempo en tareas que no tienen ningún interés en la situación actual. En la figura 9.5 se puede observar además, como se produce un *cambio de comportamiento* (marcado con una línea vertical paralela al eje de ordenadas del gráfico).

- Cuando el $\mathcal{DS}$ decide no hacer nada, se lo comunica al $\mathcal{RS}$, y éste puede adelantar la ejecución de la siguiente parte de una tarea, siempre que ésta sea una parte final. Éste es el caso que se puede ver en la ejecución de la tarea T1 de dicho cronograma, y se observa como al adelantar la ejecución de dicha parte final, se reaprovecha la holgura no utilizada en dicha parte para la ejecución de partes opcionales de otras tareas, dando cabida a la ejecución de la tarea T11.

- Tal y como se puede observar en el zoom de la figura 9.5 se ha mejorado considerablemente la gestión del tiempo por parte del $\mathcal{DS}$, con lo que es capaz de reanudar la tarea T9 después de haberla interrumpido.

Figura 9.5: Ejecución del *ereMeta* con Meta-Razonamiento

- Al mejorar la comentada gestión del tiempo, no sólo es capaz de reanudar tareas interrumpidas, sino que incluso tiene tiempo de ejecutar más tareas, como, en el ejemplo, la tarea T10 (segunda parte opcional de la tarea T1), todo ello intercalado con ejecuciones del $\mathcal{EM}$ que da respuesta a los distintos eventos que se van generando.

9.3.4. Conclusiones del Ejemplo

El objetivo del presente ejemplo era mostrar (que no demostrar) la potencia de las extensiones planteadas. Así, se han dejado patentes las deficiencias del modelo previo a este trabajo, y se han mostrado algunas de las mejoras conseguidas con las extensiones realizadas:

- La gestión de múltiples comportamientos permite centrar el proceso de razonamiento, y por tanto el uso del tiempo de procesador, tan sólo en las tareas que resultan de interés a la situación actual.
- La comunicación entre las dos partes que conforman el Módulo de Control permite que no se *desperdicie* la holgura existente, permitiéndole al $\mathcal{DS}$ decirle al $\mathcal{RS}$ que no tiene más trabajo para realizar, y, por tanto, que adelante, si es posible, tareas a ejecutar, para así poder recuperar posteriormente parte de esa holgura.
- La mejora de la gestión del tiempo por parte del $\mathcal{DS}$ hace incluso posible el funcionamiento con datos incorrectos como se ha visto en el ejemplo en el que los tiempos de ejecución en el peor caso estaban mal estimados.

9.4. Tanques de Aguas Residuales

9.4.1. Descripción del Problema

El objetivo del agente en este ejemplo es controlar una serie de depósitos de agua residual interconectados entre sí y con diferentes entradas y salidas de flujos de agua (ver figura 9.6). El $\mathcal{AA}$ debe controlar que el nivel de líquido en estos depósitos se mantenga en unas referencias que el usuario puede variar de forma dinámica.

Como dato a tener en cuenta, el sistema a controlar compuesto por los tanques comentados estará simulado en un ordenador conectado vía puerto serie al del $\mathcal{AA}$.

El programa utilizado para realizar la simulación es *LabView* 7[4].

De forma más concreta, se deben controlar tres depósitos de agua, A, B y C, con una capacidad de 5000 litros los dos primeros y de 10000 el tercero. Para poder controlar el nivel de los depósitos, se dispone de cinco sensores en cada depósito situados cada 20 % del volumen del depósito (figura 9.6).

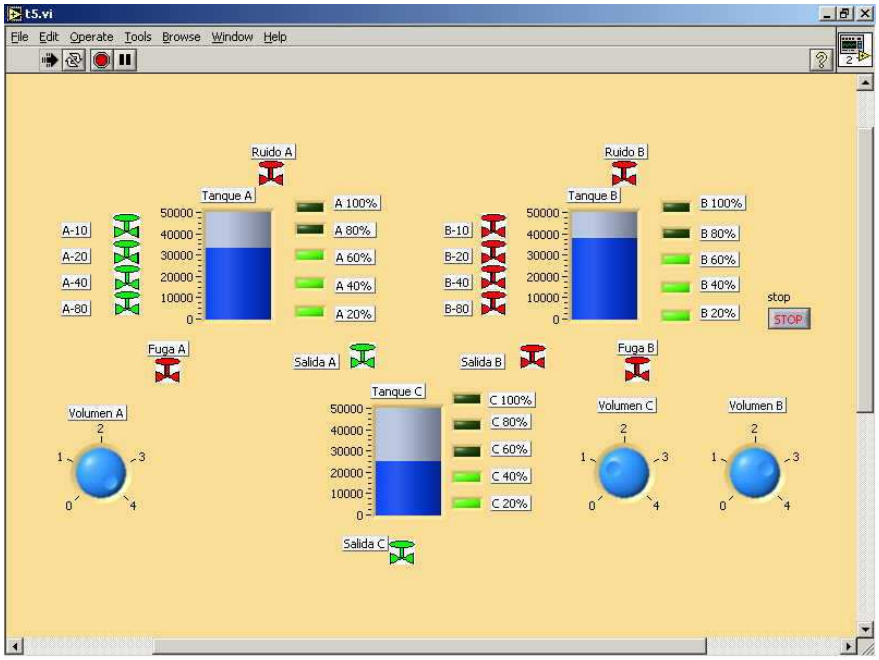

Figura 9.6: Depósitos simulados en *LabView* 7

Las entradas de agua al sistema vienen dadas por 4 grifos con caudales de 10, 20, 40 y 80 litros por segundo, respectivamente. Cada uno de estos grifos puede tan sólo ser abierto o cerrado, pero no se puede regular el caudal que envía, siendo éste, por tanto constante (es decir son grifos *todo/nada*).

Los depósitos A y B disponen de una salida controlada con una válvula que les permite vaciar su contenido sobre el depósito C de forma controlada

[4]Programa generador de instrumentación virtual desarrollado por *National Instruments* que permite la simulación de procesos físicos y su control en tiempo real

(por medio de dichas válvulas). El caudal de estas salidas es de 95 litros por segundo cada una. Por su parte, para poder vaciar el depósito C se dispone de una salida controlada por una válvula con un caudal de 180 litros por segundo. Todas estas válvulas también son del tipo *todo/nada*).

Para introducir perturbaciones en el sistema se han incluido una serie de entradas y salidas adicionales que no serán controladas automáticamente, sino que servirán para modificar de forma manual la cantidad de líquido que entra a un depósito o la que sale. En concreto, los depósitos A y B disponen de un grifo de entrada, o ruido, cada uno de caudal 20 litros segundo y un grifo de salida, o fuga, de caudal 10 litros segundo. El agua procedente de la fuga no va a parar el depósito C como ocurría con los controlados automáticamente. Además posee unos controles que sirven para fijar el intervalo deseado de volumen almacenado. Estos controles están representados por mandos giratorios para cada depósito con el siguiente significado:

Posición	Nivel Mínimo	Nivel Máximo
0	0 %	20 %
1	20 %	40 %
2	40 %	60 %
3	60 %	80 %
4	80 %	100 %

9.4.2. Diseño del $\mathcal{AA}$

El $\mathcal{AA}$ diseñado para controlar todo el sistema está formado por un único comportamiento con tres *in-agents*, uno por cada uno de los depósitos. Debido a esto, los *in-agents* encargados de los depósitos A y B serán idénticos.

Cada *in-agent* está formado por tres partes:

1. MKS de sensorización y se encarga de leer los sensores que indican el nivel de llenado de su depósito respectivo. También se encarga de leer los pulsadores de elección de nivel deseado, tanto los que se encuentran en el panel de control, como los de emergencia en el propio simulador. Este MKS dispone de un solo nivel, en este caso nivel 0 al ser un MKS crítico.

2. MKS para calcular las acciones a realizar (apertura o cierre de los grifos correspondientes). Este MKS crítico dispone de tres niveles, (el nivel 0 crítico y 2 niveles opcionales). En el caso del nivel 0 el algoritmo calcula una respuesta rápida aunque de baja calidad:

   - Si el volumen almacenado está por debajo de lo solicitado, abrir todos los grifos de entrada y cerrar la válvula de salida del depósito correspondiente
   - Si dicho volumen está por encima, cerrar todos los grifos de entrada y abrir la válvula de salida.

3. Un KS de acción para llevar a cabo las acciones calculadas por el MKS anterior, es decir, mandar las acciones oportunas al simulador para abrir o cerrar grifos y válvulas.

Las características temporales principales de los *in-agents* críticos de este $\mathcal{AA}$ son las siguientes:

in-agents	*Deadline*	Período	Opcionales	*WCET* Ini	*WCET* Fin
In_Agent_A	2500000	2500000	T6 – T9	200000	200000
In_Agent_B	2700000	2700000	T10 – T13	200000	200000
In_Agent_C	2900000	2900000	T14 – T17	200000	200000

En este ejemplo se puede observar una prueba de la posibilidad de gestionar de forma más flexible el tiempo del que dispone un $\mathcal{AA}$ pues se ha incorporado una $\mu\mathcal{R}egla$ para controlar que al alcanzar el máximo nivel en un depósito se varíe el *Grado de Reactividad* para que pase a valer 0. Esto consigue que *no* se ejecuten más partes opcionales y se avance la ejecución de las partes finales de las tareas, es decir, que se actúe lo antes posible para evitar/paliar el desbordamiento de los depósitos.

9.4.3. Ejemplo de Ejecución

A continuación se muestran algunos cronogramas fruto de ejecutar el ejemplo anterior, a partir de los cuales se pueden observar algunas de las mejoras obtenidas con la extensión de Meta-Razonamiento que se ha desarrollado.

Para ello, en la figura 9.7 se puede observar el cronograma fruto de la ejecución del ejemplo anterior sin la utilización de las extensiones de Meta-Razonamiento.

Una de las deficiencias que se pueden observar de esta ejecución, es que, al igual que pasaba en el ejemplo del *"ereMeta"*, la parte deliberativa del $\mathcal{AA}$ no es lo suficientemente flexible como para sobreponerse a una mala estimación del tiempo de ejecución en el peor caso. Hay que tener en cuenta que, como ya se ha comentado, los planificadores del $\mathcal{AA}$ trabajan con tiempos de ejecución en el peor caso, pero en la parte no crítica habrá ocasiones en la que estos tiempos o no están lo suficientemente bien calculados o no es posible calcularlos (por ejecutar un código que no esté acotado).

En estos casos, como ocurre en la ejecución de los niveles opcionales T6, T10 y T14, se puede observar como el planificador los lanza a ejecución y dichos niveles se interrumpen al consumir el tiempo que se les ha dedicado. Por la falta de flexibilidad del antiguo $\mathcal{DS}$, no es capaz de volver a ejecutar dichos niveles, pues en el intervalo de ejecución en el que se interrumpen ya no ejecuta nada. La siguiente vez que entra en ejecución, como puede observarse, se ha ejecutado la parte final de la tarea a la que pertenecen y por tanto se eliminan de las correspondientes agendas (como aparece marcado en el cronograma con *Reset*).

En la figura 9.8 se puede ver un detalle de la ejecución del mismo ejemplo con la misma duración que en el caso anterior, pero incluyendo esta vez las extensiones de Meta-Razonamiento. Una de las primeras diferencias que podemos observar con respecto al cronograma anterior es el aprovechamiento del tiempo de holgura en este caso, pues el $\mathcal{DS}$ se ejecuta durante más tiempo, aprovechando dicho tiempo para ejecutar las partes opcionales oportunas, concediéndoles más tiempo del que en principio deberían tener para su ejecución, siempre que haya tiempo disponible. Esto permite la ejecución de tareas opcionales cuyo tiempo de ejecución en el peor caso estuviese mal estimado. Así, se puede observar, por ejemplo, como el nivel opcional T6 consigue finalizar su ejecución, pasando a ejecutarse el nivel T7 (opcional también de la primera tarea, que corresponde al *in-agent* A).

