3

030631·20

Universidad Nacional Autónoma de México

Unidad Académica de los Ciclos Profesional y de Posgrado

Instituto de Investigaciones en Matemáticas Aplicadas y en Sistemas

Un Ambiente para desarrollo de Sistemas Cooperativos

Tesis que para obtener el grado de *Maestra en Ciencias de la Computación*
presenta:

Fabiola López y López

Asesor: **M.C. Horacio Carvajal Sánchez Yarza**

Junio/1998

262998

TESIS CON FALLA DE ORIGEN

Universidad Nacional
Autónoma de México

Biblioteca Central
Dirección General de Bibliotecas de la UNAM

UNAM – Dirección General de Bibliotecas
Tesis Digitales
Restricciones de uso

DERECHOS RESERVADOS ©
PROHIBIDA SU REPRODUCCIÓN TOTAL O PARCIAL

Todo el material contenido en esta tesis esta protegido por la Ley Federal del Derecho de Autor (LFDA) de los Estados Unidos Mexicanos (México).

El uso de imágenes, fragmentos de videos, y demás material que sea objeto de protección de los derechos de autor, será exclusivamente para fines educativos e informativos y deberá citar la fuente donde la obtuvo mencionando el autor o autores. Cualquier uso distinto como el lucro, reproducción, edición o modificación, será perseguido y sancionado por el respectivo titular de los Derechos de Autor.

A la memoria de mi padre ***Fidel López Rodríguez*** ♣

A mi hija ***Ana Laura*** *con todo mi amor.*

Agradecimientos

Quisiera agradecer, en primer lugar, a la Benemérita Universidad Autónoma de Puebla las facilidades otorgadas para la realización de este trabajo, de forma particular las gestiones de las autoridades de la Facultad de Ciencias de la Computación M.C. Rafael Rivera Rodríguez y Dra. María de Lourdes Sandoval Solís.

A mi asesor de tesis M.C. Horacio Carvajal Sánchez Yarza por su paciencia, su ayuda y el tiempo cedido en la supervisión de este trabajo.

A los jurados de examen profesional: Dra Hanna Oktaba, Ing. Mario Rodríguez Manzanera, Dr José Negrete Martínez y Dr. Cristian Lemaître León por sus valiosos comentarios que permitieron mejorar el trabajo de tesis.

También quisiera agradecer de forma muy especial a mi madre Isabel y a mis hermanos Fidel, Araceli, Alejandro, Raquel, Mario y Eduardo cuyo apoyo constante ha sido muy importante para mí.

A Jaime por todos sus buenos deseos enviados desde el extranjero y a todos mis entrañables amigos por compartir los buenos y malos momentos, en particular a Ana y a Lulú, gracias por su apoyo.

Finalmente vaya un beso a la persona más importante en mi vida, Anita. Gracias por aguantar a tu madre en todos esos momentos que he tenido que robarte para lograr que te sientas orgullosa de mí.

Contenido:

Introducción		1
Capítulo 1	Sistemas Cooperativos	3
	1.1 Problemas básicos de los Sistemas Cooperativos	4
	1.2 Investigaciones en Sistemas Cooperativos	5
	1.2.1 Programación Orientada a Agentes	6
	1.2.2 INTERRAP	6
	1.2.3 KQML	8
	1.2.4 Pizarrones	8
	1.2.5 Redes de contrato	8
	1.2.6 MACE	9
Capítulo 2	ASC Ambiente para desarrollo de sistemas cooperativos	
	2.1 Descripción del ambiente ASC	11
	2.2 Arquitectura de ASC	12
	2.3 Agentes	12
	2.4 Shell	13
	2.5 Kernel	14
	2.5.1 Directorio	14
	2.5.2 Ejecutor	14
	2.5.3 Manejador de Archivos	15
	2.5.4 Depuradores	15
	2.5.5 Modificadores	15
	2.6 Clases y objetos en el sistema	16
	2.6.1 Clases ocupadas por los AGENTES	16
	2.6.2 Clases ocupadas por las ORGANIZACIONES	16
	2.6.2 Clases ocupadas del KERNEL	16
	2.6.3 Clases ocupadas por el SHELL	16
Capítulo 3	Agentes	
	3 1 Tipos de Agentes	18
	3.1.1 Reactivos	18
	3.1.2 Deliberativos	18
	3.1.3 Interactivos	19
	3.1.4 Arquitecturas híbridas	19
	3.2 Descripción de los agentes ASC	20
	3.2.1 Interacción	20
	3.2.2 Conocimiento	21
	3.2.3 Procesamiento	21
	3.3 Arquitectura de los agentes ASC	22
	3.3.1 Flujo de control	23
	3.4 Modelos en los agentes ASC	24
	3.4.1 Atributos de los modelos	24
	3.4.2 Funciones para los modelos	25
	3.5 El planificador	26
	3.5.1 Representación de los planes	26
	3.5.2 Función del planificador	27
	3.6 El ejecutor	28

3.7 Comunicación entre agentes ASC 29
3.7.1 El comunicador 29
3.7.2 El actualizador 32
3.7.3 Funciones interpretadoras de mensaje (FIM) 32
3.7 4 Conversaciones entre agentes 35
3.8 Desempeño del agente 39
3 9 Funciones en los agentes ASC 39

Capitulo 4 Organizaciones 41
4.1 Tipos de organizaciones 43
4.2 Estructuras organizacionales 44
4.3 Representando organizaciones jerárquicas 46
4 3 1 Instanciando organizaciones jerárquicas

Capitulo 5. Ejemplos 49
5.1 Aldeas de producción 49
5 1 1 Descripción 50
5 1 2 Implementación 52
5 1 3 Resultados 54
5 2 Redes de contratos 54
5 2 1 Descripción 55
5 2 2 Implementación 58
5 2 3 Resultados 62
5 3 Organizaciones jerárquicas 62
5 2 1 Descripción 63
5 2 2 Implementación 64
5 2 3 Resultados

Conclusiones 66

Apéndice A. Interfaz de usuario 69
A.1 Interfaz del sistema 75
A 2 Interfaz del modificador de agentes 77
A.3 Interfaz del modificador de planes 80
A.4 Interfaz del modificador de organizaciones

Apéndice B. Código de las clases más importantes en el sistema 83
B.1 Clase Modelo de agentes 85
B 2 Clase Agentes 88
B.3 Clase Actualizador 91
B.4 Clase Comunicador 92
B 5 Clase: Planes 92
B 6 Clase Fases 93
B 7 Clase Organizaciones 95
B 8 Clase: Roles

Glosario 97

Bibliografía 98

INTRODUCCIÓN

El ser humano siempre se ha distinguido del resto de la naturaleza por su capacidad de razonar, la cual le llevó a diseñar herramientas y estrategias que le permitieran superar sus propias limitaciones. Dentro de su mundo se enfrenta a múltiples problemas; algunos de ellos pueden ser solucionados por una sola persona; otros más pueden ser realizados entre varias personas, lo cual repercute directamente en el tiempo y la calidad de la solución; y otros más que deben solucionarse, de necesidad, entre un conjunto de personas debido a la naturaleza misma del problema, como por ejemplo el correo en una ciudad

Dentro del ambiente computacional ocurre algo similar con respecto a la forma de solucionar los problemas. Véase un ejemplo simple: invertir una matriz, bien puede realizarse por un sólo procesador en forma secuencial; si se disponen de varios procesadores la tarea puede repartirse entre ellos y posteriormente usar los resultados parciales para obtener la solución total. Sin embargo, por ejemplo, el problema de procesamiento en una red de cajeros bancarios no puede ser ejecutado por un simple procesador sino que se necesita usar una red de procesadores.

En sí, en el campo de la computación siempre se ha pretendido hacer una analogía entre el comportamiento humano y el de una computadora; métodos estudiados por la **Inteligencia Artificial**. Incluso investigaciones recientes tratan de ajustar las teorías de organizaciones humanas a organizaciones computacionales tal como lo describe Mark A. Fox [Fox89]; o bien a organizaciones integradas por humanos y computadoras como en el *TRABAJO COOPERATIVO ASISTIDO POR COMPUTADORA* (CSCW) [Gru91]; o bien en usar los elementos computacionales para trabajo en grupo (*GROUPWARE*).

Si se llaman **agentes** a las *entidades autónomas con capacidad para resolver problemas, que existen en un medio ambiente dinámico al cual pueden observar y/o modificar*, en esta definición están comprendidos tanto entes humanos como entes computacionales Claro está que esto genera controversias con diferentes corrientes humanistas que consideran los sentimientos, cualidades e incluso hasta los defectos de los seres humanos. Dejando de lado esas cuestiones y concentrándose sólo en el aspecto práctico de los agentes para resolver problemas, se pueden entonces, caracterizar y decir que poseen tres atributos principales: la capacidad de ***conocimiento*** de sí mismos y de su medio ambiente (incluyendo el conocimiento de otros agentes); la capacidad de ***procesamiento*** que les permite tomar decisiones y realizar acciones; y finalmente, la capacidad de ***interacción*** con otros agentes en su medio ambiente.

Es fácil de observar que los atributos de cada agente están limitados, (origen de la *teoría de racionalidad limitada* [LHK79]), sin embargo se considera que un agente es ***racional*** si su conducta es óptima con respecto a sus metas. La *racionalidad limitada* es una propiedad de los agentes que les permite conducirse en forma casi óptima con respecto a sus metas conforme sus recursos disminuyen, haciéndolos conscientes de

que existen problemas cuya complejidad crea la necesidad de requerir la ayuda de otros agentes.

Los agentes se unen para *cooperar y alcanzar, ya sea una meta individual o una meta común global*. Esta cooperación a nivel de agentes computadoras se ha visto reforzada con la aparición y el bajo costo de computadoras concurrentes muy potentes; de redes de computadoras; y de lenguajes y sistemas que trabajan sobre esos equipos.

La **Inteligencia Artificial** investiga la conducta inteligente de un agente al tratar de resolver un problema. El hecho de reconocer que *"la solución a muchos problemas involucra la actividad de grupos de agentes"*, ha dado origen a la **Inteligencia Artificial Distribuida** (DAI). Las investigaciones recientes en este campo están enfocadas al estudio de: *¿Cómo puede, una colección de agentes, interactuar para resolver un problema común ?*.

De **DAI** se desprenden dos grandes subáreas: *los sistemas de solución distribuida de problemas* o ***DPS (Distributed Problem Solving)***, en donde se tienen grupos de agentes que coordinan su trabajo para compartir conocimiento y desarrollar e integrar la solución a un problema dado (vgr. pizarrones, redes de contrato, etc.); y *los sistemas multiagentes* o ***MA (Multi-Agent System)*** en donde se tiene una variedad de agentes autónomos e inteligentes, quizás con metas diferentes e independientes, que pueden entrar en conflicto y necesitan coordinarse por si mismos para alcanzar sus metas (vgr. robots en una fábrica).

Dentro del área de la Inteligencia Artificial Distribuida se han enfocado estos problemas desde diferentes puntos de vista. Muchas investigaciones sugieren modelos para los sistemas de agentes cooperantes; otras sugieren arquitecturas para los agentes o bien los lenguajes y los protocolos de comunicación usados; y finalmente otras más pretenden crear herramientas que permitan representar esos modelos, es precisamente ésta última donde se pretende incidir con el presente trabajo.

El objetivo general de esta tesis es desarrollar una herramienta que proporcione un ***ambiente de experimentación para el modelado, la simulación, el monitoreo y la medición del desempeño de sistemas de agentes cooperativos***. Que sea lo bastante general para aceptar varias arquitecturas de sistemas cooperativos y diversos niveles de granularidad.

La presentación de la tesis se concentra en cinco capitulos, conclusiones y apéndices con información general. En el capitulo 1 se describen los problemas básicos de los sistemas cooperativos y se presentan los modelos más reconocidos de este tipo de sistemas. En el capitulo 2, se presentan los objetivos del trabajo y una descripción completa de la arquitectura del ambiente desarrollado. El capitulo 3 abarca los detalles de la arquitectura propuesta para los agentes. El capitulo 4 está dedicado exclusivamente a las organizaciones que se pueden representar en el ambiente propuesto. En el capitulo 5 se describen ejemplos de sistemas cooperativos que pueden desarrollarse usando el ambiente. Finalmente se presentan las conclusiones y las perspectivas del trabajo. Anexo al trabajo se incluyen la interfaz de usuario y el código de las clases más importantes en el sistema.

Capítulo 1

Sistemas Cooperativos

Los **Sistemas Cooperativos** pueden verse como una *sociedad de agentes distribuidos que se unen para encontrar la solución a un problema dado.* Dentro de estos sistemas el elemento más importante es la **cooperación**, la cual se puede definir *como la coordinación de las capacidades individuales de cada agente en el sistema.* Nótese que se habla de coordinación de capacidades y no solo de la unión de capacidades, ya que no basta sólo unir esfuerzos sino que hay que garantizar el **comportamiento coherente** del sistema

El significado del término **coordinación** es muy intuitivo, cuando se observa a un equipo ganador de basquetbol o a una línea de ensamblado trabajar correctamente, se dice que sus elementos están *"bien coordinados"*. Sin embargo en ocasiones es más fácil percibir la coordinación por su ausencia, por ejemplo cuando se espera mucho tiempo por un autobús y después de media hora llegan cinco de ellos juntos; eso indicará que el sistema de autobuses está *"mal coordinado"*

Se podría decir entonces que la **coordinación** es el *acto de trabajar juntos armoniosamente* o como lo describen Malone & Crouston [MC90] *"el acto de manejar la interdependencia entre actividades desempeñadas por varios agentes para alcanzar una meta común"*

La **coherencia** se refiere a *que "tan bien" el sistema se está comportando como una unidad* y se mide en base a la eficiencia del sistema mismo El grado de coherencia exhibida por el sistema es, hasta cierto punto, evitar *"actividades extrañas"* La incoherencia puede resultar, por ejemplo, de conflictos sobre recursos críticos que provocan embotellamientos, de agentes que escriben sobre resultados de otros, de acciones duplicadas, etc

Es importante hacer notar que dentro del ambiente computacional algunos autores han llamado a los **sistemas cooperativos** como **sistemas distribuidos**. Fox [Fox89], por ejemplo, nos proporciona la siguiente definición *"Un sistema distribuido es una organización que resulta de la distribución de un conjunto de tareas sobre un conjunto de elementos de procesamiento que están lógica o físicamente disjuntos"*. Sin embargo también existe otra definición para tal término. *"Un sistema distribuido consiste de múltiples procesadores autónomos que no comparten memoria principal, pero que cooperan enviándose mensajes sobre una red de comunicación"*, Bal, Steiner y Tanenbaum [TBS89]

A primera vista podría parecer que ambas definiciones son análogas, sin embargo la primera de ellas involucra la formación de una estructura organizacional, en donde cada elemento de procesamiento juega un rol en particular, pero puede ser sustituido por otro que sea capaz de realizar las mismas funciones Además se tiene una actitud "inteligente" por el hecho de estar formando un ambiente en el que varios elementos están

distribuyéndose el trabajo para encontrar la solución al problema y sobre todo porque la mayoría de las veces, la forma de encontrarla no esta definida totalmente al iniciar el trabajo. En contraste, la segunda definición, más bien se refiere al equipo físico que debe tomarse como soporte para el sistema distribuido de la primera definición. Para evitar este tipo de confusiones es preferible seguirles llamando *sistemas cooperativos*.

Los sistemas cooperativos proporcionan ciertas ventajas en la solución distribuida de problemas. Bond & Gasser las detallan en [BG89], de las más importantes se mencionarán sólo algunas:

Velocidad: La concurrencia incrementa la velocidad de procesamiento. Siempre y cuando, el tiempo empleado en la comunicación no sea mayor que el empleado en la solución del problema.
Confiabilidad: Estos sistemas son más confiables porque tienen la propiedad de falla parcial, es decir si un agente falla, no afecta el funcionamiento correcto de los demás puesto que su trabajo puede ser realizado por el resto de agentes en el sistema. También es posible en estos sistemas realizar tareas redundantes entre varios agentes, sobre todo cuando se tiene incertidumbre en la información, todo esto sin que repercuta grandemente en el desempeño del sistema.
Desarrollo incremental: Si un sistema está a punto de saturarse pueden agregárseles más agentes a fin de redistribuir el trabajo. Además cada parte del sistema puede desarrollarse independientemente por un especialista en un tipo particular de dominio o conocimiento.
Compartir recursos: Los agentes de un sistema tienen limitación de recursos, pero mediante la cooperación pueden disponer de los recursos de otros agentes.
Costo: En el caso de agentes computacionales esto puede verse porque, en algunas ocasiones, es mejor tener varios sistemas computacionales simples de bajo costo, que un sistema centralizado de costo elevado.
Naturalidad: Algunos problemas son por naturaleza distribuidos, por lo cual su implementación en este tipo de sistemas es relativamente sencilla. (v.g. sistema de correo, bancos, control de tráfico aéreo, etc.)

Aunque pareciera que las ventajas de implementar los sistemas cooperativos son muchas, también hay que considerar los problemas nuevos que se tienen que enfrentar para garantizar, como ya se había mencionado, la coherencia del sistema. Por ejemplo:

Confiabilidad: Si el diseño del sistema es pobre puede conducir a una pérdida de confiabilidad.
Interpretación: Tanto los eventos como los objetos que ocurren y concurren dentro del sistema pueden tener un significado diferente para cada agente.
Desorden: La descentralización del trabajo puede conducir a un crecimiento incontrolado, a imperios locales y a la dispersión de los agentes (en el ambiente social esto se conoce como burocracia). Además los agentes pueden tener autoridades o responsabilidades diferentes en determinado momento.