Otro aspecto importante a observar en el cronograma de la figura 9.8 es un cambio total del *Grado de Reactividad* del agente. Así, durante la ejecución de las partes opcionales de la primera tarea, el agente detecta una situación de emergencia (el nivel de un tanque ha sobrepasado el límite máximo per-

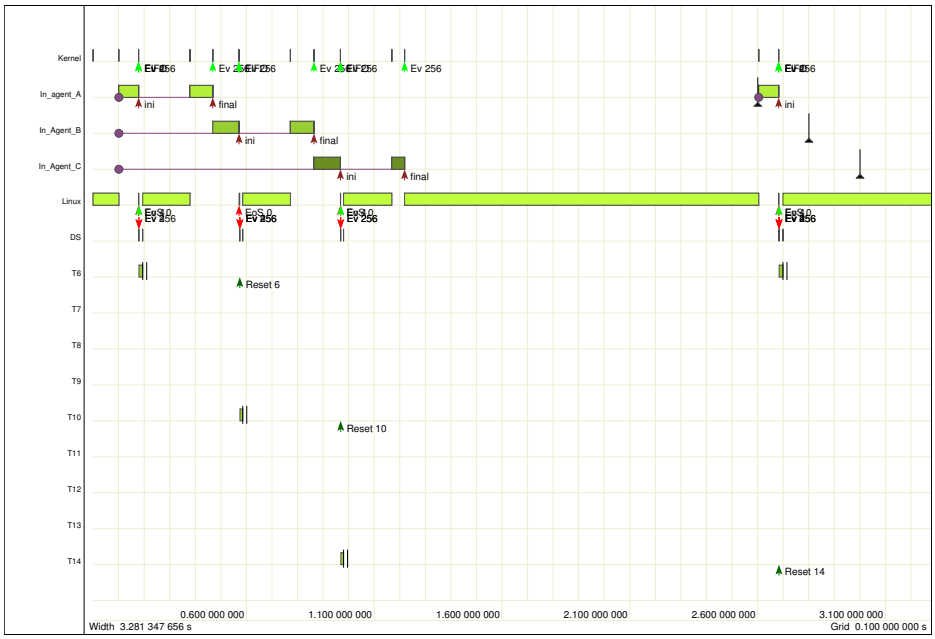

Figura 9.7: Ejecución de 3.1 segundos sin Meta-Razonamiento

mitido), con lo que se reduce el *Grado de Reactividad* a 0, para que actúe de inmediato (abriendo las válvulas de salida y cerrando los grifos correspondientes). Como se puede observar, esto hace que en el resto de la ejecución que se puede observar no se ejecuten más partes opcionales de las tareas, pues se ejecutan sus partes finales a continuación de las partes iniciales de las mismas.

En la figura 9.9 se puede ver el mismo ejemplo de ejecución que los anteriores con la misma duración, pero donde el *Grado de Reactividad* pasa a ser del 50 % en lugar de 0, con lo que se puede ver como varía la ejecución del resto de tareas, así como del $\mathcal{DS}$ y de los niveles que elige para su ejecución.

Por último, en la figura 9.10 se puede observar un detalle del funcionamiento del módulo $\mathcal{DS}$ (en concreto corresponde a la ejecución del $\mathcal{DS}$ después de la primera de la parte inicial del *In_Agent_A* en la ejecución de la figura 9.9), donde primero se ejecuta el $\mathcal{EM}$ que responde primero a un evento *CONSIDER_PERIODIC_AGENT* y después a un *MODIFICATION*. La respuesta de este último es la ejecución de una $\mu\mathcal{R}egla$ que cambia la política de planificación del $\mathcal{SLS}$ y el *Grado de Reactividad* (pasando a ser 50, como se ha comentado). Por último, antes de entrar en ejecución el $\mathcal{SLS}$, se cambia el *Grado de Introspección* del $\mathcal{EM}$ para que pase a atender como máximo a 10

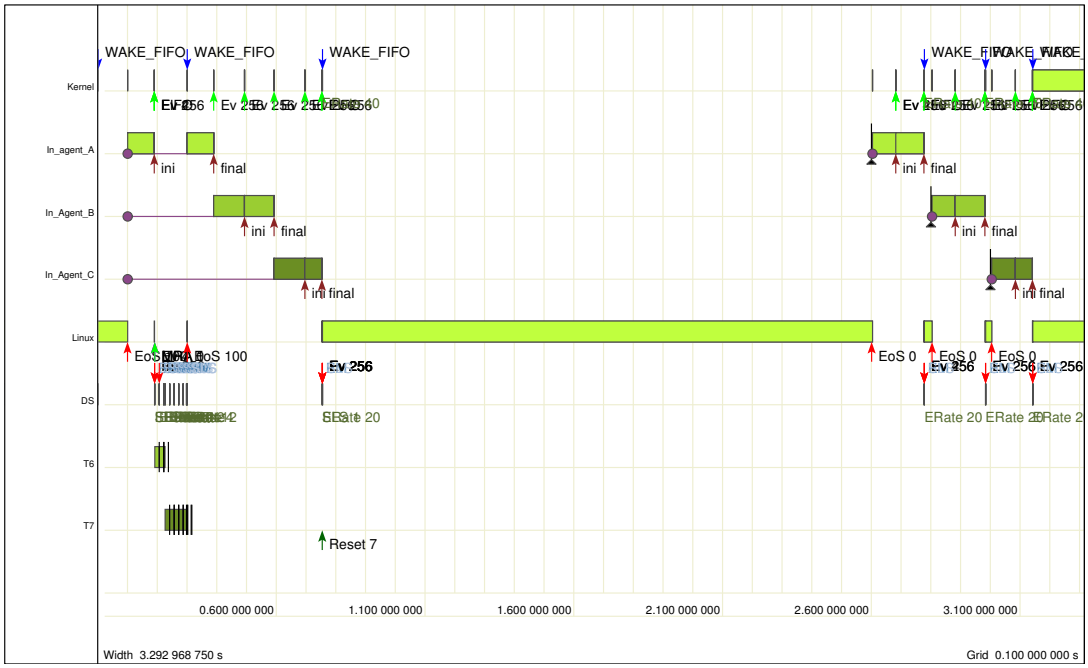

Figura 9.8: Ejecución 1 de 4 segundos con Meta-Razonamiento

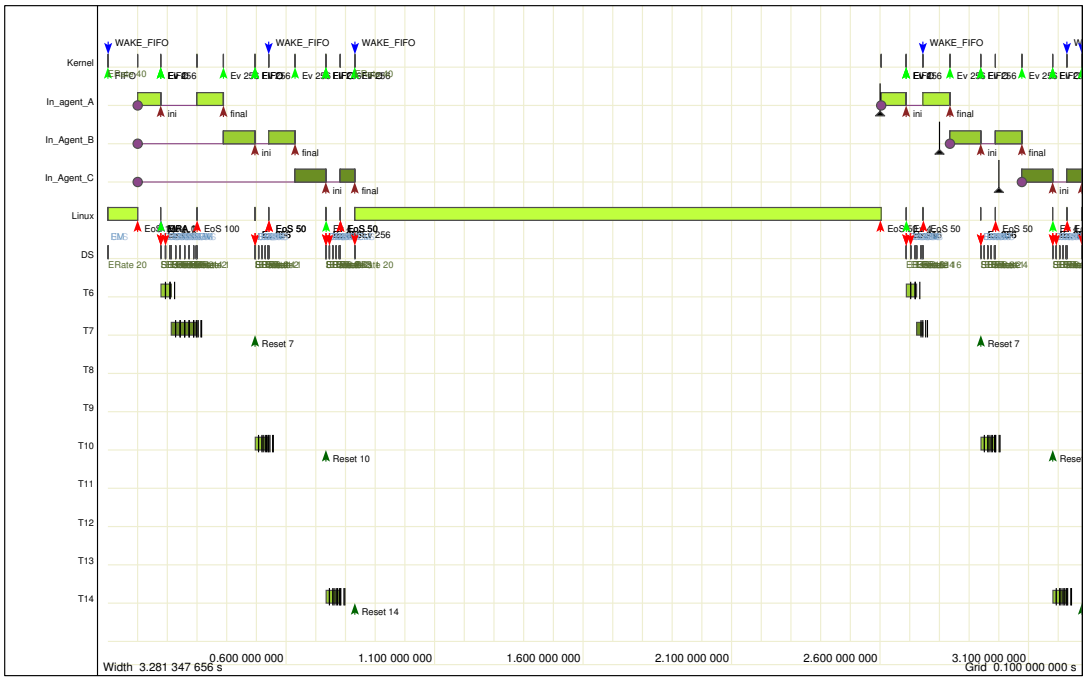

Figura 9.9: Ejecución 2 de 4 segundos con Meta-Razonamiento

eventos en su siguiente ejecución, en lugar de los 20 que tenía como máximo. Este cambio es debido a que tan sólo había 2 eventos por tratar, con lo que trata de ajustar también la tasa de eventos máxima que supone el *Grado de Introspección* para que se ajuste más a un número cercano a la cantidad de eventos que se maneja. Se intenta ajustar este valor porque se utiliza para indicar cuanto tiempo se dedica al $\mathcal{EM}$ y cuanto se le deja al $\mathcal{SLS}$.

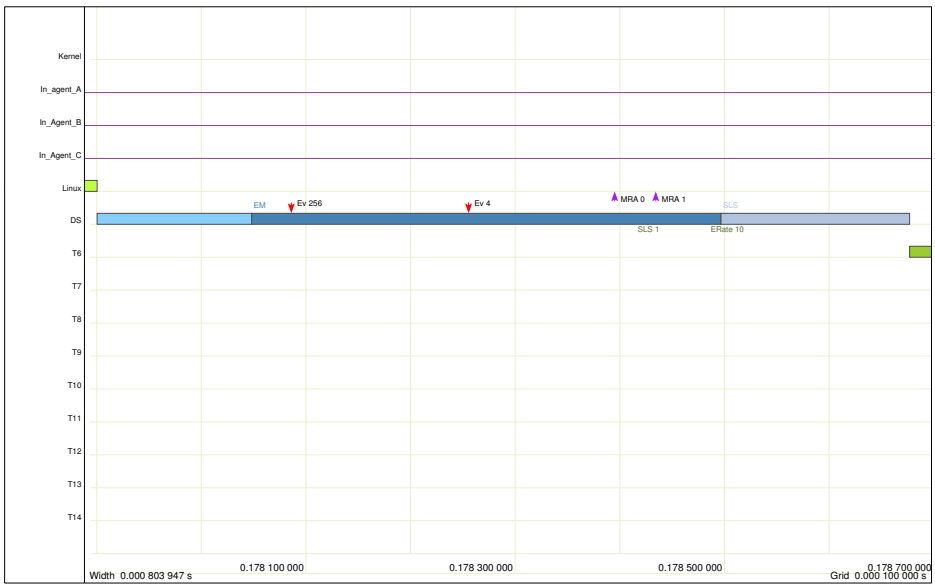

Figura 9.10: Ejecución 2 de 4 segundos con Meta-Razonamiento (detalle)

9.4.4. Conclusiones del Ejemplo

Además de volver a incidir en las limitaciones del $\mathcal{AA}$ sin Meta-Razonamiento que se habían comentado y mostrado ya en el ejemplo anterior, este ejemplo sirve para mostrar la potencia que le aporta al $\mathcal{AA}$ el poder tener un *Grado de Reactividad* variable, que permite ajustar el tiempo dedicado a deliberar a la situación actual por medio también del uso de $\mu\mathcal{R}eglas$.

En este caso, no tiene sentido comparar la ejecución sin Meta-Razonamiento con la ejecución con dichas extensiones, pues al incluirlas el $\mathcal{AA}$ consigue enfrentarse a algunas situaciones en una posición más ventajosa que la que se tenía antes, tal y como se ha visto en el ejemplo en el que el $\mathcal{AA}$ puede enfrentarse a una situación de emergencia haciendo que el *Grado de Reactivi-*

dad sea 0, y consiguiendo así que la respuesta del agente sea lo más rápida posible.

9.5. Meta-Cartero Robot

9.5.1. Descripción del Problema

El objetivo del presente ejemplo es desarrollar un $\mathcal{AA}$ que nos proporcione el comportamiento inteligente de un robot móvil cuya función es transportar y repartir el correo de una oficina a otra del mismo piso [Soler 2000]. Para hacer esto, el robot recibe peticiones via radio *ethernet*, incluyendo las localizaciones inicial y final de los objetos a mover. El robot puede recibir peticiones en cualquier momento y, de acuerdo a estas peticiones y a su localización actual, debería planificar el orden en el que debe servir dichas peticiones. El proceso de planificación se basa en las prioridades definidas por el usuario junto con el objetivo de minimizar el tiempo de reparto.

Este $\mathcal{AA}$ ha sido desarrollado para el robot móvil *Mobile Pioneer 2* (ver figura 9.11). El robot contiene todos los componentes básicos para la sensorización y navegación en el entorno del mundo real, incluyendo el nivel de la batería, controlar los motores y las ruedas, los codificadores de posición y de velocidad así como un conjunto de sensores y accesorios integrados. Todos estos componentes son gestionados por un micro-controlador junto con un *software* propio.

El robot tiene un anillo de 16 sonares. Las posiciones de estos sonares son fijas estado dispuesto en dos tiras (delantera y trasera): uno en cada lado y seis dispuesto en intervalos de cada 20 grados.

El servidor *software* tiene un mecanismo de localización interna que proporciona una estimación de su localización. Toda esta información se envía a través del puerto serie RS-232. El ordenador de a bordo tiene una radio *ethernet* que permite la conexión para los envíos de peticiones.

Figura 9.11: *Mobile Pioneer 2* con ordenador con radio *ethernet*

9.5.2. Diseño del $\mathcal{AA}$

9.5.2.1. Definición de los diferentes comportamientos

Los comportamientos de que va a disponer el $\mathcal{AA}$ para resolver el ejemplo presentado, y que a continuación se detallarán, son los siguientes:

- *Espera de Objetivos*.
- *Reparto de Cartas*.
- *Entrega de una Carta*.
- *Vuelta a la Base para Recargar*.

9.5.2.1.1. Espera de Objetivos

Este comportamiento controlará al $\mathcal{AA}$ en aquellas situaciones en las que el agente no dispone de objetivos y se encontrará parado en la base a la espera de los mismos. Además de esperar la llegada de un nuevo objetivo, en este comportamiento el agente debe controlar el estado de sus baterías.

Así, este comportamiento está formado por los siguientes *in-agents*:

- *monitoriza_robot*: Está compuesto por un único MKS de nombre *mks_monitoriza_robot* que monitoriza el buen funcionamiento de la parte física del agente, del robot. Así, está al cargo ,entre otras cosas, del nivel de las baterías.

- *teleoperación*: Se divide en dos MKSs llamados *mks_teleoperación* y *mks_envio_informacion*, y es el encargado de recibir (por medio de una conexión por radio *ethernet*) el nuevo objetivo (carta) a repartir (*mks_teleoperacion*), así como de enviar la información del estado interno del robot, junto con su posición a la máquina remota para su visualización (*mks_envio_informacion*).

En la siguiente tabla se detallan las características de estos dos *in-agents* críticos:

in-agent	*Deadline*	Período	Nº Opc.	*WCET* Ini	*WCET* Fin
monitoriza_robot	200000	400000	1	1500	500
teleoperación	200000	400000	3	100	100

9.5.2.1.2. Reparto de Cartas

Este es el comportamiento que controlará al $\mathcal{AA}$ cuando éste tenga un objetivo, es decir, una carta que repartir a alguno de los despachos. De esta manera, el $\mathcal{AA}$ calcula la secuencia con recorrido mínimo de los despachos que debe visitar para repartir las cartas que les han encargado.

Los *in-agents* críticos de los que se compone este comportamiento son los siguientes:

- *monitoriza_robot*: Este es el mismo *in-agent* que en el comportamiento anterior.

- *evita_obstáculos*: Este *in-agent* es el que se encarga de modificar los movimientos del *in-agent* cuando se encuentra un obstáculo en mitad de un trayecto hacia un objetivo, tratando de que la modificación del trayecto sea la mínima posible.

- *robot_acciones*: Es el encargado de interactuar con la parte física del robot, actualizando en el $\mathcal{KDM}$ el estado interno del robot que llega por el

puerto serie, marcando en el $\mathcal{KDM}$ también si hay obstáculos cercanos (a partir de la información leída de los sonares), y, por último, enviando por el puerto serie la acción a ejecutar por el robot (que habrá leído previamente del $\mathcal{KDM}$).

- *teleoperación*: Este es el mismo *in-agent* que en el comportamiento anterior.

- *planifica_cartas*: Este *in-agent* se encarga de generar el plan de entrega de todas las cartas que tiene el cartero, con el objetivo de minimizar el recorrido respetando las prioridades definidas por el usuario. Cabe destacar que se ha utilizado *CLIPS* para implementar parte de uno de los MKSs que componen este *in-agent*.

- *cartero*: Su función es la de establecer el camino (plan) hasta el lugar donde debe entregar la siguiente carta.

in-agent	*Deadline*	Período	Nº Opc.	*WCET* Ini	*WCET* Fin
monitoriza_robot	200000	400000	1	1500	500
evita_obstáculos	200000	300000	1	100	100
robot_acciones	200000	300000	1	1500	500
teleoperación	200000	400000	3	100	100
planifica_cartas	200000000	400000000	1	100	100
cartero	200000	400000	1	100	100

9.5.2.1.3. Entrega de una Carta

Cuando el robot llega a uno de los despachos en los que debe entregar una carta, debe entrar en el mismo. Si la puerta se encuentra cerrada, el robot *llamará* a la misma chocando contra ella.

Los *in-agents* que componen este comportamiento ya han sido comentados por formar parte de otros comportamientos, y se recogen en la tabla si-

guiente:

in-agent	*Deadline*	Período	Nº Opc.	*WCET* Ini	*WCET* Fin
monitoriza_robot	200000	400000	1	1500	500
robot_acciones	200000	300000	1	1500	500
teleoperación	200000	400000	3	100	100
planifica_cartas	200000000	400000000	1	100	100
cartero	200000	400000	1	100	100

Como se puede observar, el cambio más importante que supone este comportamiento es que no incluye el *in-agent* encargado de evitar obstáculos, pues se quiere que el robot golpee la puerta.

9.5.2.1.4. Vuelta a la Base para Recargar

Cuando el robot detecte que el nivel de sus baterías se acerca al mínimo necesario para regresar a la zona de recarga, se olvidará de cualquier objetivo que tuviese o que le pueda llegar, volviendo a la base para su recarga. Sólo cuando las baterías alcancen su estado óptimo podrá el robot volver a hacer caso a sus objetivos.

Los *in-agents* que componen este comportamiento ya han sido comentados por formar parte de otros comportamientos, y se recogen en la tabla siguiente:

in-agent	*Deadline*	Período	Nº Opc.	*WCET* Ini	*WCET* Fin
monitoriza_robot	200000	400000	1	1500	500
evita_obstáculos	200000	300000	1	100	100
robot_acciones	200000	300000	1	1500	500
teleoperación	200000	400000	3	100	100

9.5.2.2. Construcción del AFND

Esta construcción supone la detección de las situaciones en las que se debe cambiar de comportamiento, y por tanto de las condiciones para las metareglas que se deben encargar de activar dichos cambios.

Así, se genera el siguiente autómata finito no determinista (figura 9.12):

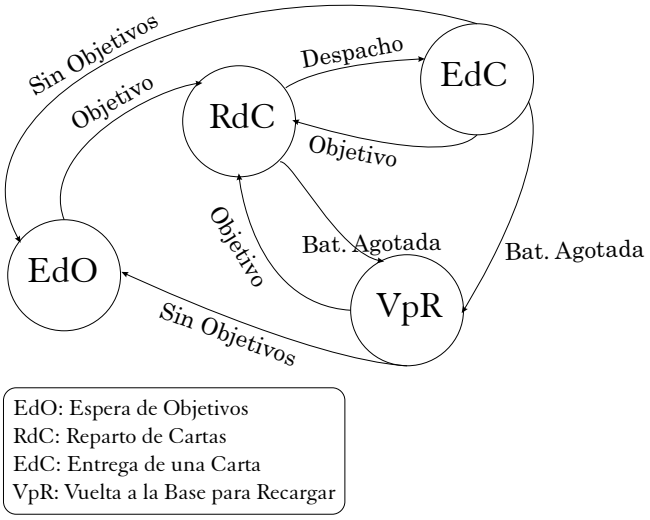

Figura 9.12: Autómata Finito No Determinista del $\mathcal{AA}$ Meta-Cartero

Como se puede observar, no se ha marcado ningún estado inicial ni final, ya que pueden existir diversos estados/comportamientos iniciales (dependiendo de si inicialmente existen objetivos, o de si la batería está cargada, ...), y además se supone que no hay un comportamiento final, el agente debería seguir ejecutándose indefinidamente.

Como ya se ha comentado, las transiciones entre estados/comportamientos se controlan por meta-reglas, que comprueban que la condición que hacía que ese comportamiento fuese el activo ya no es cierta, y además se da la condición para que se active otro comportamiento. Según las etiquetas indicadas en la figura 9.12 estas condiciones son:

- *Sin Objetivos*: No queda ningún objetivo en la memoria del $\mathcal{AA}$, es decir, no hay ninguna carta que repartir.
- *Objetivo*: Hay algún objetivo en la memoria del $\mathcal{AA}$, o lo que es lo mismo, tiene cartas que repartir.
- *Despacho*: Se ha alcanzado el despacho correspondiente a la siguiente carta a entregar.

- *Bat. Agotada*: Se han agotado las baterías, o su nivel tan sólo llega al necesario para regresar a la zona de recarga.

9.5.3. Conclusiones del Ejemplo

El objetivo del presente ejemplo era mostrar la forma en la que las nuevas extensiones al $\mathcal{AA}$ han afectado a la forma en la que se debe diseñar un agente en base a esta arquitectura.

Así, en primer lugar se deben localizar las situaciones que requieren distintas formas de actuar, y, por tanto, distintos comportamientos. Además, hay que construir un autómata finito no determinista para indicar las posibles transiciones entre dichas situaciones/comportamientos.

A partir de ahí, se debería asegurar la planificabilidad de las partes críticas de cada comportamiento, y programar las $\mu\mathcal{R}eglas$ que provocarán las diferentes transiciones entre comportamientos.

9.6. Conclusiones

En este capítulo se ha presentado por una parte el sistema de ejecución utilizado, prestando especial atención a la forma en que se ha desarrollado un modo de depuración que permite generar un cronograma para cada ejecución del agente, solventando el problema de las diferentes medidas de tiempo entre Linux y RT-Linux. Por otra parte, se ha mostrado mediante una serie de ejemplos las mejoras conseguidas con las extensiones desarrolladas, de entre las que cabe destacar:

- La gestión de múltiples comportamientos, que permite una gestión más eficiente de los recursos (destacando entre ellos el tiempo de procesador) dedicándolos en cada momento tan sólo a las tareas relevantes a la situación actual del agente.
- La comunicación asíncrona entre diferentes módulos de la arquitectura, que es la base del mecanismo de activación de $\mu\mathcal{R}eglas$ que permite la adaptación del agente a cambios significativos de la situación en que se encuentra. Este mecanismo de comunicación también soporta parte del incremento de flexibilidad en la utilización del tiempo de holgura

disponible, que es comunicado por el $\mathcal{RS}$ al $\mathcal{DS}$ y al revés, cuando el $\mathcal{DS}$ no tiene más operaciones que realizar puede devolverle parte de dicho tiempo al $\mathcal{RS}$

- La mejora de la gestión del tiempo por parte del $\mathcal{DS}$ le permite incluso el funcionamiento con datos incorrectos, tal y como se ha visto con los tiempos de ejecución en el peor caso mal calculados.
- El *Grado de Reactividad* variable, como un mecanismo fundamental para la adaptación del agente a diferentes situaciones, permitiendo al agente múltiples formas de comportarse, que van desde la completamente reactiva hasta la deliberativa en tiempo real.
- Por último, aunque no por ello menos importante, se ha presentado la forma en la que varía el diseño de un $\mathcal{AA}$cuando se tiene que tener en cuenta que el agente va a disponer de diferentes comportamientos como respuesta a las diferentes situaciones de interés a las que debe enfrentarse.

Parte V

Conclusiones

10 Conclusiones

"La lógica es el arte de no sacar conclusiones"
Atrib. Alan Wood, "El escéptico apasionado"
– Bertrand Russell, (1872-1970)

10.1. Conclusiones

En un sistema en tiempo real crítico tradicional se asegura el cumplimiento de las restricciones temporales críticas, pero no se asegura el efectivo aprovechamiento del tiempo restante después de ejecutar las partes críticas del sistema. Este uso además, viene condicionado no sólo por cuánto tiempo sobrante exista en el sistema, sino por los momentos en los que este tiempo esté disponible. En los últimos tiempos se han desarrollado en este tipo de sistemas algunas técnicas que permiten introducir flexibilidad en el sistema, como por ejemplo la gestión de tiempo de holgura, los cambios de modo, etc. Sin embargo, estas técnicas han aparecido como aportaciones aisladas donde además, se presentan como mecanismos de muy bajo nivel que le dejan al diseñador la responsabilidad de decidir cómo se usan.