1.1 Problemas básicos de los Sistemas Cooperativos

Los problemas básicos de los sistemas cooperativos han sido estudiados dentro del campo de la Inteligencia Artificial Distribuida desde diferentes marcos de referencia. Para

el presente trabajo se ha tomado como base el marco propuesto por Horacio Carvajal y otros [HC93]:

Primero, se parte del hecho de que toda tarea que requiera más recursos o conocimiento, de los que posee un agente, debe descomponerse para que pueda terminarse. En este caso, cuando el trabajo se realiza por un sistema cooperativo, es importante responder la pregunta de la división y organización del trabajo: *¿Cuál de los agentes hará qué tarea y cuando?* La racionalidad limitada de un agente puede dirigirse a establecer una ***organización*** de trabajo y una ***planeación*** adecuada que puede incluir la ***negociación*** o ***distribución directa*** de tareas entre los diferentes agentes que formarán la organización del sistema cooperativo.

Una distribución de las tareas entre agentes, para su ***procesamiento***, requiere que la tarea se formule y describa de tal forma que permita ser distribuida. Para asignar una tarea a un agente en particular deben existir ***mecanismos de interacción*** entre los agentes que permita determinar *¿A quién se le asignará qué tarea?, ¿Cómo se le asignará la tarea?, ¿Cómo se efectuará el seguimiento de su ejecución?* y finalmente debe existir un mecanismo para saber *¿Cómo sintetizar los resultados parciales en la solución global del problema?*

Para la ***interacción*** entre agentes es necesario que se tengan ***lenguajes y protocolos de comunicación*** *preestablecidos*. Además se deben habilitar a los agentes individuales para *representar y razonar las acciones, planes y conocimiento de otros agentes* para coordinarse entre ellos y asegurar de este modo la coherencia del sistema. Este problema se ha resuelto estableciendo ***modelos de agentes*** y de organizaciones de agentes y creando diferentes lenguajes y protocolos de comunicación.

Una vez establecido un modelo cooperativo de trabajo se debe hacer un análisis de que tan eficiente está resultando el sistema y después de una evaluación del mismo decidir si se continua con el mismo esquema o es necesario plantear uno alternativo al ya existente. Para esto se tiene la ***reorganización.***

En el trabajo de Carvajal et al [HC93] se parte de que las limitaciones de los agentes pueden manifestarse en varios aspectos muy relacionados entre sí: **conocimiento, procesamiento e interacción**, y después de un análisis de varios trabajos se llega a la conclusión de que para superarlas es necesario permitir que los agentes se adapten a su medio ambiente, tomando para ello técnicas heurísticas tales como: **planeación, organización, reorganización, modelado y negociación**.

1.2 Investigaciones en Sistemas Cooperativos

El estudio de la Inteligencia Artificial Distribuida sigue dos caminos, el estudio formal de la solución distribuida de problemas y la parte de experimentación de los mismos. Dentro de los estudios formales se han propuesto: diferentes arquitecturas para los agentes tales como en la Programación Orientada a Agentes de Shoham [Sho90] y en el INTERRAP de Müller [Mull96], se han descrito diferentes lenguajes y protocolos de comunicación entre agentes como en KQML [FF94]; y se han propuesto modelos de organizaciones de agentes tales como las organizaciones basadas en Pizarrones [Cue91], las Redes de Contrato [DS80I] y el Trabajo Cooperativo Asistido por Computadora [Gru91].

Dentro de la parte experimental se han creado buenos ambientes para implementar diversas arquitecturas de sistemas cooperativos tal como el caso de MACE [GBH87].

1.2.1 Programación Orientada a Agentes

La *Programación Orientada a Agentes* [Sho90] es una especialización de la programación orientada a objetos, en donde la unidad principal es el agente en lugar del objeto. Al igual que los objetos, los agentes se comunican con paso de mensajes pero en este caso la sintaxis y semántica están restringidas por el objetivo de lograr la coordinación entre agentes, los tipos de mensajes incluidos sirven principalmente para: informar, comprometer, ofertar, prometer, negar, etc. El estado de los objetos en este caso está restringido al conjunto de parámetros que definen el estado mental de un agente, a saber sus creencias, sus decisiones, sus compromisos, sus capacidades, etc. Además el estado mental del agente también determina su comportamiento, el cual generalmente, está determinado por los compromisos adquiridos. Por lo tanto se puede decir que la conducta de los agentes está gobernada por programas cuyo control y estructuras de datos se basan en el estado mental de los agentes.

1.2.1.1 Flujo de control en agentes POA

La conducta de los agentes es muy simple. Cada agente itera en los dos siguientes pasos a intervalos regulares:

- Lee e interpreta los mensajes actuales, actualiza su estado mental (incluyendo sus creencias y compromisos).
- Ejecuta los compromisos del tiempo actual, posiblemente de esto resulten más cambios en las creencias y compromisos. Las acciones para las cuales los agentes están comprometidos incluyen acciones comunicativas tales como proporcionar informes y solicitar alguna información o bien acciones internas en los agentes.

1.2.2 INTERRAP

El modelo INTERRAP [Müll96] pretende definir un agente por medio de tres capas de conocimiento y de control: una ***capa basada en conducta*** que incorpora reactividad y conocimiento procedural para las tareas rutinarias; una ***capa de planeación local*** que proporciona las facilidades para razonamiento basado en fines para alcanzar las tareas locales y producir una conducta dirigida por metas; una ***capa de planeación cooperativa*** que permite a los agentes a razonar acerca de otros agentes y que supone acciones coordinadas con otros agentes.

De la misma forma, las creencias de los agentes, están divididas en: un ***modelo del mundo*** que contiene creencias a nivel de objeto acerca de su medio ambiente; el ***modelo mental*** que mantiene creencias de meta nivel que el agente tiene acerca de sí mismo; y el ***modelo social*** que mantiene creencias (de meta nivel) acerca de otros agentes.

La activación de las acciones del agente es iniciada siempre por situaciones específicas, i.e. por subconjuntos relevantes de sus creencias. Las metas son clasificadas en ***metas de reacción, metas locales y metas cooperativas***. Las metas locales corresponden a la noción estándar de una meta; las metas cooperativas están

compartidas entre un grupo de agentes. Las metas de reacción difieren de la noción común de metas: denotan metas a corto plazo que son activadas por eventos externos y que requieren una reacción rápida o la ejecución de un procedimiento de rutina.

1.2.2.1 Flujo del control en agentes Interrap

Generación y revisión de creencias. Se asume que los sensores proporciona la percepción del ambiente en una representación simbólica que es igual a sus creencias.
Reconocimiento de situaciones. El reconocimiento de situaciones le permite al agente identificar las necesidades para su actividad. En la capa basada en conducta, la situación debe identificarse enseguida y requiere una reacción rápida. Así, el reconocimiento de la situación por si misma debe ser rápida y eficiente. A nivel de la capa de planeación local se necesitan reconocer situaciones que requieren de planeación local o que de alguna forma afecten las metas locales actuales del agente. La capa de planeación cooperativa necesita identificar situaciones cuyo tratamiento deberá involucrar planeación cooperativa, tales como metas conflictivas entre diferentes agentes, o una cooperación potencial.
Activación de metas. Una situación que el agente ha reconocido, modifica el estado motivacional de un agente y activa el surgimiento de una meta En la capa de conducta, este proceso esta basado en primitivas operacionales llamadas *patrones de conducta (PoB)*. En la planeación local y cooperativa el proceso de activación de metas es más compleja ; puede involucrar la construcción explícita de un estado meta, el cual se usa por los mecanismos de planeación. En el caso cooperativo, una descripción de meta es un conjunto de metas para diferentes agentes.
Planeación. La función de planeación calcula que hacer y en que orden para alcanzar una meta En la capa basada en conducta, esta decisión es "hard-wired" es decir una meta corresponde a un PoB, y el plan para conseguir la meta esta dado por el cuerpo procedural del PoB. En la capa de planeación local, la planeación para una meta involucra un análisis dirigido por fines y la generación de un plan local. Finalmente la planeación cooperativa significa hallar un plan conjunto cuya ejecución satisfaga la unión de las metas contenidas en el descriptor de metas
Scheduling. Dado un conjunto de metas, y un plan de como alcanzar esas metas, el *scheduling* soluciona el problema de unir esos subplanes en un *scheduler* ejecutable. En la capa basada en conducta , *scheduling* significa decidir, de un conjunto de PoBs activos, cuál será ejecutado en el próximo ciclo de control. En particular el *scheduling* de PoBs debe asegurar que aquellos PoB que reaccionan ante situaciones urgentes sean tratados con alta prioridad. En la capa de planeación local las tareas del *scheduling* involucran la secuenciación de planes no lineales, la asignación de tiempos iniciales y finales para los pasos de planes restringidos en el tiempo, y el reconocimiento de incompatibilidades entre planes para alcanzar diferentes metas, causando que el planeador deba modificar sus planes.
Ejecución. En cada capa los agentes hacen compromisos que deben ser implementados. La función de ejecución es responsable de iniciar la ejecución de esos compromisos a tiempo y de monitorear su ejecución. En la capa basada en conducta, la función de ejecución mantiene la ejecución de PoBs. La capa de planeación local ejecuta pasos primitivos del plan. La capa de planeación cooperativa mantiene la ejecución de protocolos de negociación (meta nivel) por un lado y compromisos para planes conjuntos (nivel de objeto) en el otro.

1.2.3 KQML

KQML (Knowledge Query Manipulation Languaje) [FF94] es un lenguaje y un protocolo para soportar la comunicación entre agentes de software por medio de acciones lingüísticas explícitas. Es un formato de mensaje y un protocolo de manejo de mensajes para soportar el ***compartir conocimiento*** y la ***interacción*** entre agentes en tiempo de ejecución (run-time). KQML puede usarse en cualquier medio ambiente donde los agentes necesitan comunicarse algo mas que sentencias fijas y predefinidas y que necesiten además proporcionar interacción dinámica en tiempo de ejecución, de tal forma que permita a los agentes combinar sus esfuerzos y hacer uso de las habilidades de otros agentes para alcanzar sus propias metas.

1.2.4 Pizarrones

En los *Sistemas de Pizarrón* [DLC89], un conjunto de agentes, típicamente llamados **fuentes de conocimiento (KS)** comparten una base de datos común o un pizarrón de estructuras simbólicas, frecuentemente llamadas las hipótesis. Cada agente es un experto en algún área, y puede hallar una hipótesis sobre la cual pueda trabajar y resolverla, crear nuevas hipótesis, y modificar otras hipótesis existentes. En este caso los agentes cooperan al compartir un pizarrón común, al igual que lo haria un grupo de expertos humanos, cada uno dotado de su experiencia, un pedazo de gis y un pizarrón común. Los sistemas de pizarrón convencionales, están ejemplificados en HEARSAY II y sus descendientes, aunque éstos no fueron verdaderamente paralelos ya que corrían sobre máquinas seriales e incorporaban despachadores que cuidaran la invocación secuencial de agentes para mantener la consistencia entre ellos y el pizarrón

1.2.5 Redes de Contratos

Las *Redes de Contrato* son sistemas para la solución distribuida de problemas, diseñados por Smith y Davis [DS89] La meta principal de este sistema es la asignación oportunística de tareas entre una colección de agentes resolvedores de problemas, usando un marco llamado **negociación** basado en la anunciación de tareas, y en ordenes y adjudicaciones de *contratos*.

La arquitectura básica comprende agentes con reglas de manejadores y trabajadores; cualquier agente podrá ser en cualquier tiempo un manejador o un trabajador para tareas diferentes. Un agente manejador para una tarea deberá:

- Dividir la tarea en subtareas. La división de tareas se supone dada, es decir la tarea debe ser descrita en un camino tal que permita la división.
- Construir anuncios de tarea para cada subtarea y distribuirlos entre otros agentes, ya sea transmitiéndolo a todos o bien enfocando un conjunto particular de agentes.
- Seleccionar la oferta más apropiada y asignar la tarea al mejor postor (i.e. elaborar un contrato)
- Monitorear el progreso del contrato, posiblemente requiriendo información, reportes internos y así por el estilo, y es libre de reasignar la subtarea si el contratista falla al no completar la tarea oportunamente.
- Integrar el resultado parcial producido por sus contratistas en una solución completa a su nivel de descomposición.

Los agentes trabajadores tienen las siguientes funciones:

- Recibir el anuncio de la tarea.
- Evaluar sus propios recursos y regresar ofertas para hacer la tarea si es de su interés.
- Una vez que se les asigna, ejecutan las subtareas.
- Proporcionan reportes internos sobre sus progresos, resultados parciales, etc.
- Reportar resultados producidos para las tareas ejecutadas, al manejador de la subtarea.

El *Protocolo de Redes de Contrato* (CNP) fue diseñado para resolver este problema, donde los agentes pudieran intercambiar procedimientos y datos para argumentar sobre sus capacidades durante la solución del problema, de tal forma que siempre se proporcionen a los mejores trabajadores para realizar la tarea.

1.2.6 MACE

El ambiente MACE [GBH87], es un ambiente experimental que proporciona las siguientes facilidades

- Mapea los agentes en procesadores
- Maneja la comunicación entre agentes (facilidades para interpretar mensajes).
- Los agentes corren en paralelo.
- Proporciona un lenguaje para descripción de agentes (ADL)
- Permite medir características del problema (tráfico de mensajes, tamaño de colas y base de datos, trabajo realizado por cada agente y la carga sobre un nodo procesador)
- Permite invocación, dirigida por patrones, de eventos asíncronos.
- Permite el acceso a bases de datos asociativas dentro de los agentes.

MACE es un sistema distribuido orientado a objetos que está formado por una colección de componentes que incluyen:

Una colección de agentes. Las unidades computacionales básicas en MACE son agentes y son por naturaleza "sociales", ellos conocen a otros agentes en su medio ambiente, y esperan prepararse y coordinarse con la experiencia de lo que conocen de ellos. De esta forma los agentes necesitan conocer la identidad y localización de sus conocidos, algunas cosas acerca de sus capacidades, sus metas, etc. Los agentes de MACE representan esta información en forma de "modelos de otros agentes en el mundo". Los agentes pueden organizarse en subunidades las que actúan en respuesta a problemas particulares, es decir pueden responder como grupos organizados o compuestos.
El lenguaje de descripción de agentes de MACE que incluye facilidades para describir a los agentes y a grupos organizados de agentes.
Una comunidad de agentes del sistema. Los agentes predefinidos del sistema MACE facilitan la interpretación de comandos, la interfaz estándar con el usuario, el manejo controlado de errores, el seguimiento y el monitoreo de la ejecución, etc.. Los agentes específicos de MACE sirven como herramientas interactivas para construir programas MACE. Tales herramientas incluyen agentes para construir y editar la conducta y

estructura de otros agentes, tan pronto como cambien los parámetros del medio ambiente de ejecución o simulación.

Una colección de facilidades. Las cuales pueden usar todos los agentes. Estas incluyendo un patrón de coincidencias, un simulador, varios motores estándar de agentes, manejador estándar para errores y mensajes estándares entendidos por todos lo agentes.

El "**probador**" de MACE es un instrumento para permitir medir las características de los resolvedores de problemas durante las corridas experimentales. El tráfico de mensajes, tamaño de colas y bases de datos, trabajo realizado por un agente (en el término de tiempo real transcurrido o número de invocaciones), y la carga sobre un nodo procesador. MACE incorpora extensas facilidades para coincidencia de patrones en la interpretación de mensajes, invocación dirigida por patrones de eventos asíncrono (v.g., demonios y otros monitores de eventos), y para acceso de base de datos asociativa sin agentes.

Una base de datos descriptiva. La descripción de agentes se mantiene en una base de datos descriptiva por un grupo de agentes del sistema los cuales construyen nuevas descripciones, verifica demostraciones, y construyen agentes ejecutables desde la descripción

Una colección de núcleos. Los núcleos de MACE manejan en forma colectiva la comunicación y el ruteo de mensajes, ejecución de E/S para terminales, archivos y otros dispositivos, mapean agentes en procesadores y planean la ejecución de los agentes

Como se ha descrito en ésta última sección, el estudio de la Inteligencia Artificial Distribuida tiene dos vertientes: el estudio formal de modelos de sistemas cooperativos y el desarrollo de herramientas de experimentación. Dentro de este contexto, el presente trabajo presenta el desarrollo de una herramienta que facilita la descripción de agentes y que permite la construcción y depuración de diferentes modelos de sistemas cooperativos

Capítulo 2

ASC ambiente para desarrollo de sistemas cooperativos

2.1 Descripción del ambiente ASC

Existen muchas razones por las cuales se requieren herramientas de experimentación, de las más importantes podríamos decir que ayudan a verificar teorías y permiten simular sistemas ante la dificultad de construirlos completamente, además de facilitar la generación de nuevas ideas en base a los resultados obtenidos.

En los últimos años las investigaciones sobre sistemas cooperativos como un medio para la solución distribuida de problemas se han incrementado, de ahí la necesidad de contar con una herramienta que permita construir dichos sistemas a diferentes niveles de granularidad.

ASC (Ambiente para desarrollo de Sistemas Cooperativos) es una herramienta que proporciona un ***ambiente de experimentación para el modelado, la simulación, el monitoreo y la medición del desempeño de sistemas de agentes cooperativos***. Es un ambiente general que facilita la definición de agentes y acepta la modelación y depuración de diversas arquitecturas de sistemas cooperativos.