Desde esta perspectiva de los sistemas de tiempo real, el trabajo presentado, que incorpora este tipo de técnicas de tiempo real, permite que el

diseñador especifique el conocimiento de forma cercana al problema que está resolviendo, abstrayéndolo de las peculiaridades / complejidad de dichas técnicas de bajo nivel de tiempo real.

Por otra parte, los mecanismos de meta-razonamiento introducidos permiten razonar acerca de cuál es el mejor uso de dichas técnicas en ejecución. Por ejemplo, el concepto de *grado de reactividad* permite razonar dinámicamente acerca de la utilización de la holgura que el sistema dedica a la planificación del $\mathcal{DS}$.

Desde la perspectiva de la inteligencia artificial, cabe destacar que todas las aportaciones desarrolladas tienen en cuenta su funcionamiento en un sistema con restricciones temporales críticas.

De una forma más concreta, el trabajo aquí presentado desarrolla nuevas técnicas y métodos que permiten que un agente con restricciones temporales críticas (en este caso el agente ARTIS) adapte su comportamiento a diferentes *situaciones*, entendiendo por situación la conjunción del estado interno del agente y del estado externo (del entorno).

Para ello, se ha extendido el modelo formal del $\mathcal{AA}$, el modelo de usuario, así como toda una serie de aportaciones al modelo de sistema o bajo nivel que permiten el diseño e implementación de esta capacidad de adaptación. De entre todo lo relatado en los capítulos anteriores cabe resaltar las aportaciones que se comentan en las siguientes secciones.

Antes de pasar a las aportaciones directas del presente trabajo, cabe destacar que además de dichas aportaciones, este trabajo permite la incorporación de la capacidad de comunicación dentro de un agente ARTIS, lo que le dota al $\mathcal{AA}$ de sociabilidad y, como consecuencia permite su incorporación en un sistema multi-agente. Esta es la base de un trabajo paralelo a éste en el que se está desarrollando una arquitectura/plataforma de sistemas multi-agente de tiempo real basados en la arquitectura de agente ARTIS a la que se ha denominado $\mathcal{SIMBA}$, "**SI**stema **M**ulti-agente **B**asado en **A**RTIS"[Julián 2002b, Soler 2002, Soler 2003, Carrascosa 2003b].

10.1.1. Detección de situaciones significativas de cambio

Para lograr la capacidad de adaptación comentada, es necesario que el agente sea capaz de detectar las situaciones frente a las cuales debe adaptarse.

Para ello se ha introducido el concepto de situación significativa de cambio, realizando las extensiones oportunas a los diferentes modelos que permiten que:

1. El diseñador especifique una situación significativa de cambio (por medio de la extensión realizada al lenguaje de usuario que permite definir los eventos que se deben generar).

2. Se genere un evento que indique la posibilidad de estar en una situación significativa (por medio del mecanismo de eventos incluido en el modelo de sistema que permite esta comunicación asíncrona).

3. El diseñador puede especificar la reacción frente a una situación significativa de cambio concreta (por medio del lenguaje de control incluido en las extensiones al módulo de usuario, y del mecanismo de gestión de meta-reglas y focos de atención).

10.1.2. El concepto de *comportamiento*

Este concepto, presente en las extensiones a todos los niveles de abstracción aquí presentadas, supone un cambio radical en la misma forma de plantearse qué es y para qué sirve un $\mathcal{AA}$ con respecto a la aproximación existente previa a este trabajo, siendo el verdadero eje de la adaptatividad del agente. De esta manera, el $\mathcal{AA}$ pasa de ser un agente que dispone de un único conjunto de tareas ordenadas por unas prioridades estáticas asignadas en la fase de diseño del mismo, que se van ejecutando periódicamente sin ninguna variación [1], a convertirse en un agente que puede disponer de formas diversas de enfrentarse al entorno, pudiendo así tener controladas respuestas especiales a situaciones críticas que poco o nada tengan que ver con la forma de enfrentarse a una situación normal.

Este cambio, ha supuesto, no sólo un cambio en la definición del agente, sino también en la forma de diseñarlo. Hay que diseñar los distintos comportamientos de que dispone el agente así como las transiciones existentes entre los mismos (para lo que se construye un autómata finito no determinista).

[1] lo que ha hecho que en alguna ocasión se le comparase, erróneamente, a algo similar a un *ejecutivo cíclico*

10.1.3. Reactividad flexible

El concepto de reactividad es común a la hora de hablar de agentes, no en vano es una de las características básicas exigidas a un agente en las definiciones más aceptadas del término. De hecho, su uso dentro de este ámbito es doble: lleva a hablar de que un agente ha de responder al entorno, o a hablar de agentes reactivos como de aquellos que al no poseer un modelo del entorno tienen una respuesta rápida al mismo (ver el capítulo del estado del arte para más información sobre el tema). Sin embargo, ni en este ámbito ni en el de los sistemas de tiempo real se habla de la posibilidad de que la reactividad del sistema sea *modificable*. Sí es cierto que en los sistemas de tiempo real se establecen conceptos como *deadline* o plazo máximo de ejecución, o se habla de sistemas donde la calidad del sistema se degrada conforme se retrasa la respuesta del mismo, pero en ningún caso existe la posibilidad de que el propio sistema decida cual va a ser el máximo retraso aplicable a la hora de dar una respuesta, ni que esa decisión sea dinámica y adaptable.

Esta capacidad (que se implementa mediante el concepto de *Grado de Reactividad*) le permite al $\mathcal{AA}$ controlar, de acuerdo a la situación en la que se encuentre, si va a calcular su respuesta de forma refleja, o mediante un proceso deliberativo, y, en este último caso, cuánto tiempo va a dedicar como máximo a este proceso. Evidentemente, teniendo en cuenta que, aún en el caso de que se desee que el $\mathcal{AA}$ sea tan deliberativo como pueda, existe un tiempo máximo para la respuesta del mismo dado por las restricciones temporales críticas que posea dicho agente.

10.1.4. Flexibilizar el aprovechamiento del tiempo de deliberación

Desde su concepción, la deliberación dentro de un $\mathcal{AA}$ tiene que tratar con un tiempo acotado para su realización, sin embargo, previo al trabajo expuesto, eso suponía tan sólo el controlar hasta cuando había de tiempo para deliberar. En las extensiones que se han desarrollado se lleva esto más allá, el $\mathcal{DS}$ puede decidir si quiere dedicar más tiempo a deliberar sobre cambios en el exterior o a procesos introspectivos de deliberación pura (mediante el *Grado de Introspección*), pudiendo variar esta decisión de acuerdo a la situación.

Se ha mejorado el ciclo de control haciéndolo más flexible y permitiéndo-

le un mejor aprovechamiento del tiempo disponible, dividiendo el ciclo de control en dos partes ($\mathcal{EM}$ y $\mathcal{SLS}$) y controlando de forma dinámica y adaptativa (por medio del *Grado de Introspección*) cuánto tiempo se le dedica a la gestión de eventos y cuánto se dedica a planificar y ejecutar niveles opcionales. Además se controla dentro del $\mathcal{DS}$ que acabe su ejecución cuando no queda tiempo para que ninguna de estas dos partes pueda realizar ninguna operación, o bien, ninguno de estos submódulos tiene nada que hacer.

Como último detalle en el aprovechamiento del tiempo disponible para deliberación, el $\mathcal{DS}$ puede ahora decidir si no tiene o no quiere seguir deliberando, comunicárselo al $\mathcal{RS}$ y éste, siempre que ello sea posible, adelantar la ejecución de la parte crítica del agente. Esta decisión influye en el anteriormente comentado aumento del control sobre la reactividad del agente, pues permite que dentro de los límites marcados por el *Grado de Reactividad* la deliberación del agente pueda acabar antes de dichos límites y el agente responder antes de ese límite, considerando el *Grado de Reactividad*, por tanto, como un límite máximo de la deliberación del agente.

10.2. Visión global de la nueva arquitectura

A modo de resumen, se presenta una visión global de cómo queda la arquitectura de $\mathcal{AA}$ después de las aportaciones del trabajo aquí presentado (figura 10.1).

Un $\mathcal{AA}$ viene definido por un conjunto de comportamientos (entre los cuales siempre habrá uno que será el activo, y que guiará la actuación del $\mathcal{AA}$). Cada comportamiento está compuesto por un conjunto de *in-agents*, cada uno al cargo de resolver una parte concreta del cometido global del $\mathcal{AA}$.

La arquitectura de $\mathcal{AA}$ es una arquitectura a dos niveles: uno encargado de la parte crítica (se ejecuta en RT-Linux) y otro de la parte opcional (se ejecuta en Linux). Tanto los *in-agents*, como el Módulo de Control participan de dicha división. La parte crítica de un *in-agent* es la capa *Refleja –Reflex–* (y se ejecuta en RT-Linux), mientras que la parte opcional de un *in-agent* es la capa *Deliberativa en Tiempo Real, RTD –Real-Time Deliberative–*. De la misma manera, el Módulo de Control se divide en:

- *Servidor Reflejo*, $\mathcal{RS}$*–Reflex Server–*: Incluye el *Planificador de Primer Nivel, FLS –First Level Scheduler–* y se encarga de la gestión de las partes

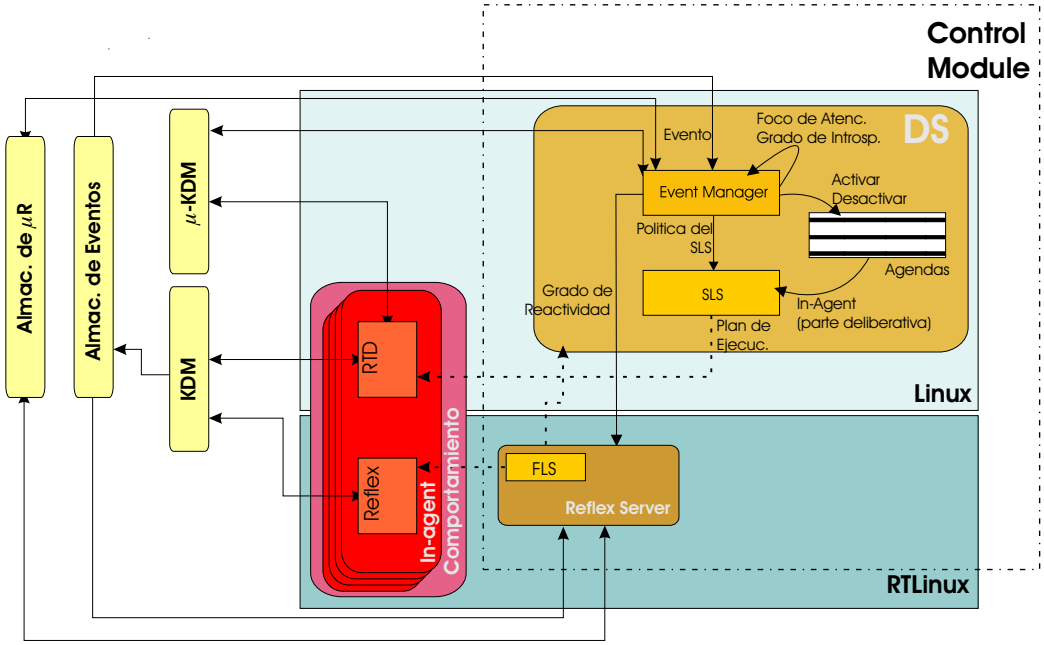

Figura 10.1: Nueva arquitectura de sistema de un $\mathcal{AA}$

críticas del $\mathcal{AA}$. Se ejecuta en RT-Linux.

- *Servidor Deliberativo, $\mathcal{DS}$–Deliberative Server–*: Se encarga de gestionar las partes opcionales del $\mathcal{AA}$. Se ejecuta en Linux. Está dividido a su vez en dos módulos, el *Gestor de Eventos, $\mathcal{EM}$–Event Manager–* y el *Planificador de Segundo Nivel, $\mathcal{SLS}$–Second Level Scheduler–*.

Existen diversas zonas de memoria común tanto a la zona Linux como a la RT-Linux:

- Todas las creencias del $\mathcal{AA}$ se encuentran almacenadas en dos zonas de memoria común, el $\mu\mathcal{KDM}$ (para las meta-creencias, o creencias para el proceso de meta-razonamiento) y el $\mathcal{KDM}$ (para el resto de creencias).

- Un almacén de eventos significativos ($\mathcal{KDM}$*Event Store*), tanto relacionados con los datos del $\mathcal{KDM}$ y del $\mu\mathcal{KDM}$ como con diversa información que se deben comunicar el $\mathcal{RS}$ y el $\mathcal{DS}$.

- Un almacén de Meta-Reglas, para almacenar los diversos focos de atención del proceso de meta-razonamiento, junto con las diferentes meta-reglas que los componen.

Por otra parte, en un sistema de tiempo real con restricciones temporales críticas, como es el $\mathcal{AA}$, es parte primordial la eficiente gestión del tiempo, es por ello que se definen los siguientes términos (cuya relación queda reflejada en la figura 10.2):

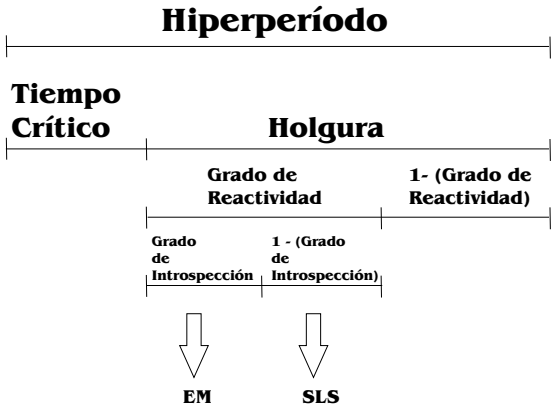

Figura 10.2: Gestión de tiempos de un $\mathcal{AA}$

- Hiperperíodo: Es el mínimo común múltiplo de los períodos de las tareas críticas del sistema de tiempo real, y, por tanto, indica el instante en el que todo el sistema vuelve a repetir su ejecución.

- Tiempo Crítico: Es el tiempo dedicado al cumplimiento de las restricciones temporales críticas, es decir, para la ejecución de las partes inicial y final de las tareas críticas del sistema.

- Holgura: Es el tiempo sobrante o *slack* después de haber ejecutado las partes inicial y final de las tareas críticas, y se dedica a la mejora de la solución del sistema.

- Grado de Reactividad: Define el porcentaje de la holgura que se va a usar para el proceso deliberativo del $\mathcal{AA}$.

- Grado de Introspección: Define el porcentaje del tiempo que se dedica para el proceso deliberativo que se va a dedicar al gestor de eventos ($\mathcal{EM}$), y que se utilizará en procesar eventos, y ejecutar $\mu\mathcal{R}eglas$ como respuesta a dichos eventos.

  El porcentaje sobrante, una vez descontado el definido por el grado de reactividad, del tiempo dedicado al proceso deliberativo lo gestionará el $\mathcal{SLS}$ para ejecutar los niveles opcionales de tareas críticas y los niveles aperiódicos acríticos (activados por $\mu\mathcal{R}eglas$ por medio del $\mathcal{EM}$ como respuesta a eventos).

10.3. Líneas de Desarrollo Futuras

Este trabajo aquí presentado no es un punto y final a esta línea de investigación, sino un punto y seguido que deja pendientes algunas cosas y abre la puerta a otras más. A continuación se detallan algunos de los temas a desarrollar e investigar en el futuro.

10.3.1. Incorporación de algoritmos de aprendizaje

Parte del trabajo a realizar para complementar/completar el que aquí se presenta será el estudio de las técnicas de aprendizaje adecuadas y su posterior adaptación e implementación dentro del $\mathcal{AA}$. Este aprendizaje se debería aplicar a varios niveles diferentes que van desde el ajuste de los tiempos de ejecución en el peor caso (*wcet –worst case execution time–*) de las distintas tareas que conforman un comportamiento, hasta la posible deducción de reglas de comportamiento en tiempo de ejecución.

Para poder realizar este aprendizaje hay que estudiar qué técnica es la más adecuada (redes neuronales artificiales, algoritmos genéticos, reglas o técnicas de *machine learning*, ...). Para ello habrá que tener en cuenta las restricciones con las que trabaja el agente ARTIS, siendo la más importante el trabajo con un tiempo de ejecución limitado.

De esta manera, una primera aplicación de los algoritmos de aprendizaje a un agente ARTIS es ajustar los tiempos de ejecución en el peor caso, tanto los que están asociados a cualquier tarea crítica y opcional, como los que controlan el tiempo de ejecución de cada una de las fases del $\mathcal{DS}$.

Por otro lado, cabe recordar que todo el diseño e implementación del almacén de Meta-Reglas se ha realizado teniendo en cuenta que en el futuro se prevé la incorporación de un mecanismo que basándose en técnicas de aprendizaje sea capaz de incorporar (y de olvidar) meta-reglas.

La última aplicación a desarrollar de los algoritmos de aprendizaje sería la de los ya planteados *comportamientos reflejos adquiridos*.

10.3.2. Transición Asíncrona entre comportamientos

Como se ha comentado previamente, debido al estado actual de la implementación del $\mathcal{KDM}$ que utiliza tan sólo un mútex para la protección del acceso al mismo, se ha implementado un protocolo síncrono de cambio de modo para la transición entre comportamientos. Sin embargo, en cuanto se modifique el $\mathcal{KDM}$, se podrá proceder a la implementación y prueba del protocolo asíncrono de cambio de modo o transición entre comportamientos propuesto en esta tesis.

10.3.3. Dotar de mayor flexibilidad al $\mathcal{RS}$

Una de las continuaciones lógicas del trabajo aquí presentado es, de forma similar a como se ha hecho con el $\mathcal{DS}$, extender las capacidades del $\mathcal{RS}$ para conseguir un mejor funcionamiento. Una de estas mejoras consiste en la posibilidad de disponer de diversos algoritmos de planificación para el $\mathcal{FLS}$, y que sea el $\mathcal{RS}$ el que los vaya intercambiando de acuerdo a la situación actual.

Para que esto resulte posible, hay que estudiar en qué situaciones se podría intercambiar de forma segura el algoritmo de planificación del $\mathcal{FLS}$, pues hay que tener sumo cuidado a la hora de introducir una sobrecarga en el $\mathcal{FLS}$, pues es, como ya se ha comentado, el responsable de asegurar el cumplimiento de las restricciones críticas del $\mathcal{AA}$. De esta manera, las situaciones más seguras para poder intercambiar dichas políticas serían:

- En el hiperperíodo, pues es el *reset* por antonomasia de un sistema de tiempo real, donde todo comienza de nuevo.
- Cuando el $\mathcal{DS}$ devuelve el control al $\mathcal{RS}$ antes de consumir toda la hol-

gura disponible[2] al ser invocado después de que se haya ejecutado una parte final: Este caso habría que estudiarlo más detenidamente.

Hay que tener en cuenta además, que el disponer de distintas políticas de planificación para el $\mathcal{FLS}$ supone que dichas políticas pueden estar activas en cualquier momento, coincidiendo con cualquier comportamiento, con lo que deberían existir análisis de planificabilidad que permitiesen en la fase de diseño del $\mathcal{AA}$ asegurar que todos los comportamientos sean planificables con cualesquiera de los algoritmos de planificación que estuviese activo.

10.3.4. Meta-Razonamiento en Sistemas Multi-Agente de Tiempo Real

De forma paralela al desarrollo de este trabajo, se ha estado trabajando en el desarrollo de una arquitectura/plataforma de sistemas multi-agente de tiempo real basados en la arquitectura de agente ARTIS a la que se ha denominado $\mathcal{SIMBA}$, "**SI**stema **M**ulti-agente **B**asado en **A**RTIS"[Julián 2002b, Carrascosa 2003b].

El sistema multi-agente, como un todo, trata de resolver un problema, por eso tiene sentido plantearse el meta-razonar acerca del sistema multi-agente como un todo, dedicar un agente dedicado al proceso de meta-razonamiento acerca del sistema multi-agente, tal y como hacen en [Pěchouček 2003].

[2]Por medio de la rt-fifo establecida para ello

Parte VI

Anexos

A

Gestión de Eventos

"No pretendo haber controlado los eventos, confieso,
por el contrario, que ellos me han controlado a mí."
Carta,4-IV-1864
– Abraham Lincoln, (1809-1865)

A.1. Introducción

Mecanismo de Comunicación Asíncrono entre el $\mathcal{KDM}$ y el Módulo de Control, y entre el $\mathcal{RS}$ y el $\mathcal{DS}$

Como ya se ha comentado previamente en este documento, uno de los mecanismos básicos necesarios para poder adaptar el comportamiento de un $\mathcal{AA}$ a cambios en el entorno, es la posibilidad de gestionar eventos desde el Módulo de Control. Desde el punto de vista del modelo de sistema, lo que se ha implementado en este caso es un mecanismo de comunicación asíncrono entre el $\mathcal{KDM}$ y el Módulo de Control.

Una cuestión de elevada importancia y que dificulta la implementación de esta gestión es el hecho de que este mecanismo debe funcionar de la misma manera desde una parte crítica que desde una parte acrítica.

De esta manera, a lo largo de este capítulo se va a detallar cómo se incorpora esta gestión al Módulo de Control del $\mathcal{AA}$, cuál es la estructura de datos

elegida para almacenar los eventos y qué modificaciones hay que realizar en otras partes del modelo de sistema del $\mathcal{AA}$ para poder soportar la generación de eventos.

A.2. Especificación del Almacén de Eventos

Como se ha comentado, el mecanismo de eventos implementa una comunicación asíncrona entre diversos módulos del $\mathcal{AA}$. Al ser este mecanismo asíncrono hay que almacenar todos los mensajes (los denominados eventos) cuando se envían para su posterior recuperación.

Este almacén debe funcionar de forma similar a una cola *FIFO* devolviendo, en la extracción, el evento más antiguo del almacén. De esta manera, se utiliza una cola circular de eventos, donde cada uno de estos eventos tiene la siguiente estructura definida en $< event.c >$:

```
typedef struct _KDM_Event {
    int iType;
      /*** Tipo del evento según las constantes
       *    definidas en event.h
       ***/
    void * pSlot;
      /*** Puntero al slot que genera el evento ***/
    char iImp;  /*** Importancia del evento ***/
    int iDataType;
      /*** Tipo de datos del slot con valores según
       *     las ctes. definidas en blackboard.h
       ***/
    KDM_Value_T Old_Value, New_Value;
      /*** Valores del slot que provoca el evento.
       *   KDM_Value_T: tipo def. en blackboard.h
       ***/
} KDM_Event_T;
```

A.3. Tipos de Eventos

El tipo de un evento va a venir codificado en un número con 16 bits. El byte alto para los eventos $\mathcal{RS}$– $\mathcal{DS}$, y el byte bajo para los eventos generados

en el $\mathcal{KDM}$. En concreto, la codificación utilizada es la siguiente[1]:

```
#define CREATION                    0x1
#define DELETION                    0x2
#define MODIFICATION                0x4
#define NEW_PREDICTION              0x8
#define MATCHED_PREDICTION          0x10
#define UNMATCHED_PREDICTION        0x20

#define ALL                         0x3F

#define CONSIDER_PERIODIC_AGENT     0x100
#define TIME_MESSAGE                0x200
```

A.4. Funciones de Interfaz

A.4.1. Modificar los slots que deben generar eventos

Como ya se ha comentado en el capítulo anterior, existen tres funciones que permiten acceder al $\mathcal{KDM}$ para esta gestión, y su sintaxis concreta es la siguiente:

```
KDM_Add_Event ( KDM_Static_Slot_T *pSlot, int iType,
                char iImp );
```

Esta función permite añadir al *slot* apuntado por *pSlot* el evento indicado por *iType* con la importancia *iImp*.

Se pueden añadir varios eventos simultáneamente codificándolo en *iType*.

```
KDM_Release_Event( KDM_Static_Slot_T *pSlot, int iType );
```

Permite eliminar el/los evento/s indicado/s por *iType* de los que genera

[1] El evento TIME_MESSAGE es utilizado por el $\mathcal{RS}$ para comunicarle al $\mathcal{DS}$ cuanto tiempo tiene para su ejecución, y se genera cada vez que el $\mathcal{DS}$ es activado.

el *slot* apuntado por *pSlot*.

```
KDM_Set_Event( KDM_Static_Slot_T *pSlot, int iType,
               char iImp );
```

Esta función permite establecer cuáles van a ser los eventos que debe generar el *slot* apuntado por *pSlot*, y serán los indicados por *iType* con importancia *iImp*.