Los agentes en **ASC** poseen cinco características fundamentales. Son agentes **deliberativos** con capacidad para representar creencias y metas; y conseguir éstas últimas a través de planes. Son agentes **cooperativos** puesto que pueden realizar tareas para otros agentes por medio de compromisos. Los agentes son **interactivos** y pueden comunicarse con otros agentes, explícitamente por medio de mensajes o implícitamente usando modelos que le representen a otros agentes en su medio ambiente. Los agentes son **reactivos**, ya que pueden realizar actividades inmediatas ante la recepción de un mensaje. Finalmente, los agentes son **sociales**, porque mantienen relaciones con otros agentes en el sistema y pueden incluso formar parte de una estructura organizacional, en donde tengan asignado un rol y ciertas responsabilidades.

El ambiente de **ASC** está diseñado para permitir la simulación de la ejecución concurrente de varios agentes. Da las facilidades necesarias para crear, destruir y modificar agentes, planes y organizaciones en el sistema; permite además el monitoreo de la actividad de agentes y organizaciones; y proporciona elementos que permiten medir el desempeño de los sistemas cooperativos.

En particular **ASC** cuenta con las siguientes características:

- Permite la definición de agentes y sus atributos básicos: procesamiento, conocimiento e interacción, a través de interfaces de ventanas que facilitan su descripción.

- Los agentes en **ASC** son capaces de alcanzar metas para sí mismos y para otros agentes, por medio de planes, previo compromiso establecido.
- **ASC** permite la definición de planes como mecanismo de actuación de los agentes. Dichos planes formarán la biblioteca de planes genéricos. La representación de ellos se hace por medio de programas escritos en el lenguaje **KAL** de ***Kappa-PC*** [Int92].
- Los agentes pueden adquirir conocimiento de otros agentes durante las sesiones de trabajo y representarlo por medio de modelos.
- **ASC** proporciona diferentes funciones para que cada agente pueda manipular el conocimiento representado en los modelos; y contará además con un lenguaje de comunicación que le permita obtener el conocimiento directamente de los agentes modelados.
- La comunicación directa entre agentes es por medio de mensajes. **ASC** cuenta con un kernel básico de comunicación.
- Es posible la definición de organizaciones jerárquicas a través de representación de roles, responsabilidades y habilidades de los agentes.

En cuanto al ambiente de trabajo proporcionado por **ASC**, tenemos las siguientes características:

- **ASC** permite la creación, destrucción, modificación y almacenamiento y recuperación en disco de los elementos más importantes en el sistema (agentes, planes, organizaciones, etc.), por medio de una interfaz basada en ventanas y menús.
- **ASC** cuenta con herramientas para medir las características de un sistema cooperativo modelado, tales como el tráfico de mensajes, carga de trabajo por agente, metas conseguidas y fracasos de los agentes, tiempo empleado en la consecución de sus metas, etc
- Cuenta además con mecanismos de simulación de concurrencia.

2.2 Arquitectura de ASC

ASC está formado por tres elementos principales (fig. 2.1):

- Un **conjunto de agentes** definidos por el programador con ayuda de una interfaz interactiva.
- Un **shell** que consta de un conjunto de menús y ventanas que dan acceso a los comandos para: iniciación del sistema, manipulación de agentes, manipulación de organizaciones, manipulación de planes y llamadas al kernel.
- Un **kernel** encargado de la depuración, modificación, monitoreo y la ejecución de los agentes, planes y organizaciones definidos en el sistema, así como de las operaciones de E/S a archivos y de la comunicación entre agentes.

2.3 Agentes

En [HC93] se define un agente como una entidad que existe en un medio ambiente sobre el cual interactúa con otros agentes y con capacidad de procesamiento y conocimiento. Para la implementación de tales características se ha propuesto una arquitectura, que dada su importancia será discutida en detalle en el siguiente capítulo.

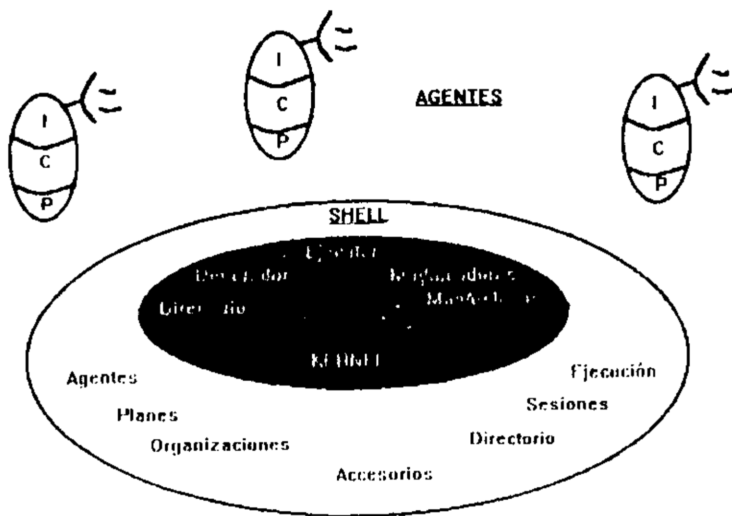

fig. 2.1. Arquitectura de ASC

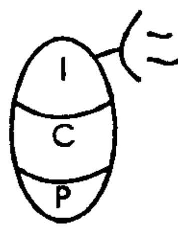

fig. 2.2 Representación de un agente
(I= Interacción, C= Conocimiento, P= Procesamiento)

2.4 Shell

El shell es un conjunto de comandos, accesados a través de menús, para la utilización de las herramientas del sistema, para monitoreo de eventos, para activación y monitoreo de los agentes y sus organizaciones.

Los comandos del shell permiten crear, destruir, mostrar o modificar un agente, un plan o una organización y sus atributos principales. Además permiten leer /almacenar de/en un archivo el código correspondiente a su estructura.

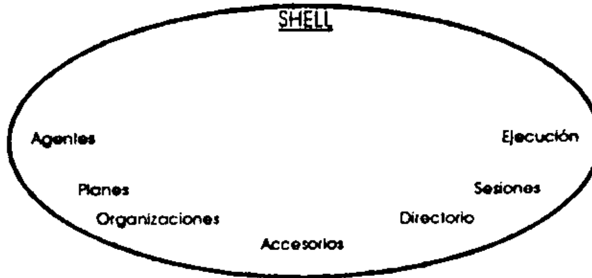

fig. 2.3 Shell

El shell proporciona comandos para mostrar directorio de agentes, y el desempeño de cada uno de ellos y del sistema en general (estadisticas de tráfico, de carga de trabajo, metas obtenidas, fracasadas, etc.). Incluye además comandos para iniciar la actividad de un agente en particular, de todos los agentes del sistema, o de todo el sistema en general. También permiten acceso al sistema que detecta puntos de ruptura.

Otro tipo de comandos del shell, permiten iniciar, reiniciar, leer o escribir en un archivo una sesión completa de trabajo. Cuenta además, con comandos adicionales, de utilidad en el sistema que incluye la creación de mensajes en el sistema y la impresión de sesiones de trabajo.

2.5 Kernel

El kernel del sistema esta formado de cinco subsistemas: Directorio, Depuradores, Ejecutor, Manejador de Archivos y Modificadores (fig. 2.4).

2.5.1 Directorio

Su función es mantener una lista actualizada de todos los agentes en el sistema y proporcionar información acerca de ellos. En base a su lista de agentes podrá proporcionar los siguientes servicios:

- Información acerca de la ubicación y clase de un agente en el sistema.
- Dar de alta o de baja a un agente.
- Proporcionar estadisticas acerca del desempeño de un agente (tráfico de mensajes, trabajos realizados, etc.).

2.5.2 Ejecutor

Encargado de la ejecución de los agentes en el sistema. Entre sus funciones se encuentran

- Ejecutar un ciclo de control en un agente determinado.
- Ejecutar un ciclo de control para todos los agentes en el sistema.
- Iniciar la ejecución cíclica de todos los agentes en el sistema, hasta que no exista más actividad en el sistema (i.e. no existan compromisos, ni mensajes pendientes).
- Permite el acceso a la ventana para establecer puntos de ruptura en el sistema.

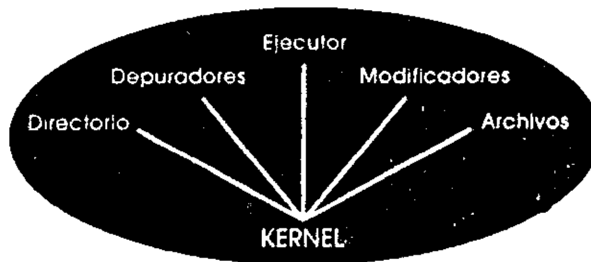

fig. 2.4 Componentes del kernel

2.5.3 Manejador de archivos

Encargado de la manipulación de archivos y la entrada y salida al sistema

- Leer e interpretar un archivo o que contiene una sesión de trabajo
- Almacenar una sesión de trabajo en un archivo en particular
- Leer e interpretar un archivo o que contiene el código de la definición de un agente, plan o estructura organizacional.
- Almacenar el código de un agente, plan o estructura organizacional en un archivo en particular

2.5.4 Depuradores

Conjunto de depuradores de los objetos principales del sistema. Se auxilian de los modificadores. Permiten crear, eliminar, renombrar y acceso a la modificación de agentes, planes y estructuras organizacionales. Se cuentan con los siguientes depuradores **Depurador de Agentes, Depurador de Planes, Depurador de Fases y Depurador de Organizaciones.**

2.5.5 Modificadores

Conjunto de modificadores de objetos principales en el sistema, auxilian en su desempeño a los depuradores. Dan acceso a cambiar los atributos particulares de los elementos más importantes en el sistema.

Modificador de Agentes. Permite la modificación de elementos pertenecientes al conocimiento de un agente en particular. Se auxilia de los **modificadores de compromisos, recursos y agentes conocidos.**
Modificador de Planes. Compuesto de funciones que dan acceso a la modificación de elementos pertenecientes a la estructura de un plan. Se auxilia del **modificador de fases**.
Modificador de Organizaciones. Permite la modificación de la estructura organizacional Facilita la asignación de roles y la definición de habilidades, recursos y metas de la organización. Se auxilia del **modificador de roles.**

2.6 Clases y objetos en el sistema

Tal como se muestra en la figura 2.5, existen elementos claves en el sistema, representados por varias clases y objetos que se describen brevemente a continuación:

2.6.1 Clases ocupadas por los AGENTES

MDAg. Modelos de los agentes conocidos por el agente actual.
RCS. Representación de los recursos del agente.
CMPS. Representación de los compromisos de un agente.
MSG. Mensajes a los agentes.
Agentes. Plantilla usada para la definición de agentes en el sistema.
Actualizador. Conjunto de funciones interpretadoras de mensajes.
Comunicador Funciones para la comunicación entre agentes.
PLANES. Plantilla usada para la representación de planes.
FASES. Plantilla usada para la representación de fases asociadas a un plan.

2.6.2 Clases ocupadas por las ORGANIZACIONES.

EstOrg. Plantilla para la definición de organizaciones jerárquicas.
Roles. Representación de los roles organizacionales.

2.6.3 Clases del KERNEL

EjAg. Clase encargada de la ejecución de agentes
MngArch. Clase del manejador de archivos.
DIR. Directorio del sistema.
Depurador. Clases encargadas de las funciones principales del depurador (modifica, crea, destruye, renombra, lee y escribe) de agentes (**DepAg**), de planes (**DepPlan**), de fases (**DepFases**) y de organizaciones (**DepOrg**).
Modificador. Clases encargadas de modificaciones a objetos principales en el sistema: agentes (**ModAg**), planes (**ModPlan**), organizaciones (**ModOrg**), roles (**ModRoles**), fases (**ModFases**), recursos (**ModRec**), compromisos (**ModCmp**) y conocidos (**ModConoc**).

2.6.4 Clases ocupadas por el SHELL

MenDepObj. Clases ocupadas del despliegue de menús asociados a los depuradores de agentes, planes, organizaciones y fases.

MenModObj. Clases encargadas del despliegue de menús correspondientes a las ventanas de los modificadores de: agentes, planes, fases, recursos, compromisos y conocidos.
MenAcc. Menú de accesorios.
MenSes. Menú de sesiones.
MenEj. Menú de ejecución de agentes.

Además, el shell está formado por los objetos asociados a las imágenes de la interfaz de usuario, las cuales se describen a detalle en el apéndice A.

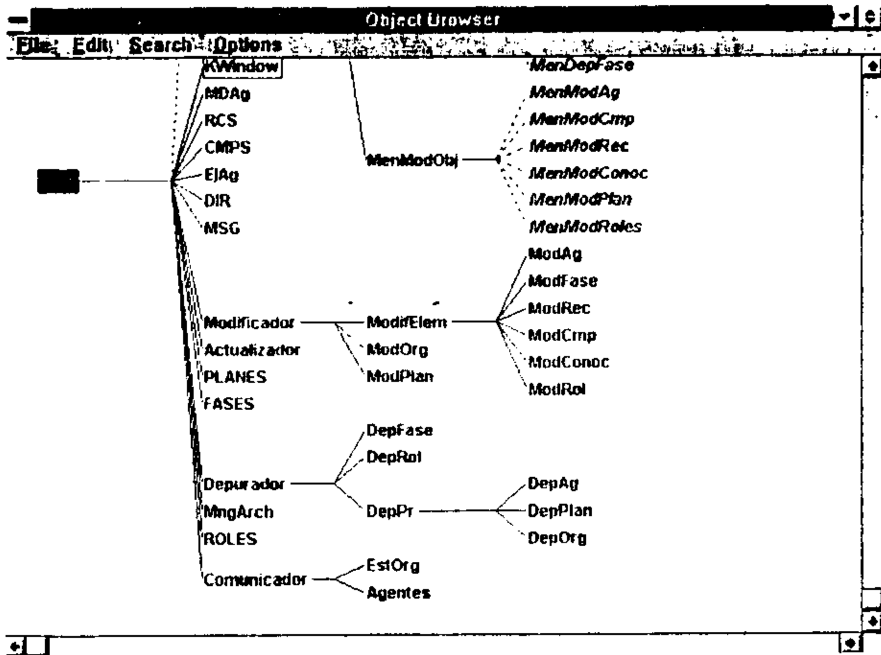

fig. 2.5 Clases y Objetos principales de ASC

Capítulo 3

Agentes

3.1 Tipos de Agentes

En los últimos años, se han efectuado investigaciones acerca de diferentes esquemas para representar a los agentes cooperativos. De ellas es posible hacer una clasificación en cuatro tipos de agentes: *agentes reactivos, agentes deliberativos, agentes interactivos* y las llamadas *arquitecturas híbridas*. [Müll96]

3.1.1 Reactivos

A mediados de los 80's emerge una corriente, dentro de la Inteligencia Artificial Distribuida, influenciada por la psicología conductista, los agentes propuestos por ellos son llamados ***agentes basados en conducta, situados o reactivos***. Estos agentes toman decisiones al tiempo de ejecución (run-time), usualmente se basan en una cantidad muy limitada de información, y en un conjunto simple de reglas acción-situación. Algunos investigadores, niegan la necesidad de cualquier representación simbólica del mundo; en su lugar los agentes reactivos toman decisiones directamente basados en sus entradas sensoriales. El diseño de las arquitecturas reactivas, está parcialmente guiada en la hipótesis de Simon [Sim81] de que la complejidad de la conducta de un agente puede ser una reflexión de la complejidad del medio ambiente en la cual el agente está operando, en lugar de un reflejo del diseño interno y complejo de un agente.

3.1.2 Deliberativos

A los agentes deliberativos también se le conoce como agentes BDI, por las siglas en inglés: ***belief, desire and intention***; (creencias, deseos e intenciones). La idea básica de las arquitecturas BDI es describir el estado del procesamiento interno de un agente por medio de un conjunto de categorías mentales, y definir una arquitectura de control por medio de la cual el agente, racionalmente, selecciona el curso de sus acciones basado en su representación del mundo. Las categorías mentales son precisamente las creencias, los deseos y las intenciones. En algunos trabajos, sin embargo, éstas han sido suplantadas por las nociones de metas y planes (goal and plans). Informalmente se tiene una descripción para tales conceptos:

Creencias. Expresan la *representación* que el agente hace de sí mismo y del estado actual de su medio ambiente.
Deseos. Son una noción abstracta que expresan las *preferencias* del agente sobre futuros estados de sí mismo y de su medio ambiente, aunque a veces esos deseos pueden ser inconsistentes.
Metas. Representan un subconjunto de *deseos consistentes* que el agente tiene como propósito.
Intenciones. Dado que un agente está acotado en recursos, no puede proponerse todas sus metas u opciones a la vez. Aún si el conjunto de metas es consistente,

frecuentemente es necesario seleccionar ciertas metas (o un conjunto de metas) para comprometerse con ellas. Así las intenciones actuales de los agentes están descritas por un conjunto de metas seleccionadas junto con sus estados de procesamiento.

Planes. **A pesar que no son un ingrediente conceptual de la teoría BDI, los planes son importantes para una implementación pragmática de las intenciones.** De hecho podríamos decir que las intenciones son planes parciales de acciones que el agente está de acuerdo en ejecutar para conseguir una meta. De esta forma es posible definir las intenciones de los agentes como el plan que actualmente tiene adoptado para conseguir sus metas.

3.1.3 Interactivos

Esta corriente mantiene su enfoque principal en la coordinación entre procesos y sobre los mecanismos de cooperación y coordinación entre agentes autónomos y deja en un segundo plano la estructura misma de los agentes. Básicamente las investigaciones se han basado en dos tipos de interacción: la *interacción explícita* realizada a través de la comunicación vía mensajes, y la *interacción implícita* que se da en base a predicciones acerca de la conducta de otros agentes. De forma breve se mencionan los principales tópicos dentro de este tipo de investigaciones:

Comunicación La comunicación juega un papel significante en la interacción entre agentes. Desde el inicio de los 90's, han existido considerables investigaciones en establecer estándares en los lenguajes de comunicación. The *Knowledge Query and manipulation Languaje (KQML)* [FF94 es un buen ejemplo de las investigaciones sobre desarrollo de lenguajes y protocolos para intercambiar información y conocimiento, basados en la teoria de los actos de habla.

Teoría de juegos e interacción entre agentes. Rosenschein y otros [GGR89] contribuyeron al análisis de la teoria de juegos e interacción entre agentes racionales. En sus trabajos plantean que para cada agente individual, se tenga una *matriz de decisión* que represente una distribución de preferencias sobre diversas alternativas que sirven para derivar decisiones racionales dada la existencia de otros agentes.

Planeación multiagente. Un área relativa a la generación y ejecución de planes para y/o por múltiples agentes.

Conflictos, Cooperación y Negociación. Estas investigaciones la resolución de conflictos y la cooperación basados en la negociación. Por ejemplo: las redes de contrato de Davis y Smith [DLC89].

3.1.4 Arquitecturas hibridas

Últimamente se ha tratado de aprovechar las ventajas que ofrecen todos las investigaciones mencionadas antes, a fin de crear agentes que soporten diversas propiedades tal como: la ***reactividad*** para reconocer eventos inesperados y reaccionar a tiempo y apropiadamente; la ***deliberación*** que les permita seleccionar sus acciones en base a los fines que quiere alcanzar tomando en cuenta los medios que tiene a su alcance; y la ***cooperación*** alcanzable a través de la interacción con otros agentes en su medio ambiente.

3.2 Descripción de los agentes ASC

Para el sistema ASC se definen a los **agentes** como: *entidades con identidad propia que perciben e inciden en su medio ambiente y son capaces de resolver problemas a pesar de no tener un conocimiento total del mismo*. Los agentes en ASC están ubicados dentro de las llamadas arquitecturas híbridas, debido a que como se mencionó anteriormente, poseen cinco características fundamentales: son **deliberativos** (con capacidad para representar creencias, metas y planes); son **cooperativos** (realizan tareas para otros agentes); son **interactivos** (se comunican entre sí); son **reactivos** (pueden realizar actividades inmediatas a la recepción de mensajes); y son **sociales** (mantienen relaciones con otros agentes). Para lograr lo anterior se ha dotado a los agentes ASC de tres atributos principales: capacidad de **interacción**, de **conocimiento** y de **procesamiento**.

3. 2.1 Interacción

Es posible entender la interacción como algún tipo de *acciones colectivas donde un agente puede llevar a cabo una acción o crear una decisión en base a la presencia o conocimiento de otros agentes en su medio ambiente*. El proceso de interacción es el que hace posible a varios agentes combinar sus esfuerzos para conseguir una meta común. La interacción puede ocurrir a través de acciones lingüísticas explícitas tal como el intercambio de mensajes, o bien a través de otro tipo de acciones en donde no es necesaria la comunicación.

La **interacción explícita**, también llamada **comunicación**, sirve para establecer un diálogo de comunicación entre dos o más agentes por medio de una serie de intercambios de mensajes con un propósito específico. Esta forma de interacción necesita un *medio físico* mediante el cual enviar los mensajes, un *protocolo de comunicación* para el intercambio confiable de información y un *lenguaje* que permita darle significado a la información recibida. Además se necesita conocimiento previo de los agentes con los cuales se va a interactuar, conocimiento por ejemplo de su ubicación, de su identidad, de sus metas, etc.

La **interacción sin comunicación** puede darse haciendo "*predicciones* y *especulaciones*" acerca de la conducta de otros agentes en el medio ambiente. Es decir un agente debe conocer que reacción esperar de otros agentes ante: la toma de una decisión, la realización de una acción o sobre la recepción de un mensaje enviado, para planear sus actividades de forma inteligente. Esta forma de interacción es una técnica de coordinación muy potente, sobre todo cuando se tienen agentes aislados o cuando el costo de la comunicación constante es muy alto.

Para que un agente pueda especular acerca de la conducta de otro, es necesario que tenga un ***modelo*** que le represente al otro, como por ejemplo cuando se conoce a un amigo lo suficiente como para saber de que manera responderá ante determinado evento sin necesidad de preguntárselo directamente. De esta forma los agentes pueden coordinar sus acciones basados en modelos, sin interacción directa a menos que se trate de comunicación para actualizar el modelo. La predicción puede ser útil para reducir la comunicación porque sólo se comunicarán los datos más necesarios. Los modelos también son útiles para evaluar la credibilidad, la utilidad y la confiabilidad de un agente.

3.2.2 Conocimiento

Un agente posee conocimiento, de sí mismo y de otros agentes. Como todo conocimiento esté debe estar en constante cambio y estos cambios se dan en base a la interacción con su medio ambiente. A grandes rasgos se dice que un agente posee:

Conocimiento de sus capacidades, el cual se necesita para tomar decisiones de asignación de tareas, para la evaluación del desempeño del agente y también para que los agentes puedan conocer que resultados son capaces de proporcionar para otros agentes.
Conocimiento de sus recursos, para saber de que disponen y que necesitan para realizar su trabajo. Esto es muy útil para tomar decisiones sobre todo cuando se tienen restricciones de tiempo real.
Conocimiento de metas y planes. La coordinación requiere que se establezcan metas y los planes para conseguirlas.
Conocimiento para comunicación: conocimiento de direcciones, medios, lenguajes, protocolos y de toda la información que será útil para la interacción explicita.
Conocimiento de su organización. Como parte de un sistema cooperativo el agente no esta sólo, sino que pertenece a alguna organización, por lo cual es importante el conocimiento de a que organización pertenece, cual es el rol desempeñado, como se relaciona con otros agentes, etc.
Conocimiento de sus responsabilidades o compromisos, el agente debe de haber sido comunicado de sus responsabilidades dentro de su organización, esto es muy útil cuando se tiene asignación adaptiva de tareas, puesto que se puede evaluar el desempeño en base a que tan bien ha cumplido con sus responsabilidades y compromisos.
Conocimiento del progreso de la solución, para detectar embotellamientos y pérdida de vida, también es útil para predecir cuando será útil el intercambio de información.

3.2.3 Procesamiento

Lo que da vida a un agente es su **capacidad de procesamiento**, es decir la capacidad de producir acciones en base a su conocimiento y a la interacción con otros agentes en su medio ambiente. Dado que las acciones de un agente pueden transformar su llamado *estado mental* [Sho90] (creencias, decisiones, capacidades, compromisos, metas, etc.) o producir acciones que van a activar las acciones de otros agentes, un agente puede tener varios objetivos a conseguir:

- Lograr comunicación con otros agentes vía mensajes.
- Interpretar los mensajes recibidos.
- Efectuar las acciones activadas ante la recepción de un mensaje.
- Establecer tanto sus metas particulares como las metas de su organización, activando los planes para conseguirlas.
- Realizar los trabajos para los cuales se ha comprometido.
- Informar sobre la consecución de metas o los avances para conseguirlas.

Las acciones activadas por los mensajes recibidos pueden ser desde requerimientos de información o actualización de creencias y la aceptación de compromisos, hasta la firma de contratos de trabajo.

Con el intercambio de mensajes, el agente establece compromisos y las metas para cumplirlos. Para alcanzar esas metas genera planes que son una serie de actividades o tareas a realizarse bajo determinadas circunstancias. Cuando los planes se completan y se alcanzan las metas es necesario informar, si es el caso, a los agentes con los cuales se establecieron compromisos.

En base a lo anterior se plantea un modelo de agentes basado en la Programación Orientada a Agentes de Yoav Shoham [Sho90], el ambiente para multiagentes MACE de Gasser, Braganza y Herman [GBH97], el lenguaje de comunicación KQML de Yannis Labrou y Tim Finnin [LaFi93], así como también en el trabajo sobre Sistemas Cooperativos de Horacio Carvajal y otros [HC93]. Para el diseño de los agentes se utiliza la metodología orientado a objetos, definiendo objetos, clases y subclases con sus respectivos métodos y atributos [Kat96].

3.3 Arquitectura de los agentes ASC

Los agentes ASC son entidades cuya arquitectura consta de seis elementos (fig. 3.1.).

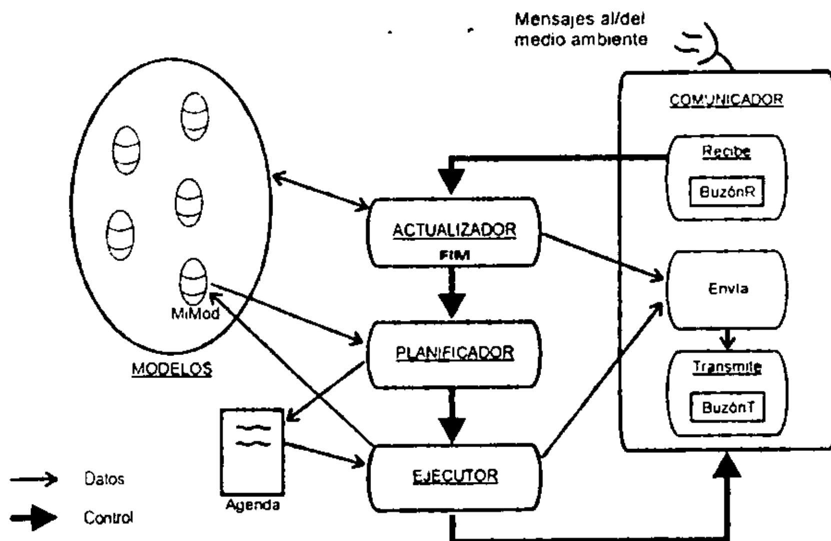

fig.3.1 Arquitectura de un agente

Modelos de agentes. Representan el conocimiento de sí mismo (**MiMod**) y de otros agentes del sistema que le son conocidos. Estos modelos mantienen el estado mental de un agente (creencias, metas y planes), su identidad y su conocimiento organizacional.

Agenda. Contiene las fases de los planes que están listas para su ejecución en el presente ciclo.
Comunicador. Parte activa de un agente que se encarga de enviar, recibir, validar y clasificar los mensajes usados en la comunicación explícita. Está compuesto de tres funciones: **Envia** , **Recibe** y **Transmite**.
Actualizador. Es el encargado de la interpretación de mensajes, los cuales pueden abarcar desde la actualización del estado mental del agente y sus agentes conocidos, hasta la creación de nuevos compromisos. Está compuesto por un conjunto de funciones interpretadoras de mensaje (**FIM**), quienes serán activadas de acuerdo al tipo de mensaje recibido.
Planificador. Realiza la búsqueda en la biblioteca de planes genéricos, de uno de ellos que permita alcanzar las metas fijadas en los compromisos adquiridos por el agente También es el responsable de generar la agenda de actividades del agente seleccionando para ello los métodos o las fases de los planes cuyas precondiciones estén dadas.
Ejecutor. Encargado de la ejecución de las actividades contenidas en la agenda. Además será su responsabilidad checar la consecución de metas y generar los mensajes correspondientes hacia el exterior , es decir, avisar a sus clientes por compromisos terminados.

3.3.1 Flujo de control

La ejecución, de todos los agentes ASC, sigue una secuencia determinada en cada ciclo de control (NOTA se ha puesto en negritas los elementos de la arquitectura involucrados en cada paso)

1. ***Comunicador.Recibe***, lee y valida los mensajes recibidos por el agente contenidos en el **BuzonR**. Clasifica los mensajes y los transfiere al Actualizador.
2. ***Actualizador***, activa la **FIM** que procesa el mensaje.
3. Los mensajes correspondientes a requerimientos de información acerca del agente o actualizaciones de modelos tales como: **Preg**, **Dile**, **PuedoHacer**, **PuedeHacer**, **Niega**, etc., se procesan y responden inmediatamente (***Comunicador.Envia***).
4. Los mensajes correspondientes a solicitudes de tareas (**Comp**), generan una meta y un compromiso, por parte del agente actual, para conseguirla así como un mensaje de aceptación para el agente que la solicito. (***Comunicador. Envia***).
5. ***Planificador.Búsqueda***. En esta etapa a cada compromiso aún sin iniciar y cuyo tiempo de procesamiento se cumpla, se le busca un plan en la biblioteca de planes genéricos que pueda ayudar a conseguir la meta. Se crea entonces, una instancia de dicho plan y se ejecuta la función de iniciación del mismo.
6. ***Planificador.Scheduler***, localiza las fases, de los planes actuales, cuyas precondiciones estén dadas y las coloca en la **Agenda**.
7. ***Ejecutor***, ejecuta todas las actividades contenidas en la **Agenda**.
8. Al vaciar la agenda revisa si existe alguna meta lograda en esta etapa, para ello verifica aquellos planes cuyas fases estén terminadas completamente, en ese caso ejecuta la función de termino de un plan y actualiza el estado mental del agente, además de informar al agente cliente sobre la culminación exitosa del compromiso (***Comunicador. Envia***)
9. **Comunicador.Transmite**, transmite todos los mensajes, generados durante este ciclo, que se localizan en el **BuzonT** del agente.

3.4 Modelos en los agentes ASC

En un modelo se representa **el conocimiento de los agentes**, cada modelo estará compuesto de 3 secciones principales de atributos y de algunas otras secciones que dependerán del dominio particular de cada agente. Además se cuenta con un conjunto de funciones aplicables a los mismos.

3.4.1 Atributos de los modelos

Los atributos del agente están agrupados en tres secciones: **Identidad** que mantiene los atributos que identifican al agente; **estado mental** que describen al agente mismo; y **organizacional** aquellos atributos que lo identifican como miembro de una sociedad u organización

Dada la importancia de la parte organizacional del modelo, esta será descrita en un capítulo posterior en donde se describe con detalle el comportamiento organizacional de un agente

IDENTIDAD

Nombre: Nombre del agente que está siendo modelado.

Dirección: Localización del agente modelado (usada en la comunicación con el agente).

ESTADO MENTAL

Capacidades. Lista que representa las habilidades propias del agente, y que pueden hacerse coincidir con las metas de otros agentes, esto es útil para la asignación de tareas

Compromisos: Representan las actividades a las cuales está comprometido el agente, es decir lo que el agente modelado quiere alcanzar para si mismo, para algún agente en particular o para su organización. De esta forma, en esta estructura se representa a las metas y a los planes actualmente adoptados por el agente, es la representación simbólica de *las intenciones del agente*, de ahi su importancia. Se simbolizan usando n_adas con la siguiente información:

(Agente, Meta, Plan, Resultados, Estado, TFirm, TMax, TInic, TConc)

significa que a más tardar en el tiempo ***TMax*** el agente actual tiene que alcanzar una ***Meta*** especifica para el agente ***Agente***; ***Plan*** es el nombre de la instancia del plan asignado a la meta; ***Resultados*** es una lista donde se colocarán los resultados parciales o finales que se pretendan enviar al agente contrator cuando se consiga la meta. Se anexan, además, datos que permitirán una evaluación del desempeño del agente en base a compromisos cumplidos en tiempo y forma: ***TFirma*** representa la hora en que se estableció en compromiso; ***TInic*** la hora en que se inicio el plan para conseguir la meta; ***TConc*** la hora en que se alcanzo la meta; finalmente, ***Estado*** cuyo valor dará una medida del avance de la solución, sus posibles valores son: ***NoIniciado, EnProceso, Terminado, Suspendido*** (cuando el agente contrator le indique que no termine el compromiso) o ***Cancelado*** (cuando el compromiso nunca llegue a realizarse por carecer de planes o métodos especificos). Las *metas actuales* del agente serán aquellas contenidas en los compromisos, cuyos estados estén ***EnProceso***. De igual forma las *metas conseguidas* serán aquellas contenidas en los

compromisos cuyos estados sean *Terminado* y las *metas fracasadas* aquellas cuyos compromisos hayan sido *Suspendido* o *Cancelado*.

Recursos: Aunque en realidad no forman parte del estado mental de un agente, por comodidad se ha decidido incluir a los recursos de hardware, software, asesores, material, etc., como parte de las creencias del agente. Su representación esta dada por:

(nombre_recurso, tipo, cantidad)

ORGANIZACIONAL.

Org: Nombre de la organización de la cual es parte.

Relación: Vínculo que el agente modelado tiene con el agente actual. Puede ser algunas de las mencionadas abajo o bien algunas otras definidas por el programador.

- *Mío:* El modelo es del agente mismo.
- *MiTrab:* El modelo es de un agente miembro de la organización administrada por este agente.
- *JefeOrg:* El modelo es del agente administrador de la organización a la cual pertenece este agente.
- *Colega:* El modelo es de un agente de la organización de la cual también es parte.
- *Conocido:* El modelo es de un conocido. Usado cuando no importa tanto la relación establecida.

Roles: Papel desempeñado por el agente modelado dentro de la organización y dependerá del dominio particular del trabajo cooperativo.

3.4.2 Funciones para los modelos

Sobre un modelo se pueden realizar las siguientes funciones predefinidas. y donde *atributo* es cualquiera de los mencionados arriba:

Crea: Crea un modelo para un agente conocido y le asocia una relación. Regresa el nombre del modelo.

Crea(nombre_agente, relación);

Destruye: Destruye el modelo asociado a un agente.