A.4.2. Acceso del $\mathcal{KDM}$ a la cola de eventos – Escritura

Para realizar este acceso el $\mathcal{KDM}$ utilizará la siguiente función definida en el fichero $< KDM\_events.c >$:

```
KDM_Insert_Event ( KDM_Event_T regEvent );
```

A.4.3. Acceso del Módulo de Control a la cola de eventos – Lectura

Para realizar este acceso el Módulo de Control utilizará las siguientes funciones definidas en el fichero $< KDM\_events.c >$:

```
KDM_Extract_Event(KDM_Event_T *regEvemt);
```

```
int bIsEmpty ();
```

Esta función devolverá un 1 en el caso en el que no haya ningún evento pendiente y un 0 en otro caso. Para su implementación se ha utilizado una variable que controle el número total de eventos (nodos no vacíos) que hay en cada momento (*nEvents*). Esta variable se incrementa en la función *KDM_Insert_Event()* y se decrementa en la función *KDM_Extract_Event()*

A.5. Implementación del almacén de eventos

La decisión final sobre la forma de almacenar eventos es utilizar como memoria el mismo mecanismo que se utiliza para generar la zona de memoria que se usa para almacenar el $\mathcal{KDM}$, para hacer esta memoria visible tanto desde la parte crítica como de la acrítica de un $\mathcal{AA}$.

Este mecanismo utiliza un buffer de memoria junto con unas funciones de acceso según se declaran en $< mbuff.h >$. Para su gestión como base para el $\mathcal{KDM}$, esta memoria se gestiona desde un nivel más alto de abstracción (*storage*), pero para su gestión como base del almacén de eventos no se utiliza este nivel adicional de abstracción, utilizando directamente la interfaz de $< mbuff.h >$.

De esta manera, desde la inicialización de la parte crítica del $\mathcal{AA}$ se reserva la memoria que se va a utilizar como almacén de eventos (por medio de la función *mbuff_alloc()*), mientras que en la inicialización de la parte opcional se enlaza con dicha memoria para poder acceder también a ella (por medio de la función *mbuff_attach()*).

A.5.1. Estructura Abstracta de Datos

Al tener que acceder a esta estructura de datos desde la parte crítica de una tarea, hay que tener en cuenta consideraciones de eficiencia y sobre todo de seguridad en los accesos que de otra manera se podrían haber obviado.

En cuanto a la implementación del almacén de eventos, se ha decidido implementarlo mediante un vector dinámico, *aKDMEvImp[]*, de tantos elementos como niveles de importancia distintos existen[2], para tasar los eventos, más un elemento adicional. Cada uno de los elementos de este vector gestiona una lista circular doblemente enlazada de nodos. Dentro de cada uno de estos nodos se puede almacenar la información que el Módulo de Control necesita sobre un evento del $\mathcal{KDM}$ o del $\mathcal{RS}$. El último elemento del vector (el de índice *MAX_IMPORTANCE_EV*) mantendrá en todo momento la lista circular doblemente enlazada de los nodos libres (no asociados a eventos) de *aKDMEvImp[]*. Esto es así porque se quiere mantener constante el número de nodos que existan en *aKDMEvImp[]* (en concreto, *MAX_KDM_EV* nodos).

[2] *MAX_IMPORTANCE_EV*

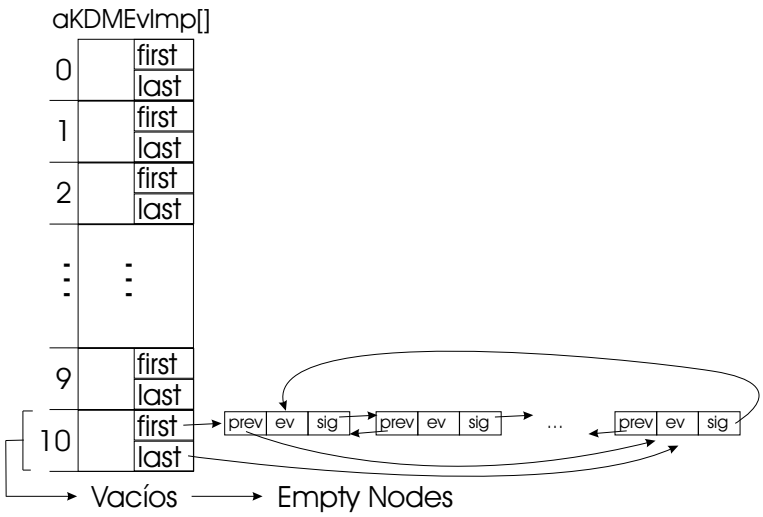

Figura A.1: Vector de listas doblemente enlazadas

De esta manera, cuando se quiere insertar un evento nuevo en *aKDMEvImp[]* se elige la lista adecuada según la importancia del evento (será la de índice igual a su importancia). Para insertar un elemento en esa lista se mira si hay nodos vacíos (en la última lista). Si es así, se coge uno y se pone en esta lista. Si no, se elige el nodo más antiguo de la lista de menor importancia, siempre que ésta sea inferior a la importancia del evento que se desea insertar. Si no hay ninguno de importancia inferior, se elimina el nodo más antiguo de importancia igual a la del nuevo evento. Si tampoco hay ninguno, el nuevo evento no se inserta.

A.5.2. Modificaciones en los ficheros ya existentes

Debido a que la implementación de este almacén de eventos no es algo aislado, sino que tiene que incorporarse en una implementación ya existente, además de un conjunto de ficheros que implementen esta funcionalidad *se han tenido que modificar ciertos ficheros* ***que no eran fruto del autor*** *de este documento.*

De esta manera se han creado los ficheros $< KDM\_events.h >$ y $< KDM\_events.c >$ que recogen las constantes y funciones necesarias para la gestión del almacén de eventos:

Constantes (declaradas en $< KDM\_events.h >$):

```
#define EVENT_STORE "EventStore"
                /*** Id for the mbuff ***/
#define MAX_KDM_EV            40
#define MAX_IMPORTANCE_EV     9
                /*** Event importance: 0..9
                 *        - Lowest: 0
                 *        - Biggest: 9
                 ***/
```

Además es en estos ficheros donde se declaran las dos variables que van a permitir luego gestionar el almacén de eventos:

```
int * piEvents;  /*** Current number of events ***/

typedef struct {
        n_KDM_Event * pFirst;
        n_KDM_Event * pLast;
} n_Imp;

n_Imp * aKDMEvImp;
     /*** Dynamic array of KDM-event queues. Each
      *     element of the array has the pointers
      *     to the queue of KDM-events which
      *     importance is the same as the index of
      *     the element in the array.
      *   The last element of this array will have
      *     the queue of empty nodes. This is to
      *     maintain a constant number of nodes in
      *     the array.
      ***/
```

Por último, la gestión de este almacén de eventos se realiza mediante las

siguientes funciones de interfaz (declaradas en el fichero $< KDM\_events.c >$):

```
int CreateEventQueueArray();
   /*** Used in critical initialization time ***/

int InitEventQueueArray();
   /*** Used in optional initialization time. ***/

int KDM_Insert_Event( KDM_Event_T regEvent );

int KDM_Extract_Event( KDM_Event_T * regEvent );

int bIsEmpty ();
```

La función *CreateEventQueueArray()* es invocada desde la función *KDM_Init_Blackboard()* (declarada en $< kdm.c >$) para realizar la reserva de memoria necesaria para el almacén de eventos. Como ya se ha comentado, esta función de inicialización se ejecuta desde la parte crítica del $\mathcal{AA}$, mientras que desde la parte opcional de dicho agente hay que indicar que se quiere utilizar dicha memoria. Para ello, desde la función *init_control()* (declarada en $< is.c >$) se invoca a *InitEventQueueArray()*.

A.6. Modificaciones necesarias del $\mathcal{KDM}$

A.6.1. Sobre las estructuras de datos

Se ha modificado la estructura de los *slots* para permitir almacenar los siguientes datos adicionales:

- Tipo de evento que produce el *slot*:

```
int iEvent_Type;
```

- Importancia del evento que se produce:

```
char iEvent_Imp;
```

Este valor está en el rango 0..9 y es función del Módulo de Control el que este dato sea correcto (realizar los castings a *char*).

Estas modificaciones se van a realizar sobre el tipo de datos *struct _KDM-_Static_Slot*. Este tipo de datos implementa la información de los *slots* estáticos, y está declarado en uno de los ficheros de cabecera del $\mathcal{KDM}$, el denominado $< blackboard.h >$

A.6.2. Sobre las funciones

Dentro del fichero $< kdm.c >$ se han realizado las siguientes modificaciones:

```
int KDM_Init_Blackboard();
```

Dentro de esta función, que implementa la inicialización del $\mathcal{KDM}$, se inicializa a 0 el *iEvent_Type* de todos los *slots* estáticos para indicar que por defecto no tienen asociado ningún evento.

```
int KDM_Set_Slot_Value();
```

Dentro de esta función que implementa el acceso al $\mathcal{KDM}$ desde la parte no crítica para modificar el valor de un *slot* se comprueba el valor almacenado en el campo *iEvent_Type* del *slot* a modificar y se ve si hay que invocar a *KDM_Insert_Event()* para insertar un evento asociado en la cola correspondiente.

Además se han añadido las siguientes funciones que se van a comentar posteriormente (pues forman parte de las acciones que se pueden realizar desde una meta-regla) y que conforman la interfaz que ofrece el $\mathcal{KDM}$ para

indicar cuando se quiere o no que se generen eventos:

```
int KDM_Add_Event();
```

```
int KDM_Release_Event();
```

```
int KDM_Set_Event();
```

Las modificaciones anteriores sobre $< kdm.c >$ se verán reflejadas en $< kdm.h >$ donde se han añadido los prototipos de las nuevas funciones.

Hay que tener en cuenta que existe una interfaz distinta para el acceso al $\mathcal{KDM}$ dependiendo de si éste se realiza desde la parte crítica u opcional de un $\mathcal{AA}$. Las modificaciones comentadas arriba sobre $< kdm.c >$ gestionan el acceso desde la parte opcional de un $\mathcal{AA}$ al $\mathcal{KDM}$ (a excepción de la inicialización del $\mathcal{KDM}$ que es única). De esta manera, estas modificaciones se han replicado en $< rt\_kdm.c >$, fichero que implementa el acceso desde la parte crítica.

A.7. Pruebas

Para poder comenzar a hacer pruebas, se ha optado por modificar los ficheros de la aplicación directamente, sin la necesidad de tener que modificar la herramienta de simulación y desarrollo de $\mathcal{AA}$ (*InSiDE*).

Se ha elegido un ejemplo de sobra conocido como aplicación de un $\mathcal{AA}$, el de la gestión de unos tanques de agua de los que se controla el nivel de los mismos pudiendo abrir distintas válvulas que permiten desalojar agua de los mismos antes de que se desborden.

Para poder realizar las mencionadas pruebas, de las que se darán cumplido detalle en el capítulo 9, se han modificado los ficheros *tanques3_rtkdm.c* y *tanques3_kdm.c*, que contienen una función *Handle_Register_Class()* que permite la inicialización de las variables de acceso al $\mathcal{KDM}$ desde la parte crítica y opcional del $\mathcal{AA}$, respectivamente. Estos ficheros habían sido generados

previamente por *InSiDE* en el directorio *app/source*[3].

Las modificaciones realizadas suponen el añadir dentro de la función anterior las llamadas oportunas a la función *KDM_Add_Event()*, para indicar qué eventos son de interés inicialmente (incluyendo previamente el fichero de cabecera en el que se encuentra definida, esto es, $< KDM\_events.h >$).

Además, se ha modificado el *Makefile* global para que copie el fichero objeto *KDM_events.o* al directorio *modules* (estas modificaciones se han realizado en las secciones *control_modules* e *install* de dicho *Makefile*). Adicionalmente, se ha cambiado el *script rtload* para que inserte el módulo *KDM_events.o* en el núcleo del RT-Linux.

En el capítulo posterior correspondiente se detallarán los requerimientos y funcionamientos del presente ejemplo.