Destruye(nombre_mod);

Actualiza: Cambia el valor de un atributo por uno nuevo.

Actualiza(atributo, valor);

Agrega: Agrega un valor a la lista de valores de un atributo especifico.

Agrega(atributo, valor);

Informa: Proporciona el o los valores contenidos en un atributo.

Informa(atributo, valor);

Modifica: Modifica interactivamente (con valores proporcionados por el usuario) los valores asociados al modelo.

Modifica();

Borra: Elimina el valor de un atributo.

Borra(atributo, valor);

Es?: Pregunta si el atributo del modelo tiene un valor específico.

Es?(atributo, valor);

Recursos?: Proporciona una lista con todos los recursos de un agente conocido registrados en su modelo.

Recursos();

TengoRecursos: Regresa **TRUE** en caso de que en el modelo de un agente conocido se tengan registrados los recursos especificados.

TengoRecursos(recursos);

Mostrar: Despliega en las ventanas asociadas al agente el valor de cada atributo del modelo.

Mostrar();

3.5 El planificador

La conducta específica de un agente **ASC** puede ser determinada en dos formas. Usando funciones interpretadoras de mensajes (**FIM**) cuando se requiere que el agente responda con una acción especifica ante la llegada de un mensaje, o bien usando **planes**, para aquellas acciones comprometidas que requieran su ejecución en una o varias etapas

Las **FIMs**, se agregan directamente al actualizador cada vez que se agreguen nuevos mensajes a reconocer por el agente. Los **planes** son usados para tareas más complejas que requieren ejecución por fases, en este caso es necesario programar cada fase del plan y luego ir construyendo el plan en base a estas acciones atómicas. Es importante mencionar que no existe una generación de planes en el sistema, asi que la construcción de planes se refiere más bien a una acción directa de los usuarios del ambiente.

3.5.1. Representación de los planes

Un **PLAN** representa el camino que el agente debe tomar para conseguir sus metas. Un plan posee: un **Nombre**, que debe ser igual al de la meta a conseguir; una lista que contenga los nombres de las **Fases** que constituyen el plan (puede ser vacia); una función de **Iniciar** que será ejecutada siempre que el plan se asigne a una meta; y finalmente una función de **Terminar** para que sea ejecutada al conseguir la meta. Estas dos funciones son útiles para declarar variables a usarse dentro del plan (i.e., en alguna de sus fases) o bien para ejecutar algún procedimiento en particular al iniciar o finalizar el plan.

PLANES.**Nombre**
PLANES.**Iniciar**
PLANES.**Fases** (Fase1, Fase2, ..., Fasen)
PLANES.**Terminar**

FASE.**Nombre**
FASE.**Precond**
FASE.**Acción**

fig. 3.2 Representación de PLANES y FASES

Una **FASE** la definimos como una estructura procedural que consta de dos elementos: las ***Precond**iciones* y la ***Acción***. Las Precondiciones son una función que regresa **TRUE** en caso de que las precondiciones de la fase estén dadas para que se pueda ejecutar la ***Acción***, en caso contrario regresará **FALSE**. Es importante mencionar que las precondiciones pueden ser funciones que regresen **TRUE** o **FALSE**, o bien relaciones lógicas, o bien una combinación de ambas. La ejecución de las fases sólo depende de las precondiciones , por lo cual es posible ejecutar más de una fase de un plan en un mismo ciclo de control.

Los planes son *independientes* de los agentes. Es decir pueden ser usados por uno o más agentes al mismo tiempo, pues sólo están representando las formas de conseguir un objetivo. Los planes representan en general clases de planes, y cada vez que es seleccionado uno de ellos para conseguir la meta señalada en un compromiso, se genera una instancia del plan a usarse para cumplir con el compromiso. Por lo anterior es necesario que tanto las funciones de ***Iniciar*** y ***Terminar*** de los planes, como las ***Precondiciones*** y ***Acciones*** de las fases del plan, reciban como parámetro de entrada el agente para el cual esta siendo ejecutado el plan y la instancia del plan correspondiente.

3.5.2 Función del planificador

La función del planificador está ligada con la representación de las intenciones del agente cuya implementación es en base a compromisos representados por la siguiente estructura:

(Agente, Meta, Plan, Resultados, Estado, TFirm, TMax, TInic, TConc)

Agente: Nombre del agente con quien se tiene el compromiso (cliente).
Metas: Nombre de la tarea comprometida.
Plan: Nombre de la instancia asociada al plan para conseguir la meta.
Resultados: Lista de resultados generados por este compromiso. Esta lista será enviada al agente cliente cuando el compromiso este terminado. También, este campo puede usarse para pasar información inicial al plan que conseguirá la meta.
Estado: Estado actual del compromiso.
TFirm: Hora en que se firmó el compromiso.
TMax: Hora limite para finalizar el compromiso.
TInic: Hora de inicio del compromiso. Si al establecer el compromiso **TInic es -1** significa que el cliente no especifica hora determinada para ejecutar el compromiso y puede realizarse en cualquier momento dentro del límite de **Tmax**. Si **Tinic >0** significa que el agente cliente pide una hora fija para iniciar el compromiso.
TConc: Hora de finalización del compromiso).

El planificador está dividido en dos etapas: **Búsqueda** y **Scheduler**. En la primera de ellas, el objetivo principal es la búsqueda e instanciación de planes para los nuevos compromisos adquiridos por el agente cuyo tiempo de iniciación se satisfaga. Para esto trata de encontrar un plan en la biblioteca de planes que satisfaga la meta establecida, una vez encontrado procede a generar una instancia de dicho plan y se lo asigna al compromiso. La parte del **Scheduler** tiene como objetivo formar la **Agenda** actual del agente con las fases, de todos los planes actuales, cuyas precondiciones estén dadas.

```
/*************** METHOD: Planificador ***************/
MakeMethod( Agentes, Planificador, [],
/* Búsqueda */
{  /* Revisando por nuevos compromisos cuyo
      tiempo de iniciación se cumpla */
 EnumList(Self:MiMod:Compromisos, x,
  If (( x:Estado #= NoIniciado) And
      ((x:TInic == -1) Or
       (x:TInic == EjAg:Reloj)))
  Then  /* La meta requiere un plan */
    {    Self:xmod = NULL,
```

```
EnumSubClasses(PLANES, y,
If ( y#=x:Meta )
Then   Self:xmod = y);
 If (Not ( Null?(Self:xmod)))
Then   /* Creando instancia del plan */
   {Let [ instplan SendMessage(Self:xmod, Crea, INST)]
       { /* Actualizando compromiso */
              instplan:Comp = x;
              x:Plan = instplan,
              SendMessage(instplan, Iniciar, Self);
              x:Estado = EnProceso;
              x:TInic = EjAg Reloj,    }}
Else     /* Cancelando compromiso, no lo puede hacer el agente */
     {      x.Estado = Cancelado;
            x:TInic  = EjAg.Reloj;
            ClearList(Self:xlisttmp);
            AppendToList(Self:xlisttmp, x:Meta);
            SendMessage(Self, Envia, x:Agente, LoSiento , 0,Self:xlisttmp),
     };    },  });
/* Scheduler */
/* Generando la agenda actual*/
/*Revisa compromisos, En Proceso, ( planes actuales)*/
 EnumList(Self:MiMod Compromisos, x,
  If (x.Estado #   EnProceso)
  Then    /* Checa precondiciones *
  {    EnumList(x Plan,Fases, y,
       If (SendMessage(y, Precond, x Plan, Self))
       Then
       {   /* Prepara accion a ejecutar */
         AppendToList(Self Agenda,y),
         AppendToList(Self AgendList, x Plan),
         RemoveFromList(x Plan Fases,y),
       }),   });  }),
```

3.6. El ejecutor

El ejecutor realiza dos funciones principales: la primera de ellas es la ***ejecución*** de todas las actividades contenidas en la **Agenda** del agente; su segunda función es ***monitorear*** las metas, para verificar si alguna de ellas se ha conseguido, en cuyo caso deberá actualizar los parámetros de evaluación en el compromiso que generó la meta e informar al agente cliente el cumplimiento del mismo.

```
************** METHOD  Ejecutor  **************
MakeMethod( Agentes, Ejecutor, {},
{/* Cumpliendo agenda */
Let [ nacc  LengthList(Self Agenda)]
 For n From 1 To nacc  Do
  { Let [ fase GetNthElem(Self:Agenda,n)]
        [ inst GetNthElem(Self AgendList,n)]
    { SendMessage( fase, Accion, inst, Self);
      Self:Acciones +=1;    };  },
ClearList(Self Agenda);
ClearList(Self:AgendList);
/*Monitoreando por metas alcanzadas */
EnumList(Self:MiMod:Compromisos, y,
   If ((y:Estado #= EnProceso) And
      ( (LengthList(y:Plan:Fases) == 0)))
   Then
```

```
{ y:Estado = Terminado;
  y:TConc = EjAg:Reloj;
  SendMessage(y:Plan, Terminar, Self);
  DeleteInstance(y:Plan);
  MakeSlot(Self:lista2);
  SetSlotOption(Self:lista2, MULTIPLE);
  ClearList(Self:lista2);
  AppendToList(Self:lista2, y:Meta, y:TConc, y:Result);
  SendMessage(Self, Envia, y:Agente, CompTer, 0,Self:lista2);
  DeleteSlot(Self:lista2);
     }); } };
```

3.7 Comunicación entre agentes ASC

El desarrollo de la forma de comunicación está basado en algunas suposiciones acerca de los agentes en el sistema.

- *cada agente posee una identidad y nunca estarán registrados en el sistema dos agentes con el mismo nombre.*
- *los agentes son honestos y por lo tanto responsables de los mensajes enviados,*
- *los agentes son crédulos y adoptan los creencias de otros como suyas;*
- *los agentes son confiados y contestan todas las preguntas;*
- *los agentes sólo responderán a preguntas sobré el mismo y nunca responderá a nombre de otro,*
- *los agentes responderán a cualquier mensaje recibido, para el cual se esté esperando una respuesta,*
- *los agentes son cooperativos y adoptan las metas de otros que son consistentes con sus capacidades.*
- *finalmente se asume que los agentes no envían mensajes que perjudiquen el trabajo de otros.*

Para poder establecer una comunicación explícita, es necesario que los agentes estén de acuerdo en tres cosas: ***transporte***, para definir como enviarán y recibirán los mensajes; ***lenguaje***, que permita darle un formato y significado a cada mensaje (protocolos de comunicación), ***política***, políticas de conversación que permitan darle una estructura y una secuencia lógica a cada conversación establecida.

De estos asuntos están encargados tanto ***el comunicador*** como ***el actualizador***, ellos dependen mucho del tipo de lenguaje y protocolo de comunicación usada. Para este trabajo se ha generado un comunicador y actualizador basados en el lenguaje **KQML** (Knowledge Query & Manipulation Languaje) [FF94], mismo que puede aumentarse de acuerdo a las características de cada sistema.

3.7.1 El comunicador

El **comunicador** es el encargado del ***transporte de los mensajes***. Los mensajes pueden ser enviados a todos los agentes en el sistema (*broadcast)*, o a un grupo específico formado a partir de una relación entre ellos, o a un agente en particular. Para el *transporte* dispone de dos recipientes:

BuzonR: Contiene los mensajes recibidos por el agente.
BuzonT: Contiene los mensajes a enviarse a otros agentes.

El comunicador está formado por tres funciones:

Envía: Función activada explícitamente por el agente cuando desea enviar un mensaje. Es la encargada de crear los mensajes, darles el formato adecuado y localizar las direcciones de los agentes a los cuales se envía el mensaje. Si el mensaje está dirigido a **Todos**, busca las direcciones de todos los agentes dados de alta en el sistema haciendo uso del directorio del *kernel* del sistema. Los grupos de agentes se definen en base a su relación con el agente actual, de modo que, por ejemplo, puede enviar mensajes a sus trabajadores (**MiTrab**), a su jefe (**JefeOrg**), a sus colegas (**Colega**) o simplemente a sus conocidos (**Conocido**), en este caso las direcciones de los agentes se buscan en los modelos de sus conocidos. Una vez creados los mensajes se depositan en el **BuzonT** para su posterior transmisión.

Envia (agente, tipo_mensaje, id_respuesta, lista_contenido);

```
/************** METHOD: Envia **************/
/* "Forma el mensaje y lo deja en el buzon de transmisión." */
MakeMethod( Agentes, Envia, [agente tipo idres contenido ],
{
 If (agente #= Todos)        /*Broadcast*/
 Then
  {     /* Enviando mensaje a todos */
  AppendToList(Self:Todos, DIR:agentes);
  RemoveFromList(Self:Todos, Self);
  RemoveFromList(Self:Todos, USER);
  EnumList( Self:Todos, x,  SendMessage(Self, EnviaUno,
          x, tipo, Self:nmen, idres, contenido));
  Self:nmen +=1;   }
 Else
  {  /* Enviando mensaje a un  grupo  definido por una relación
        entre los agentes:  MiTrab, JefeOrg, Colega, Conocido*/
  If( Member ? (DepAg.Relacion, agente))      /*Grupo*/
  Then
   {   /* Encontrando a los miembros del grupo */
   EnumList(SendMessage(Self, Selecciona, Relacion.agente) ,
    x, SendMessage(Self, EnviaUno, x, tipo,Self:nmen, idres, contenido));   }
  Else    /* Enviando a un agente */       /*Directo*/
    SendMessage(Self, EnviaUno, agente,
              tipo,Self:nmen, idres, contenido);
 }; } );
/************** METHOD: EnviaUno **************.
/* "Envia un solo mensaje.
Regresa identificador del mensaje." */
MakeMethod( Agentes, EnviaUno, [agente tipo idmen idres contenido ],
{ Let [nmen MSG:nmen]
      [ modelo SendMessage(MSG, Crea)]
  { SetValue(modelo, Receptor, agente);
   ClearList(modelo:Contenido);
   AppendToList(modelo:Contenido, contenido);
   SetValue(modelo, Tipo, tipo);
   SetValue(modelo, Remitente, Self);
   SetValue(modelo, RespondaCon, idmen);
   SetValue(modelo, EnRespA, idres);
   AppendToList(Self:BuzonT, modelo);
  nmen;
  };      } );
```

Transmite: Esta función es activada automáticamente en cada ciclo de control del agente. Es la encargada de transportar los mensajes desde el **BuzonT** del agente actual, hacia los **BuzonR** de los agentes receptores. Verifica, con ayuda del *kernel* del sistema, que el agente destino exista. Si el agente no existe deberá dejar un mensaje de error en el buzón de recepción del agente emisor.

```
/*************** METHOD: Transmite ***************/
MakeMethod( Agentes, Transmite, [],
{ Let [ nmen  LengthList(Self.BuzonT)]
  For n From 1 To nmen  Do
   Let [men GetNthElem(Self:BuzonT,1)]
    {  RemoveFromList(Self:BuzonT, men),
/* Verifica que exista el agente */
      If (SendMessage(DIR,Existe, men:Receptor) )
      Then  /*Transmite mensaje */
       { AppendToList(men Receptor:BuzonR, men),
        If Not( men:Receptor #= Self)
        Then Self envios+=1;  }
      Else
       { Advertencia("Agente desconocido "),
        SendMessage(MSG,Destruye,men);
       },    };  } ).
```

Recibe: Función activada automáticamente en cada ciclo de control del agente. Todos los mensajes que llegan al agente, son recibidos y encolados en el **BuzonR** de acuerdo al orden de llegada. Una vez validado cada mensaje en el **BuzonR** se activará el **Actualizador** a fin de procesar cada mensaje en particular.

```
/*************** METHOD  Recibe ***************/
/* "Recibe los mensajes  los valida y los transfiere al actualizador" */
MakeMethod( Agentes, Recibe, [],
{ Let [ nmen  LengthList(Self.BuzonR)]
    For n From 1 To nmen  Do
    Let [men GetNthElem(Self:BuzonR,1)]
     {  RemoveFromList(Self:BuzonR, men),
       If (IsAKindOf?(men,MSG))
       Then
       {  Monitor("Agente "#men:Receptor#
            "recibe mensaje "# men:Tipo#
            " enviado por "# men.Remitente );
          Self:recibos += 1.
/* Llamando al actualizador */
          DelegateMessage(Self, Actualizador, Procesa, men);
          SendMessage(MSG,Destruye,men);};
     }; } );
```

NOTA: Para la comunicación con el exterior del sistema, se ha definido un *agente de comunicación* llamado **USER**, que representa al usuario del sistema y su única función es transmitir mensajes desde la interfaz de usuario a un agente o viceversa. Este agente, a diferencia de los demás, carece de actualizador, de planificador y de ejecutor. La función de recepción ha sido modificada también y sólo despliega los mensajes recibidos en la pantalla **Monitor** de la interfaz de usuario.

3.7.2 El actualizador

El **actualizador es el encargado de la interpretación de los mensajes** que llegan al agente. Ésta interpretación depende del tipo de mensaje recibido, así que está formado por un conjunto de **funciones interpretadoras de mensajes (FIM)** que, como su nombre lo indica, serán activadas por los mensajes y pueden abarcar desde la actualización del estado mental del agente o sus conocidos, hasta la creación de nuevos compromisos. El actualizador, por lo tanto, depende directamente del protocolo y del ***lenguaje*** de comunicación empleado. Sin embargo se ha definido un lenguaje que abarca un conjunto pequeño de **FIM's** basadas en KQML [FF94].