A.8. Conclusiones

Este capítulo resalta una parte de la extensión del modelo de sistema de un $\mathcal{AA}$ que se ha desarrollado. Esta parte supone un mecanismo que permite una comunicación asíncrona entre diferentes módulos del agente, módulos que pueden estar tanto en Linux como en RT-Linux.

Cabe destacar que esta comunicación asíncrona junto con la gestión de meta-reglas que se expone en el capítulo siguiente son la base del meta-razonamiento y de la capacidad de adaptación del agente ARTIS desarrollada en este trabajo.

[3]Para poder compilar la aplicación después de las modificaciones a dichos ficheros hay que modificar *app/source/main.c*, si no el *make* no compilará los cambios

B

Meta-Reglas de un $\mathcal{AA}$

La siguiente especificación de este lenguaje está basada en la de Arlips 3.0d3 — Sintactic specification d4 —

```
attentionFocus -> attentionFocusDec
attentionFocus -> attentionFocus attentionFocusDec

attentionFocusDec -> '(' AttentionFocus Id metaRuleList ')'

metaRuleList -> metaRuleDec
metaRuleList -> metaRuleList metaRuleDec

metaRuleDec -> '(' DefMetaRule Id number eventCondition
                            Implication rhs ')'
metaRuleDec -> '(' DefMetaRule Id number Implication rhs ')'

eventCondition -> '(' slotEvent ')' lhs
eventCondition -> '(' CONSIDER_PERIODIC_AGENT Id ')' lhs

eventCondition -> lhs

slotEvent -> MODIFICATION slotId
slotEvent -> DELETION slotId
slotEvent -> NEW_PREDICTION slotId
slotEvent -> MATCHED_PREDICTION slotId
slotEvent -> UNMATCHED_PREDICTION slotId

slotId -> Id '.' Id '.' Id; // class.instance.slot

lhs -> '(' condition ')'
lhs -> lhs '(' condition ')'

condition -> Not '(' condition ')'
condition -> expression
```

```
expression -> simple_expr
expression -> simple_expr opRel expression

simple_expr -> factor
simple_expr -> sign factor
simple_expr -> simple_expr opAdd simple_expr
simple_expr -> simple_expr '*' simple_expr
simple_expr -> simple_expr '/' simple_expr

factor -> slotId
factor -> cte

sign -> '+'
sign -> '-'

opRel -> '<>'
opRel -> '<'
opRel -> '>'
opRel -> '>='
opRel -> '<='
opRel -> '='

opAdd -> '+'
opAdd -> '-'
opAdd -> Or

cte -> IntNumber
cte -> FloatNumber
cte -> True
cte -> False
```

```
rhs -> '(' action ')'
rhs -> rhs '(' action ')'

action -> slotId Assign simple_expr  // ?x not supported
action -> TriggerSporadicLevel '(' Id ')'
action -> SetReactivityDegree '(' FloatNumber ')'
action -> SetPolicy '(' slsPolicy ')'
action -> SetFocus '(' Id ',' focusAttentionDegree ')'
action -> SetAttentionDegree '(' FloatNumber ')'
action -> SetBehaviour '(' Id ')'
action -> KDM_Add_Event '(' nombre '.' nombre '.' nombre ','
                            number ',' number ')'
action -> KDM_Release_Event '(' nombre '.' nombre '.' nombre ','
                                number ')'

focusAttentionDegree -> FloatNumber

slsPolicy -> Edf   // Earliest Deadline First
slsPolicy -> Bif   // Biggest Importance First
slsPolicy -> Eldf  // Earliest Level Deadline First
slsPolicy -> Hqf   // High Quality First
slsPolicy -> Hsf   // High Slope First
slsPolicy -> Mu    // Marginal Utility
slsPolicy -> Dbm   // Dean & Boddy Modified
slsPolicy -> Ahuf  // Anytime Highest Utility First
slsPolicy -> Sss   // Slack Sizing Scheduling
```

Bibliografía

[Agre 1987] Agre, P., and Chapman, D. 1987. An implementation of a theory of activity. In *Proceeding of th Sixth National Conference on Artificial Intelligence (AAAI-87)*.

[Audsley 1995] Audsley, N.; Burns, A.; Davis, R.; Tindell, K. W.; and Wellings, A. J. 1995. Fixed priority pre-emptive scheduling: An historical perspective. *Real-Time Systems* 8:173–198.

[Barber 1994] Barber, F.; Botti, V.; Onaindía, E.; and Crespo, A. 1994. Temporal reasoning in reakt: An environment for real-time knowledge-based systems. *AICOMM* 7(3):175–202.

[Boddy 1994] Boddy, M., and Dean, T. 1994. Deliberation scheduling for problem solving in time-constrained environments. *Artificial Intelligence* (67):245–285.

[Botti 1999] Botti, V.; Carrascosa, C.; Julián, V.; and Soler, J. 1999. Modelling agents in hard real-time environments. In *MAAMAW'99 Proceedings*, volume 1647 of *LNAI*, 63–76. Springer-Verlag.

[Bratman 1987] Bratman, M. E.; Israel, D. J.; and Pollack, M. E. 1987. Toward an architecture for resource-bounded agents. Technical Report CSLI-87-104, Center for the Study of Language and Informations, SRI and Stanford University.

[Bratman 1988] Bratman, M. E.; Israel, D. J.; and Pollack, M. E. 1988. Plans and resource-bounded practical reasoning. *Computational Intelligence* (4):349–355.

[Brooks 1986] Brooks, R. A. 1986. A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation* 2(4):14–23.

[Brooks 1991] Brooks, R. A. 1991. Intelligence without representation. *Artificial Intelligence* (47):139–159.

[Burmeister 1992] Burmeister, B., and Sundermeyer, K. 1992. Cooperative problem solving guided by intentions and perceptions. *Decentalized AI. Elsevier Science Publisher B.V.* 3:77–79.

[Carrascosa 1997] Carrascosa, C.; Julián, V.; García-Fornes, A.; and Espinosa, A. 1997. Un lenguaje para el desarrollo y prototipado rápido de sistemas de tiempo real inteligentes. In *Actas de la CAEPIA'97*, 685–694.

[Carrascosa 2003a] Carrascosa, C.; Rebollo, M.; Julián, V.; and Botti, V. 2003a. Deliberative server for real-time agents. In *Multi-Agent Systems and Applications III: 3rd International Central and Eastern European Conference on Multi-Agent Systems, CEEMAS 2003*, volume 2691 of *LNAI*, 485–496. Springer.

[Carrascosa 2003b] Carrascosa, C.; Rebollo, M.; Soler, J.; Julián, V.; and Botti, V. 2003b. Simba architecture for social real-time domains. In *EUMAS 2003: The First European Workshop on Multi-Agent Systems*.

[Castelfranchi 1995] Castelfranchi, C. 1995. Guarantees for autonomy in cognitive agent architecture. *Intelligent Agents: Theories, Architectures, and Languages* 890:56–70.

[Cockburn 1996] Cockburn, D., and Nick R., J. 1996. *Foundations of Distributed Artificial Intelligence*. John Wiley & Sons. chapter ARCHON: A distributed artificial intelligence system for industrial applications, 319–344.

[Consortium 1994] Consortium, I. 1994. Imagine. final project report. Technical report.

[Crespo 1994] Crespo, A.; Botti, V.; Barber, F.; Gallardo, D.; and Onaindía, E. 1994. A temporal blackboard for real-time

process control. *Engineering Applications of Artificial Intelligence. Pregamon Press Ltd.* 225–256.

[Davis 1993a] Davis, R. I.; Tindell, K. W.; and Burns, A. 1993a. Scheduling slack time in fixed priority preemptive systems. In *Proc. Real-Time Systems Symposium, Raleigh-Durham, North Carolina, December 1-3*, 222–231. IEEE Computer Society Press.

[Davis 1993b] Davis, R. I. 1993b. Approximate slack stealing algorithms for fixed priority preemptive systems. Technical Report YCS217, Department of Computer Science, University of York.

[Domínguez 1992] Domínguez, T. 1992. *Definición de un Modelo Concurrente Orientado a Objetos para Sistemas Multiagente*. Ph.D. Dissertation, Dep. Ingeniería de Sistemas Telemáticos, E.T.S.I. Telecomunicación, Universidad Politécnica de Madrid.

[Drogoul 1994] Drogoul, A., and Ferber, J. 1994. *Simulating Societies: the Computer Simulation of Social Phenomena*. London: University of London College Press. chapter Multi-agent simulation as a tool for studying emergent processes in societies, 127–142.

[Etzioni 1991] Etzioni, O. 1991. Embedding decision-analytic control in a learning architecture. *Artificial Intelligence* (49):129–159.

[Ferber 1992] Ferber, J., and Drogoul, A. 1992. *Distributed Artificial Intelligence: Theory and Praxis*. Boston, MA: Kluwer Academic Publishers. chapter Using reactive multi-agent systems in simulation and problem solving, 53–80.

[Ferber 1996] Ferber, J. 1996. *Foundations of Distributed Artificial Intelligence*. John Wiley & Sons. chapter Reactive distributed artificial intellligence: Principles and applications, 287–314.

[Ferguson 1992] Ferguson, I. A. 1992. *TouringMachines: An architecture for Dynamic, Rational, Mobile Agents*. Ph.D. Dissertation, Clare Hall. Univ. of Cambridge.

[Franklin 1996] Franklin, S., and Graesser, A. 1996. Is it an agent, or just a program?: A taxonomy for autonomous agents. In *Proceedings of the Third International Workshop on Agent Theories, Architectures, and Languages*. Springer-Verlag.

[García-Fornes 1996] García-Fornes, A. 1996. *ARTIS: Un modelo y una arquitectura para sistemas de tiempo real inteligentes*. Ph.D. Dissertation, Dept. Sistemas Informaticos y Computacion. Univ. Politecnica Valencia.

[García-Fornes 1997] García-Fornes, A.; Terrasa, A.; Botti, V.; and Crespo, A. 1997. Analyzing the schedulability of hard real-time artificial intelligence systems. *Engineering Applications of Artificial Intelligence* 369–377.

[Garijo 1996] Garijo, M.; Cancer, A.; and Sánchez García, J. J. 1996. A multiagent system for cooperative network-fault management. In *Proceedings of the First International Conference on the Practical Applications of Intelligent Agents and Multi-Agent Technology, PAAM-96*, 279–294.

[Georgeff 1989] Georgeff, M. P., and Ingrand, F. F. 1989. Monitoring and control of spacecraft systems using procedural reasoning. Technical Report 03, Australian Artificial Intelligence Institute, Melbourne, Australia.

[Goldman 2001] Goldman R. P.; Musliner, D. J.; and Krebsbach, K. D. 2001. Managing online self-adaptation in real-time environments. In *Proc. of Second International Workshop on Self Adaptive Software*.

[Graham 2001] Graham, J. R. 2001. *Real-Time Scheduling in Distributed Multi-agent Systems*. Ph.D. Dissertation, Dept. of computer and Information Science. University of Delaware.

[Haddadi 1996] Haddadi, A., and Sundermeyer, K. 1996. *Foundations of Distributed Artificial Intelligence*. John Wiley & Sons. chapter Belief-desire-intention agent architectures, 169–186.

[Hägg 1994] Hägg, S.; Ygge, F.; Gustavsson, R.; and Ottosson, H. 1994. Da-soc: A testbed for modelling distributed automation

applications using agent-oriented programming. In *Proceedings of the Sixth European Workshop on Modelling Autonomous Agents and Multi-Agent Worlds (MAAMAW-94)*, 39–51.

[Hayes-Roth 1988] Hayes-Roth, B. 1988. A blackboard architecture for control. *Readings in Distributed Artificial Intelligence, Morgan Kaufmann Publishers, Inc.*

[Hayes-Roth 1993] Hayes-Roth, B. 1993. A satisfacing cycle for real-time reasoning in intelligent agents. Technical Report KSL-93-17, Knowledge Systems Laboratory. Stanford University.

[Hayes-Roth 1995] Hayes-Roth, B. 1995. An architecture for adaptive intelligent systems. *Artificial Intelligence* (72):329–365.

[Hendler 1990] Hendler, J.; Tate, A.; and Drummond, M. 1990. Ai planning: Systems and techniques. *AI Magazine* 2(11):61–77.

[Horvitz 1991] Horvitz, E. J. 1991. Time-dependent utility and action under uncertainty. Technical Report KSL-91-33, Knowledge Systems Laboratory, Stanford University.

[Huhns 1998] Huhns, M., and Singh, M. P. 1998. *Readings in Agents. Chapter 1*. Morgan Kaufman. 1–24.

[Iglesias 1996] Iglesias, C. A.; González, J. C.; and Velasco, J. R. 1996. *INTELLIGENT AGENTS II: Agent Theories, Architectures, and Languages*, volume 1037 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, m. wooldridge, k. fischer, p. gmytrasiewicz, n. jennings, j. m:uller, and m. tambe edition. chapter MIX: A general purpose multiagent architecture, 251–266.

[Iglesias 1998] Iglesias, C. A. 1998. *Definición de una metodlogía para el desarrollo de sistemas multiagente*. Ph.D. Dissertation, Universidad Politécnica de Madrid.

[Intel 1997] Intel. 1997. Using the rdtsc instruction for performance monitoring. Technical report, Intel Corporation.

[Jennings 1992a] Jennings, N. R.; Mamdani, E. H.; Laresgoiti, I.; Pérez, J.; and Corera, J. 1992a. Grate: A general framework for

cooperative problem solving. *IEE-BCS Journal of Intelligent Systems Engineering* 1(2):102–114.

[Jennings 1992b] Jennings, N. R. 1992b. Using grate to build cooperating agents for industrial control. In *Proc. IFAC/IFIP/IMACS Int. Sym. on Artificial Intelligence in Real Time Control*, 691–696.

[Jennings 1993] Jennings, N. R. 1993. Specifications and implementation of a belief desire joint-intention architecture for collaborative problem solving. *Journal of Intelligent and Cooperative Information System* 2(3):289–318.

[Jennings 1998] Jennings, N. R., and Wooldridge, M., eds. 1998. *Agent Technology. Foundations, Applications and Markets*. Springer-Verlang.

[Julián 1999] Julián, V.; Carrascosa, C.; and Botti, V. 1999. Formalización y traducción a un modelo ejecutable de las entidades de un agente artis. In *Actas de CAEPIA-TTIA'99*.

[Julián 2000] Julián, V.; González, M.; Rebollo, M.; Carrascosa, C.; and Botti, V. 2000. Inside: una herramienta para el desarrollo de agentes artis. In *Proceeding of SEID'2000*.

[Julián 2002a] Julián, V., and Botti, V. 2002a. Developing real-time multi-agent systems. In *Proceedings of 4th Iberoamerican Workshop on Multi-Agent Systems (Iberagents02)*.

[Julián 2002b] Julián, V.; Carrascosa, C.; Rebollo, M.; Soler, J.; and Botti, V. 2002b. Simba: An approach for real-time multi-agent systems. In *Topics in Artificial Intelligence: 5th Catalonian Conference on AI, CCIA 2002*, volume 2504 of *LNAI*, 282–293. Springer.

[Julián 2002c] Julián, V. 2002c. *RT-MESSAGE: Desarrollo de Sistemas Multiagente de Tiempo Real*. Ph.D. Dissertation, Universidad Politécnica de Valencia.

[Larousse 2003] Larousse, B. d. C. 2003. *Diccionario de Psicología*. SPES Editorial, S. L.

[Maes 1991] Maes, P. 1991. The agent network architecture (ana). *SIGART bulletin* 115–120.

[Minton 1994] Minton, S.; Bresina, J.; and Drummond, M. 1994. Total-order and partial-order planning: A comparative analysis. *Journal of Artificial Intelligence Research* (2):227–262.

[Moulin 1996] Moulin, B., and Chaib-draa, B. 1996. *Foundations of Distributed Artificial Intelligence*. John Wiley & Sons. chapter An Overview of Distributed Artificial Intelligence, 3–56.

[Muller 1994] Muller, J. P. 1994. A conceptual model for agent interaction. In *Proceedings of the second International Working Conference on Cooperating Knowledge Base Systems*, 213–234. DAKE Centre, University of Keel: Deen, S. M. (Ed.).

[Musliner 1995] Musliner, D. J.; Hendler, J.; Agrawala, A.; Durfee, E.; Strosnider, J.; and C.J., P. 1995. The challenge of real-time in ai. *IEEE Computer* 58–66.

[Musliner 2002] Musliner, D. J. 2002. Safe learning in mission-critical domains: Time is of the essence. In *Working Notes of the AAAI Spring Symposium on Safe Learning Agents*.

[Nii 1986a] Nii, P. 1986a. Blackboard systems: Blackboard application systems, blackboard systems from a knowledge engineering perspective. *The AI Magazine* 82–106.

[Nii 1986b] Nii, P. 1986b. Blackboard systems: The blackboard model of problem solving and the evolution of blackboard architectures. *The AI Magazine* 38–53.

[Nwana 1996] Nwana, H. S. 1996. Software agents: An overview. Technical report, Intelligent Systems Research. AAT, BT Laboratories, Ipswich, United Kingdom.

[Occello 1997] Occello, M., and Demazeau, Y. 1997. Cello: An agent model with real time constraints. In *Proceedings of Autonomous Agents 97, Marina del Rey, California, USA*. ACM.

[Occello 1998a] Occello, M.; Demazeau, Y.; and Baejis, C. 1998a. Designing organized agents for cooperation with real time

constraints. In *Proceedings of 1st International Workshop CRW'98, July 4-5, 1998, Paris, France, LNAI number 1456*, volume I, 25–37. Springer Verlag.

[Occello 1998b] Occello, M., and Demazeau, Y. 1998b. Modelling decision making systems using agents satisfying real time constraints. *IFAC Proceedings of 3rd IFAC Symposium on Intelligent Autonomous Vehicles* 1:51–56.

[Onaindía 1997] Onaindía, E. 1997. *Modelo de representación y razonamiento temporal para sistemas basados en el conocimiento de tiempo real*. Tesis doctoral, Dept. Sistemas Informáticos y Computación. Univ. Politécnica Valencia.

[Ovum 1994] Ovum. 1994. Intelligent agents: the new revolution in software. Technical report.

[Pedro 1999] Pedro, P. 1999. *Schedulability of Mode Changes in Flexible Real-Time Distributed Systems*. Ph.D. Dissertation, Department of Computer Science, University of York.

[Pěchouček 2003] Pěchouček, M.; Štěpánková, O.; Mařík, V.; and Bárta, J. 2003. Abstract architecture for meta-reasoning in multi-agent systems. In *Multi-Agent Systems and Applications III: 3rd International Central and Eastern European Conference on Multi-Agent Systems, CEEMAS 2003*, volume 2691 of *LNAI*, 84–99. Springer-Verlag.

[Raja 2001] Raja, A., and Lesser, V. 2001. Real-time meta-level control in multi agent systems. In *Proceedings of Multi-Agent Systems and Applications - ACAI 2001 and EASSS 2001 Student Sessions. Also Adaptability and Embodiment Using Multi-Agent Systems: AEMAS 2001 Workshop. Prague, Czech Republic*.

[Real 2000] Real, J. V. 2000. *Protocolos de Cambio de Modo Para Sistemas de Tiempo Real*. Ph.D. Dissertation, Departamento de Informática de Sistemas y Computadores. Universidad Politécnica de Valencia.

[Rebollo 2003] Rebollo, M.; Onaindía, E.; and Botti, V. 2003. Formal modelling of dynamic environments for real-time

agents. In *Conference Eastern Europe Multiagent Systems (CEEMAS '03)*, Lecture Notes in Artificial Intelligence, 475–484. Springer-Verlag.

[Rebollo 2004] Rebollo, M. 2004. *Imaginación en tiempo real. Representación y gestión de creencias temporales para agentes en entornos dinámicos.* Ph.D. Dissertation, Universidad Politécnica de Valencia.

[Rich 1994] Rich, E., and K., K. 1994. *Inteligencia Artificial*. McGraw-Hill.

[Russell 1989] Russell, S., and Wefald, E. 1989. *First International Conference on Principles of Knowledge Representation and Reasoning*. Morgan Kaufmann Publishers, Inc. chapter Principles of Metareasoning, 400–411.

[Russell 1995] Russell, S., and Norvig, P. 1995. *Artificial Intelligence: A Modern Approach*. Prentice Hall International Editions.

[Russell 1998] Russell, S. 1998. Metareasoning. In The MIT Encyclopedia of the Cognitive Sciences, MIT Press.

[Sargent 1992] Sargent, P. 1992. Back to school for a brand new abc. *The Guardian* (12 March, p. 28).

[Sha 1989] Sha, L.; Rajkumar, R.; Lehoczky, J.; and Ramamritham, K. 1989. Mode change protocols for priority-driven preemptive scheduling. Technical Report UM-CS-1989-060.

[Soler 2000] Soler, J.; Julián, V.; Carrascosa, C.; and Botti, V. 2000. Applying the artis agent architecture to mobile robot control. In *Proceedings of IBERAMIA'2000. Atibaia, Sao Paulo, Brasil*, volume I, 359– 368. Springer Verlag.

[Soler 2002] Soler, J.; Julián, V.; Rebollo, M.; Carrascosa, C.; and Botti, V. 2002. Towards a real-time multi-agent system architecture. In *Proceedings of First Workshop Challenges in Open Agent Systems 2002, AAMAS 2002, Bolonia*.

[Soler 2003] Soler, J. 2003. *SIMBA: Una Plataforma para el desarrollo de Sistemas Multiagente en entornos de Tiempo Real*. Ph.D.

Dissertation, Departamento de Sistemas Informáticos y Computación. Universidad Politécnica de Valencia.

[Sommaruga 1996] Sommaruga, L.; Avouris, N. M.; and van Liedekerke, M. H. 1996. *Foundations of Distributed Artificial Intelligence*. John Wiley & Sons. chapter The evolution of the Coopera platform, 365–400.

[Stankovic 1988] Stankovic, J. 1988. Misconceptions about real-time computing. *IEEE Computer* 12(10):10–19.

[Stankovic 1993] Stankovic, J., and Ramamritham, K. 1993. What is predictability for real time systems. Technical Report 1990-62, Dept. of Computer and Information Science. Univ. of Massachusetts.

[Steiner 1996] Steiner, D. D. 1996. *Foundations of Distributed Artificial Intelligence*. John Wiley & Sons. chapter IMAGINE: An integrated environment for constructing distributed artificial intelligence systems, 345–364.

[Terrasa 2000] Terrasa, A. 2000. *Flexible Real-Time Linux: A new environment for Flexible Hard Real-Time Systems*. Ph.D. Dissertation, Departamento de Sistemas Informáticos y Computación. Universidad Politécnica de Valencia.

[Terrasa 2002] Terrasa, A.; García-Fornes, A.; and Botti, V. 2002. Flexible real-time linux. *Real-Time Systems Journal* 2:149–170.

[Tindell 1992] Tindell, K. W.; Burns, A.; and Wellings, A. J. 1992. Mode changes in priority pre-emptively scheduled systems. In *IEEE Real-Time Systems Symposium*, 100–109.

[Uckun 1993] Uckun, S., and Hayes-Roth, B. 1993. A control architecture for intelligent mobile robots. Technical Report KSL-93-09, Knowledge Systems Laboratory. Stanford University.

[Van Beek 1990] Van Beek, P. 1990. Reasoning about qualitative temporal information. In *Proc. of 8th National Conference on Artificial Intelligence*, number 40, 728–734. Boston, Mass.: AAAI Press.

[Van Beek 1996] Van Beek, P., and D.W., M. 1996. The design and experimental analisys of algorithms for temporal reasoning. *Journal of Artificial Intelligence Research* 4:1–18.

[Van Parunak 1999] Van Parunak, H., and Odell, J. 1999. Engineering artifacts for multi-agent systems. Invited Speech at the MAAMAW'99 Workshop.

[Wittig 1992] Wittig, T., ed. 1992. *ARCHON: an Architecture for Multi-Agent Systems*. Ellis Horwood.

[Wooldridge 1995] Wooldridge, M., and Jennings, N. R. 1995. Intelligent agents: Theory and practice. *The Knowledge Engineering Review* 10(2):115–152.

Índice de Autores

Índice Alfabético

"O Captain! my Captain! our fearful trip is done
The ship has weathered every rack, the prize we sought is won.
The port is near, the bells I hear, the people all exulting.´´
"O Captain, my Captain!",
– Walt Whitman, (1819 - 1892)

"The rest is silence"
"Hamlet",
– William Shakespeare