Los **mensajes** para tal lenguaje tienen el siguiente formato:

(Tipo,	Tipo de acción esperada ante la recepción del mensaje
Remitente,	Nombre del agente que envia el mensaje.
EnRespA,	Etiqueta esperada en la réplica del mensaje. (usada cuando el mensaje es una contestación del remitente a un mensaje previo).
Receptor,	Nombre del agente receptor del mensaje.
RespondaCon,	Etiqueta necesaria para incluir en un mensaje respuesta del receptor.
Contenido)	Contenido del mensaje (lista de valores).

Según el análisis de los desarrolladores de KQML se tienen 5 tipos de mensajes en una conversación

- **Directivas.** Ordenan algo (comandos, requerimientos o sugerencias).
- ***Asertivas.*** Afirman algo.
- **Comisivas.** Comprometen al remitente en una acción (compromisos).
- **Declarativas.** Declaran la ocurrencia de un hecho (formación de hechos).
- **Expresivas.** Sentimientos y actitudes.

De estos mensajes, las oraciones declarativas están implícitas en las acciones de los agentes cuando, durante su procesamiento cambian o declaran nuevos hechos. Las oraciones expresivas no son de interés por el momento, así que se concentrarán esfuerzos en las oraciones asertivas, directivas y comisivas. En este sistema se han implementado solo algunas **FIM** básicas y son descritas en la siguiente sección.

NOTA DSC describe brevemente al mensaje. FMT es el formato del mensaje. ACC describe las acciones tomadas por el agente receptor ante la llegada del mensaje RSP indica los mensajes enviados como respuesta al agente remitente. men:Contenido se refiere al formato de la lista con el contenido del mensaje; **atributo** es cualquier atributo definido en el modelo de un agente.

3.7.3 Funciones interpretadoras de mensaje (FIM)

Directivas

CualEs
DSC: Solicita información del, o de los, valores de un atributo del agente receptor.
FMT: *men:Contenido { atributo }*

ACC: **Toma los valores de los atributos directamente del modelo (MIMod) del agente receptor y los envía al remitente.**

RSP: **ValorEs o Error en caso de atributo no conocido.**

Preg

DSC: Pregunta si el atributo del agente receptor tiene un valor específico.

FMT: *men.Contenido { atributo, valor}*

ACC: Revisa si el valor proporcionado es igual al del modelo del agente receptor. (**MIMod**)

RSP: **Dile** si es verdad; **Niega** en caso de que el atributo no tenga tal valor. **Error** en caso de atributo desconocido.

PuedeHacer

DSC: Pregunta si el agente receptor puede hacer una tarea y (opcionalmente) si cuenta con los recursos suficientes.

FMT: *men.Contenido { tarea, [recurso1, recurso2, ...] }*

ACC: Verifica si entre sus capacidades está la de hacer la tarea específica. Verifica además, en su caso, si cuenta con los recursos suficientes para ello. Si las dos situaciones anteriores son verdaderas, regresa mensaje **PuedoHacer** y una lista de los recursos con los que cuenta para realizar la tarea. La lista de recursos regresada, siempre es un subconjunto de la lista de recursos enviados. Posiblemente estos recursos no sean suficientes y por lo tanto requerirá que, en caso de establecerse un compromiso, le envien también los recursos necesarios. Cuando el agente receptor no pueda hacer la tarea, envía mensaje **NoPuedo**.

RSP: **PuedoHacer, NoPuedo.**

Asertivas

ValorEs

DSC: Recibe los valores del atributo del agente remitente.

FMT: *men.Contenido { atributo valor1, valor2, ... }*

ACC: Actualiza el modelo que el agente receptor tiene del remitente.

RSP: Ninguna.

Dile

DSC: Indica que el atributo del agente remitente tiene el valor indicado.

FMT: *men:Contenido { atributo, valor}*

ACC: Actualiza el modelo que el agente receptor tiene del remitente.

RSP: Ninguna.

Niega

DSC: Indica que es falso el valor del atributo del agente remitente.

FMT: *men:Contenido { atributo, valor}*

ACC: Actualiza el modelo que el agente receptor tiene del remitente.

RSP: Ninguna.

Hecho

DSC: Indica que ha sido aceptado por al agente remitente, el compromiso solicitado.

FMT: *men:Contenido { tarea, tmax, tinic, tfirma }*

ACC: El agente receptor actualizará el compromiso ubicado en el modelo del agente remitente poniendo el tiempo de firma y estado EnProceso. (Esto puede usarse para controlar a los agentes proveedores de algún servicio para el agente receptor.)

RSP: Ninguna.

LoSiento

DSC: Indica que no ha sido aceptado por al agente remitente, el compromiso solicitado. También este tipo de mensaje es enviado para indicar que un compromiso EnProceso fue imposible de continuar.

FMT *men:Contenido { tarea, tmax, tinic, tfirma }*

ACC El agente receptor actualizará el compromiso ubicado en el modelo del agente remitente poniendo el estado como **Cancelado**. Además deberá actualizar el modelo del agente remitente para evitar en un futuro intentar hacer compromisos de ese tipo. Este mensaje proporciona actualización de creencias para el agente receptor.

RSP: Ninguna.

PuedoHacer

DSC Indica que el agente remitente puede hacer la tarea y que cuenta con los recursos enumerados, los cuales pueden no ser suficientes para la ejecución de la tarea.

FMT *men Contenido { tarea, recursos}*

ACC Si el agente receptor no conoce al agente remitente, se crea un modelo para el nuevo conocido. En cualquier caso, siempre se actualiza el modelo del agente remitente con la información enviada: incrementando el conocimiento que se tiene de las capacidades y los recursos del agente remitente.

RSP Ninguna

NoPuedo

DSC Indica que el agente remitente no puede hacer la tarea.

FMT *men Contenido { tarea, recursos}*

ACC Si el agente remitente es un conocido, actualiza su modelo eliminando la información que se creía cierta con respecto a la tarea y los recursos.

RSP Ninguna

Error

DSC Indica que el agente remitente recibió un mensaje inválido.

FMT *men Contenido { tipo, ContenidoOriginal }*

ACC Ninguna

RSP Ninguna

CompTer

DSC Indica que el agente remitente ha terminado un compromiso pendiente con el agente receptor y le envia los resultados.

FMT *men Contenido {tarea, Tconc, Resultados}*
Resultados { recurso, tipo, cantidad }

ACC Actualiza el modelo del agente remitente con la información enviada, marcando el compromiso como **Terminado**. Este conocimiento extra permitirá al agente receptor medir en un futuro el desempeño del agente remitente. Si el compromiso terminado generó resultados, éstos se toman como nuevos recursos para el agente receptor.

RSP: Ninguna.

Comisivas

Comp

DSC El agente remitente solicita una tarea al agente receptor. Puede de manera opcional enviarle recursos adicionales (si lo considera necesario).

FMT *men Contenido { tarea, tmax, tinic, [recursos]}*

ACC: Si el agente receptor tiene la capacidad para hacer la tarea. Crea un compromiso con el agente emisor para realizar la *tarea* en el tiempo inicial *tinic* y teniendo como máximo *tmax*. Si el agente remitente anexa recursos, los acepta para formar parte de los suyos. Responde afirmativamente a la orden, regresando el tiempo de firma del compromiso. Si el agente no puede comprometerse responde con una negativa.

RSP: **Hecho** si el compromiso se acepta, o **LoSiento** si es rechazado.

CancelaComp

DSC: El agente remitente ordena al agente receptor que suspenda la realización de un compromiso previamente pactado.

FMT. *men.Contenido (tarea)*

ACC: Actualiza la lista de compromisos del agente receptor, marca compromiso como **Suspendido** y pone la fecha de cancelación del compromiso como la fecha actual.

RSP: Ninguna.

3.7.4 Conversaciones entre agentes

La ***politica de conversación*** se refiere a una secuencia de actuaciones para que la conversación tenga un significado. En este sentido el actualizador debe clasificar los mensajes y decidir cuales inician una conversación y en que parte de la conversación son usados los otros mensajes. Dichas politicas de conversación están implicitamente representadas en la interpretación que se le da a cada mensaje recibido, tal como puede observarse en la *red de comunicación* siguiente:

Mensaje enviado	**Mensajes esperados**
Comp	Hecho, CompTer, LoSiento
Hecho	CancelaComp
CualEs	ValorEs, Error
Preg	Dile, Niega, Error
PuedeHacer	PuedoHacer, NoPuedo

Además las politicas de conversación definen una serie de restricciones sobre los mensajes enviados, por ejemplo, para enviar los mensajes **Comp, CualEs y Preg** es necesario que el agente remitente conozca a los agentes receptores (i.e. tenga un modelo de ellos), en cambio para el mensaje **PuedeHacer** no necesariamente significa que se conoce al agente que puede contestar esta pregunta. Lo mismo podriamos decir de los mensajes **Hecho, ValorEs, Dile, Niega, PuedoHacer y NoPuedo**, en donde el modelo del agente remitente no es de interés para el modelo receptor.

Esta forma de interacción entre los agentes les permite un aprendizaje dinámico acerca de otros agentes en su medio ambiente, el cual se refleja en los modelos de los agentes que va conociendo durante toda su actividad en el sistema. Veamos algunos ejemplos de conversaciones entre agentes con intercambio de conocimiento.

En la fig. 3.3, se representa a un agente **Ag1**, enviando un mensaje en *broadcast* para saber quien puede hacer la **Tarea1** que necesita los recursos **R1 y R3**. En este estado inicial, **Ag1** sólo posee conocimiento de sí mismo.

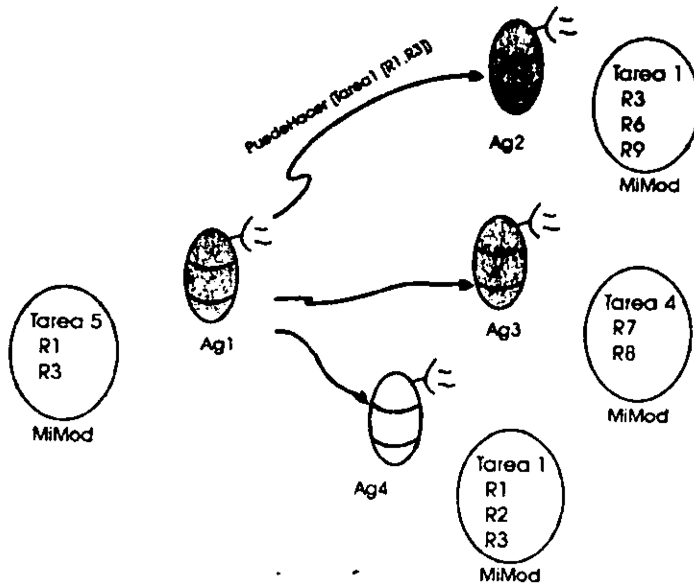

fig. 3.3 Estado inicial de un agente enviando un mensaje en *broadcast*

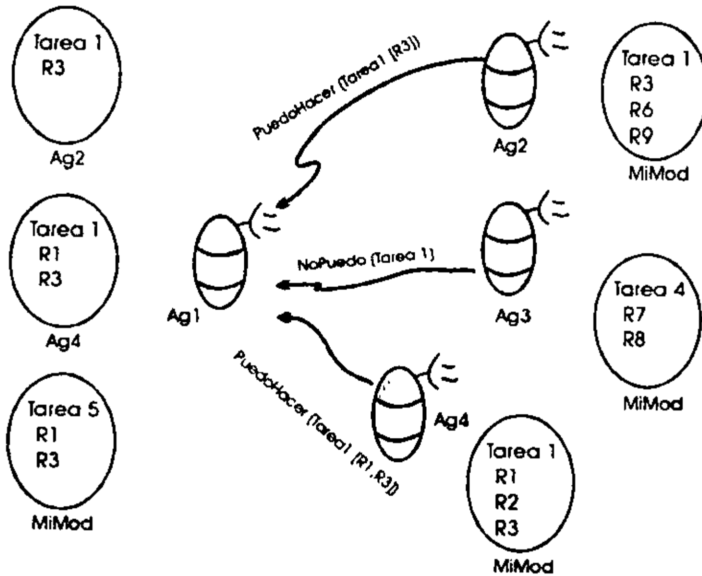

fig. 3.4 Creando modelos de agentes

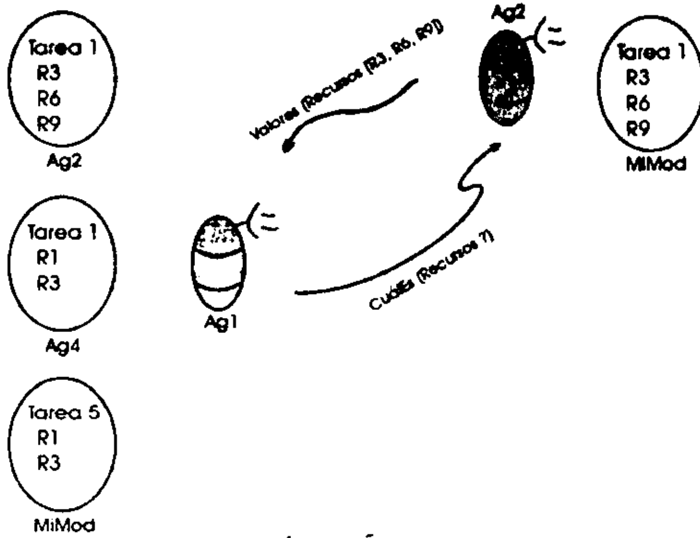

fig. 3.5 Actualización explícita de creencias

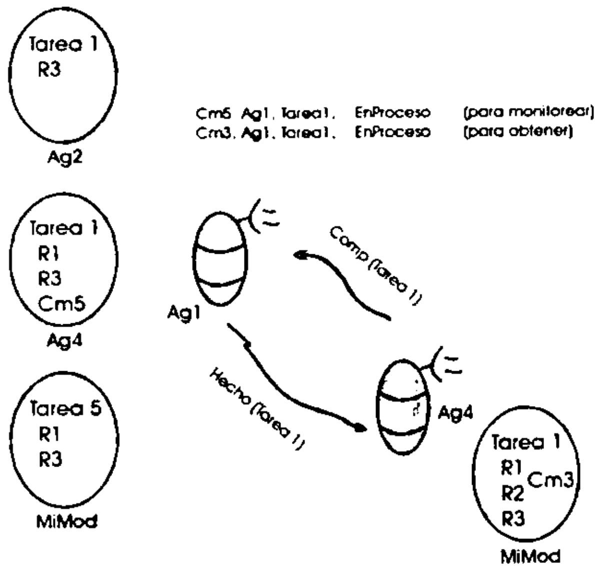

fig. 3.6 Actualización implícita de creencias (ante éxitos)

Como resultado del **mensaje anterior**, en la fig. 3.4 se puede observar la respuesta de los agentes receptores, en este caso recibe respuesta afirmativa de los agentes **Ag2** y **Ag4**, además recibe información adicional acerca de algunos recursos que poseen dichos agentes. Con esta información el agente **Ag1**, se forma un modelo de otros agentes en su medio ambiente. Dado que el **Ag3** respondió negativamente, este agente no es de interés para el agente **Ag1** y no tiene sentido formarse un modelo del mismo.

También existen mensajes que permiten actualización de creencias acerca de los agentes conocidos, a petición expresa del agente actual, véase fig. 3.5. En este caso **Ag1** **Pregunta** a **Ag2** por sus recursos disponibles, **Ag2** envía la información requerida y **Ag1** actualiza el modelo que tiene de él.

En contraparte, la fig. 3.6 representa una actualización implícita de creencias acerca de los agentes conocidos. **Ag1** cree que **Ag4** puede hacer la **Tarea1** y trata de establecer un **Compromiso**, **Ag4** contesta afirmativamente y empieza su trabajo. En este caso se actualizan los compromisos en ambos lados del contrato. Para **Ag4** el compromiso formará parte de su estado mental y para **Ag1** será parte del estado mental de su conocido, mismo que le puede servir en el momento actual para monitorear la actividad del agente comprometido y a futuro para una evaluación del comportamiento de **Ag4**.

Ahora suponga que el estado mental de los agentes ha cambiado y **Ag4** no puede más efectuar la **Tarea1**, véase fig. 3.7. **Ag1** intenta hacer un compromiso y recibe respuesta negativa, ante tal situación cambia el estado mental en el modelo que **Ag1** tiene de **Ag4**. Borra de entre sus capacidades a la **Tarea1** y marca el compromiso como **Cancelado**.

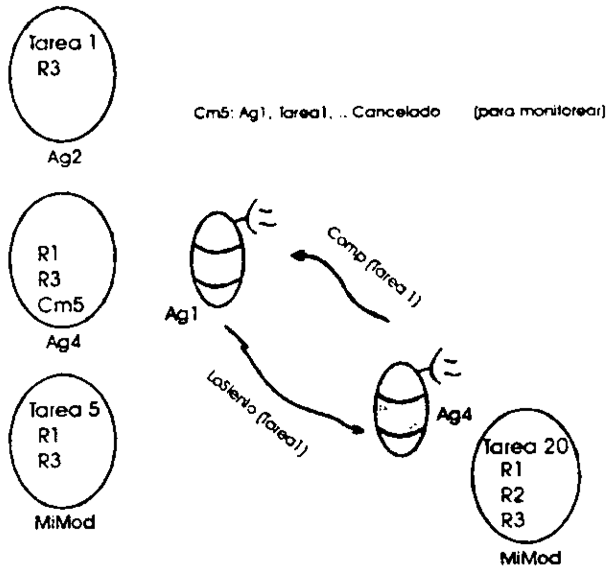

fig. 3.7 Actualización implícita de creencias (ante fallas)

Resumiendo, se puede decir que la comunicación entre agentes permite incrementar el conocimiento acerca de otros agentes en el medio ambiente.

3.8 Desempeño del agente

Para medir el desempeño de un agente se tienen varios parámetros a considerar.

Metas Conseguidas	Aquellas cuyos compromisos estén **Terminados**
Metas Fracasadas	Aquellas cuyos compromisos estén **Cancelados** o **Suspendidos**.
Metas Actuales	Aquellas cuyos compromisos estén **EnProceso**.
envios	Total de mensajes enviados.
recibos	Total de mensajes recibidos.
Actividades	Total de actividades atómicas ejecutadas por el agente.

Además se puede hacer un análisis más detallado del desempeño si se compara el tiempo de conclusión del compromiso (**Tconc**) contra el tiempo máximo permitido por al agente cliente para terminar el compromiso (**Tmax**). Otro parámetro más para medir desempeño es la comparación entre el tiempo en que se firmó el compromiso (**TFirm**) contra el tiempo en que se inicio el compromiso (**Tinic**).Todos estos parámetros estarán visibles al usuario del sistema desde la ventana del directorio del sistema.

3.9 Funciones en los agentes ASC

Sobre los agentes se tienen definidas las siguientes operaciones, las cuales pueden usarse en las actividades que definen el comportamiento o conducta de los agentes:

Crea: Crea un agente.
Crea(nombre);

Destruye: Destruye un agente.
Destruye(nombre);

ElimConoc: Elimina un modelo asociado a un conocido.
ElimConoc(conocido);

AgregaConoc: Crea un modelo asociado a un nuevo conocido del agente.
AgregaConoc (conocido, relación);

Selecciona: Proporciona una lista con los nombres de sus conocidos cuyo atributo tenga un valor especifico.
Selecciona(atributo, valor);

Performance?: Regresa una lista cuyos elementos representan: número de metas alcanzadas, número de metas en proceso, número de metas suspendidas, y número de metas canceladas respectivamente.
Performance?();

Informa: Informa el valor del atributo de un conocido.
Informa(conocido, atributo);

Modelo?: Regresa el nombre del modelo asociado a uno de sus conocidos.
Modelo?(conocido);

Conocidos?: Proporciona una lista con los nombres de todos sus conocidos.
Conocidos?();

MostrarConoc: Despliega en ventanas el nombre de sus conocidos y su relación con cada uno de ellos.

MostrarConoc();

Mostrar: Despliega en ventanas asociadas al agente, sus atributos principales.

Mostrar();

Actualiza: Actualiza el modelo de un conocido.

Actualiza(conocido, atributo, valor);

Agrega: Agrega un valor al atributo del modelo de un conocido

Agrega(conocido, atributo, valor);

Modifica: Modifica interactivamente (desde el usuario) el modelo asociado a un conocido.

Modifica(conocido);

Borra: Borra el valor de un atributo del modelo asociado a un conocido.

Borra(conocido, atributo, valor);

Capitulo 4

Organizaciones

Una de las estrategias heurísticas para superar la *racionalidad limitada* es la **organización**. Una organización considerada como el establecimiento de una estructura de interacción en donde los agentes mantienen ciertas responsabilidades [HC93]. Más aún se podría considerar una organización como un sistema de subunidades que procesan e intercambian información, en un intento por conseguir una o más metas [Bla96]. Los principales problemas de una organización son los de *asignación eficiente de tareas* y de *recursos;* y la distribución de la información sobre un conjunto de agentes.

Las organizaciones, entonces, pueden verse como una cuádrupla (**Ags**, **MetOrg**, **Habilidades**, **EstOrg**) donde: **Ags** es un conjunto de componentes funcionales, en este caso, los agentes; **MetOrg** es el conjunto de metas de la organización, que dependen del dominio particular de acción; **Habilidades** son las capacidades propias de la organización que pueden definirse como la suma de las habilidades de sus miembros, más todas aquellas que puedan formarse como una composición de éstas. Finalmente **EstOrg** es la estructura organizacional que determina los roles y las relaciones entre los agentes, y puede ser, desde grupos y jerarquías, hasta organizaciones de mercados.

La estructura organizacional establece los **roles** y las relaciones de **autoridad** (*para saber quién ejecuta qué acciones y cúando*), las relaciones de **comunicación** (*para saber a quién informar acerca de eventos o cambios de estado entre agentes*) y el **conocimiento** (*en base a las responsabilidades de cada agente*) de cada miembro de la organización. Un rol determina el papel desempeñado por el agente dentro de su organización y son instanciados a cada agente o suborganización para delimitar sus responsabilidades.

Las organizaciones tienen, además, algunas de las propiedades adaptivas encontradas en las conductas individuales de los agentes, tales como la búsqueda entre una variedad de alternativas para tomar una decisión adecuada, para diagnosticar síntomas de problemas existentes, y para aprender de sus experiencias [Bla96]. Estas propiedades son proporcionadas a la organización por todos sus miembros funcionales.

4.1 Tipos de organizaciones

Varios investigadores han estudiado las características de algunos tipos de organización. Fox [Fox89] identifica las siguientes formas organizacionales, mismas que van surgiendo una de otra conforme la complejidad del problema lo requiere:

Un simple agente. Ejecuta todas las tareas, recibe toda la información, es suficiente cuando se tienen todos los recursos para alcanzar la meta.
Grupos de agentes. En ellos existe una división de tareas, la asignación de tareas es de acuerdo a las capacidades de cada miembro del grupo, eligiendo al más adecuado para

ello. La información llega a todo el grupo y las decisiones son tomadas de mutuo acuerdo. Su limitación es que si crece el grupo, la toma de decisiones colectivas se vuelve muy costosa.

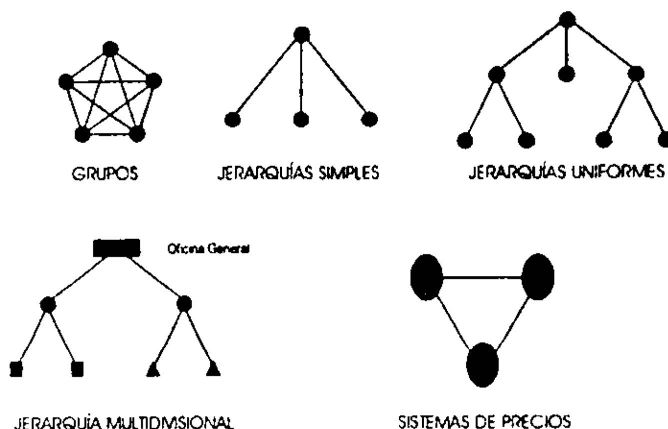

fig. 4.1 Estructuras organizacionales (FOX)

Jerarquía simple. Es una estructura de dos niveles: gerente (o manejador) y trabajadores. El manejador es quien realiza la distribución y la asignación de tareas. La información completa sólo llega al manejador quien la distribuye según convenga a los intereses de la organización Los trabajadores sólo obedecen órdenes del manejador. Esta estructura es deficiente cuando el manejador es incapaz de procesar toda la información
Jerarquía uniforme Es una estructura de varios niveles. Cada nivel actúa como un filtro de información. Las decisiones se propagan desde el nivel más alto de la jerarquia. El inconveniente se presenta cuando la organización crece y se tienen varias metas a conseguir que generan una competencia por los recursos de la organización.
Jerarquía multidivisional. Existe una división o área por cada producto a producir (meta a conseguir). Cada división controla solo las acciones que le llevarán a conseguir su meta. La planeación de la distribución de tareas se realiza en una oficina general que controla todas las divisiones. Nuevamente cuando la organización crece se tienen problemas con la coordinación y el procesamiento de la información.
Sistemas de precios. Son organizaciones disjuntas disponibles a ofrecer un servicio o producto. Las acciones se llevan a cabo en base a negociaciones. La comunicación se da en base a contratos establecidos.

Malone [Mal89] en cambio ha identificado las siguientes formas de organización:

Jerarquía de productos. Existen divisiones separadas por cada producto. Cada división consta de un manejador y de trabajadores especializados en distintas áreas. El manejador decide que trabajador ejecuta la tarea. La comunicación sólo se da entre manejador-trabajador.

Jerarquías funcionales. Los trabajadores de tipo similar (i.e. que ejecutan la misma tarea) están en un mismo departamento controlado por un manejador funcional quien decide quien de ellos ejecuta la tarea. Existe un ejecutivo de oficina que decide que tarea se necesita ejecutar. La interacción se da entre EjecutivoOficina ⇔Manejador ⇔Trabajador

Mercados descentralizados. Todos los proveedores de servicios están en contacto con todos los agentes a los que le pueden solicitar sus servicios.

Mercado centralizado. Los agentes que requieren servicio, contratan éste a través de un intermediario quien de antemano conoce quien puede proporcionar el servicio.

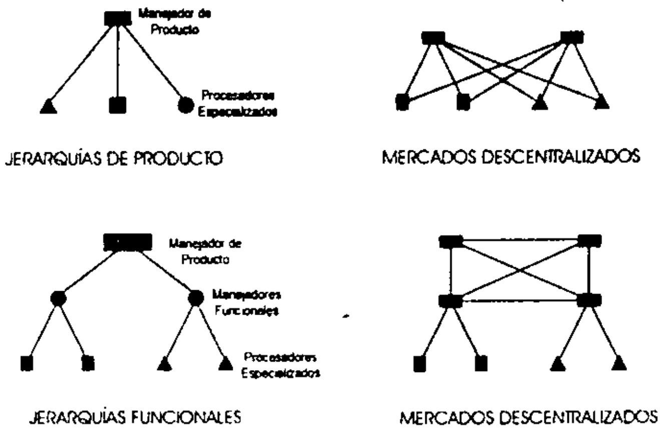

fig. 4.2 Estructuras organizacionales (MALONE)

4.2 Estructuras Organizacionales

Para propósitos de implementación y reconociendo las caracteristicas similares entre las organizaciones descritas en el punto anterior, se podrían agrupar y tener básicamente como estructuras organizacionales a: los **grupos**, las **jerarquías** y los **mercados**. Identificando los componentes de sus estructuras organizacionales se tendria lo siguiente

<u>GRUPOS</u>

Roles: Colegas y líder (en este caso la función del líder es: ser el vocero del grupo, es decir el encargado de la comunicación entre el medio ambiente y el grupo)

Comunicación: Los mensajes son enviados y recibidos por todos los agentes o bien a través del líder:

Medio ambiente ⇔ grupo
Medio ambiente ⇔ lider ⇔ grupo

Autoridad: Decisiones de mutuo acuerdo (por consenso o votación). Existe una división de tareas; y la asignación es generalmente, al más capacitado de todos.

Conocimiento: Todos los agentes se conocen entre si.

JERARQUÍAS
Roles: Manejador y trabajadores.
Comunicación: La información completa de los niveles superiores o del medio ambiente, sólo llega al manejador y sólo él está capacitado para responder a nombre de la organización. Los trabajadores únicamente reciben información de su manejador y a él informan de la consecución de sus metas.

Medio ambiente ⇔ manejador ⇔ trabajadores

cabe recalcar aquí, que la comunicación entre trabajadores y otros agentes en el medio ambiente sigue siendo posible, pero no es de interés para la organización. Además, el medio ambiente en este caso, puede a su vez ser una organización mayor de la cual es parte.
Autoridad: El manejador decide, en base a la información que posee, quién ejecuta la tarea y cuándo. Los trabajadores reciben órdenes directas del manejador y hacen lo que él les ordena.
Conocimiento: El manejador conoce a todos sus trabajadores, los trabajadores sólo conocen a su manejador.

MERCADOS
Roles: Conocidos.
Comunicación: Se da entre organizaciones proveedoras de servicios.

organizaciones ⇔ organizaciones

Autoridad: La asignación de tareas es en base a negociaciones.
Conocimiento: Pueden conocerse a través de la interacción entre ellos.

Las jerarquías uniformes, entonces, pueden definirse recursivamente si se ven como una jerarquía simple, donde sus trabajadores son a su vez otras jerarquías. Las jerarquías de producto se toman como jerarquías simples donde sus trabajadores tienen diferentes capacidades, en cambio las funcionales serán aquellas donde los trabajadores tiene las mismas capacidades.

Este trabajo sólo se concentra en la representación de organizaciones jerárquicas de dos niveles (jerarquías simples) y por el momento, por razones de alcance, no se trabajará en la representación de estructuras más complejas como los mercados centralizados o los sistemas de precios.

4.3 Representado organizaciones jerárquicas

Para representar las organizaciones necesitamos los 4 elementos principales: agentes, metas, habilidades y la estructura organizacional. La representación de los agentes ha sido descrita en capítulos anteriores.

Las metas y habilidades forman parte del conocimiento organizacional y dado que también se disponen de recursos de la organización, conviene representar todo este conocimiento en un **modelo** de la organización (**MIMod**), de la misma forma que los agentes representan el conocimiento de sí mismos. Es importante mencionar que esta información está disponible a todos los miembros de la organización, pero en el caso de las organizaciones jerárquicas, el único miembro con capacidades de modificación de este modelo (estado de los compromisos, recursos disponibles, habilidades de la

organización, etc.) es el manejador, quien tiene a su cargo el control de toda la organización.

Al igual que en los agentes, es necesario un comunicador que se encargue de la recepción y transmisión de mensajes a y para la organización. Sin embargo la organización por sí misma es algo abstracto, por lo cual debe existir un miembro de la organización quien se encargue de procesar los mensajes que entran o salen de la organización, en el caso de las organizaciones jerárquicas será el agente manejador. Las funciones **Recibe** y **Transmite** se ejecutarán automáticamente de forma indirecta desde el agente manejador.

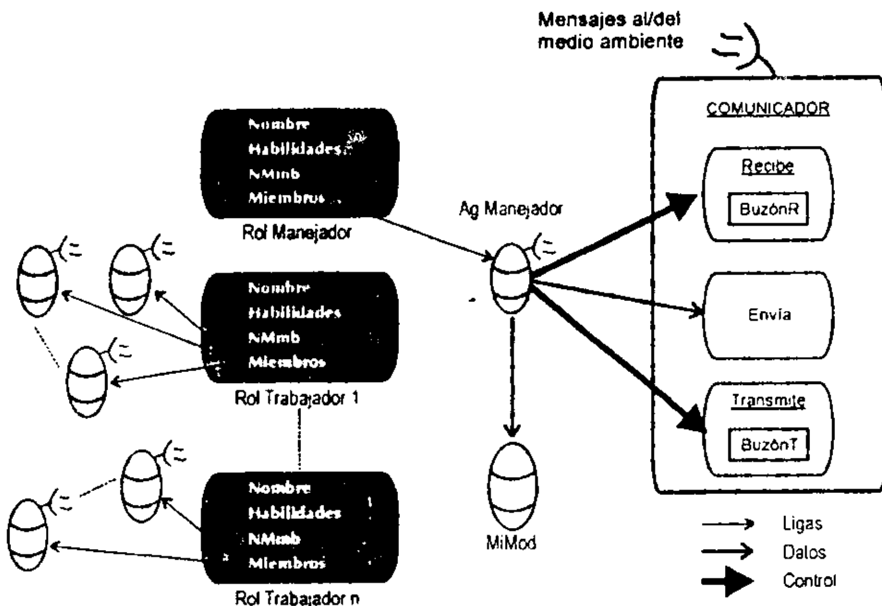

fig. 4.3 Arquitectura de las organizaciones jerárquicas

Con estas consideraciones iniciales podremos plantear una arquitectura para las organizaciones jerárquicas, tal como se muestra en la figura 4.3. La estructura organizacional (**EstOrg**) entonces, nos permite representar los roles y todos los elementos necesarios para el funcionamiento de la organización.

Identidad. Nombre de la organización.
MiMod.Contiene el nombre del modelo que representa el conocimiento de la organización (sus habilidades, sus recursos, sus compromisos, sus roles, etc.)
Manejador Rol asociado al manejador de la organización.

Trabajadores. Roles asignados a los trabajadores de la organización.
Comunicador. Encargado de la recepción, envío y transmisión de mensajes de la organización, controlado siempre por el agente manejador.

A su vez cada rol posee los siguientes atributos, representados en la clase **ROLES**:

Nombre. Identidad asociada al rol.
Habilidades. Habilidades que deben cumplir los miembros que quieran ocupar ese rol.
NMmb. Número de unidades requeridas para el rol.
Miembros. Unidades instanciadas al rol.

4.3.1 Instanciando organizaciones jerárquicas

El proceso de instanciación de una organización está dividido en varias etapas. La *primera* de ella consiste en buscar de entre todos los agentes u organizaciones en el medio ambiente, aquellos que reúnan las habilidades requeridas para desempeñar los diferentes roles dentro de la organización.

```
/*Instancia organización */
MakeMethod( EstOrg, Instancia, [],
{ MakeSlot(Self, listtmp),
  SetSlotOption(Self:listtmp, MULTIPLE),
/* Buscando un manejador para la organización */
If(Not (Null?(Self:Manejador))
 Then SendMessage(Self:Manejador, Instancia, Self),
If(LengthList(Self:Trabajadores)!= 0)
Then
/* Buscando trabajadores para la organización */
 EnumList(Self:Trabajadores, x,
     SendMessage(x, Instancia, Self));

/*Instancia roles*/
MakeMethod( ROLES, Instancia, [org ],
{ MakeSlot (Self, miemb);
  SetSlotOption(Self, miemb, VALUE_TYPE, NUMBER);
  MakeSlot(Self, listtmp);
  SetSlotOption(Self, listtmp, MULTIPLE);
/* buscando en organizaciones y agentes */
  GetInstanceList(EstOrg, Self:listtmp),
  RemoveFromList(Self:listtmp, org);
  AppendToList (Self:listtmp, DIR:agentes);
  Self:miemb = Self:NMmb - LengthList(Self:Miembros) ,
  While( ((Self:miemb) > 0) And    (LengthList(Self:listtmp) != 0)  )
  { Let [ y  GetNthElem(Self:listtmp,1)]
   {  If((SubConj?(Self:Habilidades, y:MiMod:Capacidades)) )
     Then
      { /* Instancia el agente a este rol */
       y:MiMod:Org = org;
       If (Not (Member?(y:MiMod:Roles, Self:Nombre)))
       Then
          AppendToList( y:MiMod:Roles, Self:Nombre);
       AppendToList(Self:Miembros, y);
       Self:miemb = Self:miemb -1; };
     RemoveFromList(Self:listtmp, y);      };    };
 DeleteSlot(Self:listtmp),
 DeleteSlot(Self:miemb);} );
```

La *segunda etapa* consiste en dotar de conocimiento a los agentes instanciados, según el rol desempeñado; en este caso el agente manejador deberá poseer conocimiento de todos sus trabajadores, por lo que se le dota de modelos que los representen. Los agentes trabajadores, en contraparte, sólo necesitan conocer a su jefe, de quien reciben órdenes y a quien envían resultados.

```
/* Actualizando conocimiento en la Organización */
Let [ mang GetNthElem(Self.Manejador Miembros, 1) ]
{ EnumList(Self:Trabajadores,x,
   { ClearList(Self:listtmp);
     AppendToList(Self listtmp, x Miembros);
     EnumList(Self:listtmp, y,
       { Let [modtrab SendMessage(MDAg, Crea, y, MiTrab)]
            [modjefe SendMessage(MDAg, Crea, mang, Jefe)]
          (SendMessage(y:MiMod, Copia, modtrab);
           AppendToList(mang.Modelos, modtrab);
           SendMessage(mang MiMod, Copia, modjefe);
           AppendToList(y Modelos, modjefe),
          },      }),   }),
```

La *tercera etapa* del proceso de instanciación, consiste en asignar al agente manejador de un patrón de comportamiento acorde a su rol dentro de la organización. Para esto se agregan dos funciones: **FDAR** que se ejecutará antes de la función **Recibe**; y **FDAT** para ejecutarse después de la función **Transmite** del *agente manejador*.

La función **FDAR** le permitirá al *agente manejador*. leer y procesar los mensajes que llegaron a la organización; y tomar los compromisos y recursos de la organización como suyos, será el responsable de su consecución. Es decir, las metas de la organización serán un subconjunto de las metas del agente manejador y los recursos estarán a su disposición para que pueda distribuirlos entre sus trabajadores.

```
/*Definiendo patron de comportamiento del manejador */
MakeMethod(mang, FDAR,[agent] ,
  {
/* Leyendo mensajes de la organización */
 SendMessage(agent:MiMod.Org, Recibe);
/* Tomando compromisos y recursos de la organización */
  EnumList(agent MiMod:Org:MiMod:Compromisos, y,
   { If (y:Estado == NoIniciado)
     Then
      { y Estado = EnProceso,
       Let [comp SendMessage(CMPS, Crea)]
         {  SendMessage(comp, Actualiza, agent, y:Meta,
              y:TMax, y:TFirm, y TInic),
            SendMessage(agent:MiMod, Agrega, Compromisos, comp);
         }      }     } );
 EnumList(agent.MiMod Org:MiMod:Recursos, y,
  SendMessage(agent.MiMod, Agrega, Recursos, y) ),
 ClearList(agent.MiMod Org MiMod.Recursos),
  })
```

La función **FDAT** le agrega comportamiento al *agente manejador* a fin de que pueda revisar las metas organizacionales que se alcanzaron y emitir el informe correspondiente

a nombre de la organización. También esta función realiza la transmisión de todos los mensajes de la organización.

```
/* Revisa por metas cumplidas en la Organización */
 MakeMethod(mang, FDAT,[agent] ,
  {   EnumList(agent:MiMod:Org.MiMod:Compromisos, y,
   { If (y:Estado #= EnProceso)
     Then
      EnumList(agent:MiMod:Compromisos, z,
      { If ( (z:Meta #= y.Meta) And
          (z:Estado #= Terminado) )
       Then
       {y Estado = Terminado,
        y TConc = EjAg:Reloj,
        MakeSlot(agent:lista2),
        SetSlotOption(agent lista2, MULTIPLE),
        ClearList(agent.lista2),
        AppendToList(agent.lista2, y:Meta, y TConc, y Result);
        SendMessage(agent MiMod:Org, Envia, y:Agente, CompTer, 0,agent lista2);
        DeleteSlot(agent:lista2),
       }        }),    } );
   SendMessage(agent:MiMod:Org, Transmite);
    }), },
 DeleteSlot(Self:listtmp),
 } ).
```

Nuevamente el mecanismo de actuación de los agentes deberá incluir un plan. En este caso el agente manejador deberá disponer de un plan que le permita asignar tareas en base a los roles de sus trabajadores. En el capítulo cinco se describirá a detalle un ejemplo de este tipo de organizaciones jerárquicas.

Existe mucho más trabajo en cuanto a organizaciones se refiere. Las últimas investigaciones tratan a las organizaciones como adaptivas a los cambios en el medio ambiente, organizaciones que aprenden de sus experiencias propias y de la experiencia de otras organizaciones [SHH91] [Mas93] y [MR92] [Bla96]. Sin embargo este es un buen avance para la representación y el estudio de las mismas.

Capítulo 5

Ejemplos

Para probar el Sistema ASC, se han desarrollado como ejemplos, un *sistema de aldeas de producción*, una *red de contratos* y una *organización jerárquica simple*. Los dos primeros ejemplos se ejecutaron dos veces, la primera vez suponiendo que los agentes desconocen su medio ambiente y la segunda cuando los agentes han adquirido conocimiento de otros agentes que puede serles útil en la consecución de sus metas.

5.1 Aldeas de producción

5.1.1 Descripción

Se tiene un conjunto de 7 agentes cada uno con diferentes habilidades. Se necesita generar un esquema de producción de materias primas y de productos compuestos en donde exista cooperación para que cada agente consiga sus propias metas. (Los Pi representan a los productos compuestos y las Mi a las materias primas)

Agentes	Habilidades	Recursos	Conocidos	Meta
Ag1	HacerM1, ConstruyeP1	--	--	HacerP1
Ag2	HacerM2, ConstruyeP2	--	--	HacerP2
Ag3	HacerM4, ConstruyeP3	--	--	HacerP3
Ag4	HacerM5, ConstruyeP4	--	--	HacerP4
Ag5	HacerM1, ConstruyeP5	--	--	HacerP5
Ag6	HacerM3, ConstruyeP2	--	--	HacerP2
Ag7	HacerM3, ConstruyeP1	--	--	HacerP1

fig. 5.1 Estado mental inicial de los agentes

Para que un agente pueda efectuar la producción de productos compuestos es necesario que cuente con la materia prima necesaria. Así por ejemplo, según lo muestra la tabla siguiente, para producir P3 es necesario que el agente cuente con la materia prima M4 y M5.

ProdCompuesto	MateriaPrima
P1	M1, M2
P2	M1, M2, M4
P3	M4, M5
P4	M1, M5
P5	M2, M3

fig. 5.2 Productos compuestos y materias primas

Si el agente puede producir su materia prima no necesitará de la cooperación de otros, sin embargo en este ejemplo se han seleccionado agentes que necesitan la ayuda de otros. Así por ejemplo el Ag5, para construir P5 necesita como materias prima a M2 y M3, como él no puede producir ninguna de las dos necesitará la ayuda de Ag2 para que le produzca M2, y de Ag6 o Ag7 para que le proporcionen la materia prima M3.

5.1.2 Implementación

Al iniciar el sistema los agentes sólo están dotados de conocimiento de sus habilidades, no poseen ningún recurso, ni conoce a alguien en su medio ambiente. Sin embargo cada uno de ellos tienen una meta a conseguir, representada como un compromiso consigo mismo.

```
MakeInstance( Ag1, Agentes );
MakeInstance( Md7, MDAg ),            /* Modelo de si mismo */
Md7:Nombre = Ag1;
Md7:Relacion = Mio,
Md7:Capacidades = (HacerP1, ProduceM1) ;
Md7:Compromisos = ( Ag1, HacerP1, ..),         /*Meta del agente */
```

La producción de una materia prima se obtiene a partir de un plan con un único evento.

```
MakeClass( PLANES, ProducirMat );
MakeMethod( ProduceM1, Iniciar, [agent ],
   { /* Anexa a los resultados la materia producida */
    AppendToList(Self:Comp:Result, M1, material, 1),
    Monitor("Produciendo M1");  } ),
```

La producción de un producto compuesto requiere de un plan más complejo, por ejemplo para la producción de P1 es necesario el siguiente plan.

```
MakeClass( HacerP1, HacerPro ),
SetValue( HacerP1:Fases, ObtieneM1, ObtieneM2, ConstruyeP1 ),
```

es decir, primero se obtienen las materias primas M1 y M2 y enseguida se construye el producto compuesto P1. A su vez obtener una materia prima no es un proceso sencillo si el agente puede producirla, genera una meta (compromiso consigo mismo) para hacerlo; en caso contrario, su meta será conseguir un proveedor para ella. Las precondiciones para esta fase serán que no exista la materia prima como recurso

```
MakeInstance( ObtieneM1, ObtenerMat );
MakeMethod( ObtieneM1, Precond, [plan agent ],
   { ClearList(plan:listtmp),
     AppendToList(plan:listtmp, M1);
     If (SendMessage(agent:MiMod, TengoRecursos,plan:listtmp))
     Then  FALSE
      Else  TRUE,  } ),
MakeMethod( ObtieneM1, Accion, [plan agent ],
   { Monitor(" Obteniendo M1");
   If ( Member?(agent:MiMod:Capacidades, ProduceM1 ))
      Then          /* Se compromete a producir la materia prima */
      { Monitor("Se compromete consigo mismo"),
        Let [ comp   SendMessage(CMPS, Crea)]
        {  SendMessage(comp, Actualiza, agent, ProduceM1,
            EjAg:Reloj+20, EjAg:Reloj,-1);
           SendMessage(agent:MiMod, Agrega, Compromisos,comp); }  }
      Else          /* Se compromete a buscar un proveedor */
      { Monitor("Se compromete a buscar quien ");
       Let [ comp   SendMessage(CMPS, Crea)]
       { SendMessage(comp, Actualiza, agent, ObtenerProv,
         EjAg:Reloj+20, EjAg:Reloj,-1);
         SendMessage(agent:MiMod, Agrega, Compromisos,comp),
          AppendToList(comp:Result, ProduceM1),         };   },  };} );
```

NOTA: Es importante resaltar el uso del campo CMPS:Result para enviar información extra al buscador de proveedores. En este caso la información extra es la meta que se le exigirá al proveedor.

La construcción del producto compuesto se limita a consumir la materia prima y producir un nuevo recurso, en este caso, el producto compuesto. Las precondiciones por supuesto, son que exista la materia prima suficiente.

```
MakeInstance( ConstruyeP1, ConstruyePr );
MakeMethod( ConstruyePr, Precond, [plan agent ],
   { ClearList(plan listtmp),
     AppendToList(plan:listtmp, M1, M2);
     If (SendMessage(agent:MiMod, TengoRecursos,plan:listtmp))
    Then TRUE
     Else FALSE;} ).
MakeMethod( ConstruyePr, Accion, [plan agent ],
   {/*Consume Recursos y genera producto */
    Monitor("Construyendo P1");
    SendMessage(agent MiMod, Borra, Recursos, M1),
    SendMessage(agent MiMod, Borra, Recursos, M2);
    Let [rec SendMessage(RCS, Crea)]
    { SendMessage(rec, Actualiza, P1, producto, 1);/* Actualiza estado mental del agente */
      SendMessage(agent MiMod,Agrega, Recursos, rec),
      AppendToList(plan:Comp Result, P1, producto, 1), }.
     ClearList(plan Fases);              /* ya no son necesarias las etapas anteriores del plan */ } ).
```

La obtención de un proveedor de materia prima, también requiere de un plan con dos fases, **BuscarQuien** y **HacerComp**.

```
MakeClass( ObtenerProv, PLANES ).
MakeMethod( ObtenerProv, Iniciar, [agent ],
   { /* Instancia Meta a conseguir por el provedor */
    MakeSlot(Self Meta),  MakeSlot(Self Prov),        MakeSlot(Self listtmp),
    SetSlotOption(Self listtmp, MULTIPLE);
    Self:Meta = GetNthElem(Self Comp:Result,1),
    ClearList(Self Comp Result),
    Monitor(" Iniciando busqueda de proveedor para "#Self Meta), } ).
MakeMethod( ObtenerProv, Terminar, [agent ],
   { Monitor("Se ha conseguido provedor de la meta "#Self Meta), } ).
SetValue( ObtenerProv Fases, BuscarQuien, HacerComp ).
```

La primera etapa del plan se enfoca en la búsqueda de algún agente conocido que pueda realizar la tarea para él. Si no conoce a alguien con dichas caracteristicas, deberá enviar mensajes a todos los agentes en el sistema preguntando quien puede hacerle la tarea

```
MakeInstance( BuscarQuien, FASES );
MakeMethod ( BuscarQuien, Precond, [plan, agent])
   { ClearList(plan listtmp);  /* Verifica si ya conoce a alguien */
    AppendToList(plan:listtmp,  SendMessage(agent, Selecciona, Capacidades, plan:Meta));
   If (LengthList(plan listtmp)== 0)
   Then TRUE
   Else FALSE;}.
MakeMethod( BuscarQuien, Accion, [plan agent ],
   /* Busca un nuevo conocido acorde a sus metas */
     { Monitor("Buscando conocido");
      ClearList(plan listtmp),
      AppendToList(plan listtmp, plan Meta);
      SendMessage(agent, Envia, Todos,  PuedeHacer, 0, plan listtmp), } );
```

La segunda parte del plan se activará en cuanto se conozca a alguien con el cual se pueda establecer un compromiso para obtener la materia prima.

```
MakeInstance( HacerComp, FASES );
MakeMethod( HacerComp, Precond, [plan agent ],
  { If (LengthList( SendMessage(agent, Selecciona, Capacidades,plan:Meta) ) != 0 )
    Then TRUE
    Else FALSE;} );
MakeMethod( HacerComp, Accion, [plan agent ],
  { /* Hacer un compromiso con el primer proveedor encontrado*/
    Monitor("Haciendo el compromiso");
    plan:Prov = GetNthElem(SendMessage(agent, Selecciona, Capacidades, plan:Meta), 1);
    ClearList(plan:listtmp);
    AppendToList(plan:listtmp, plan:Meta, EjAg:Reloj +20, -1);
    SendMessage(agent, Envia, plan:Prov, Comp,0, plan:listtmp);
   /* Actualizando el modelo del provedor */
   Let [comp   SendMessage(CMPS, Crea)]
      [modprov  SendMessage(agent, Modelo?, plan:Prov)]
   {   SendMessage(comp, Actualiza, agent, plan:Meta, EjAg:Reloj+20,EjAg:Reloj,-1);
    SendMessage(modprov, Agrega, Compromisos, comp); }; } );
```

5.1.3 Resultados

Despues de ejecutar los ejemplos el estado mental de los agentes ha cambiado

Agentes	Habilidades	Recursos	Conocidos	Meta
Ag1	HacerM1, ConstruyeP1	P1	Ag1, Ag2	--
Ag2	HacerM2, ConstruyeP2	P2	Ag3 Ag5 Ag1	--
Ag3	HacerM4, ConstruyeP3	P3	Ag4	--
Ag4	HacerM5, ConstruyeP4	P4	Ag5, Ag1	--
Ag5	HacerM1, ConstruyeP5	P5	Ag6, Ag7, Ag2	--
Ag6	HacerM3, ConstruyeP2	P6	Ag1, Ag2, Ag3, Ag5	--
Ag7	HacerM3, ConstruyeP1	P7	Ag1, Ag2, Ag5	--

fig. 5.3 Estado mental final de los agentes

Los resultados de las dos corridas del ejemplo son mostrados en las ventanas del sistema y en parte del listado obtenido con la opción de **Imprime** del menú de **Accesorios**.

```
SESION: C:\KAPPA\ALDEAS.SES                          21/03/98
===========================================================
RELOJ: 4      METAS: 35      FRACASOS: 0    TRAFICO: 191
===========================================================
===========================================================
AGENTE  Ag2
HABILIDADES          HacerP2           ProduceM2
RECURSOS             P2        producto          1
COMPROMISOS
( Agente, Meta, TFirma, TMax, TInic, TTerm, Estado )
Ag2   ObtenerProv    0      20     1      2      Terminado
Ag2   ProduceM2      0      20     1      1      Terminado
Ag2   ObtenerProv    0      20     1      2      Terminado
Ag1   ProduceM2      2      22     2      2      Terminado
Ag5   ProduceM2      3      22     3      3      Terminado
Ag6   ProduceM2      3      22     3      3      Terminado
Ag7   ProduceM2      3      22     3      3      Terminado
CONOCIDOS  Ag3   Conocido     Ag5   Conocido     Ag1   Conocido
```

fig. 5.4 Listado parcial de la sesión de trabajo del ejemplo de aldeas

fig. 5.4 Aldeas sin conocimiento previo

fig. 5.5 Estado mental final de un agente

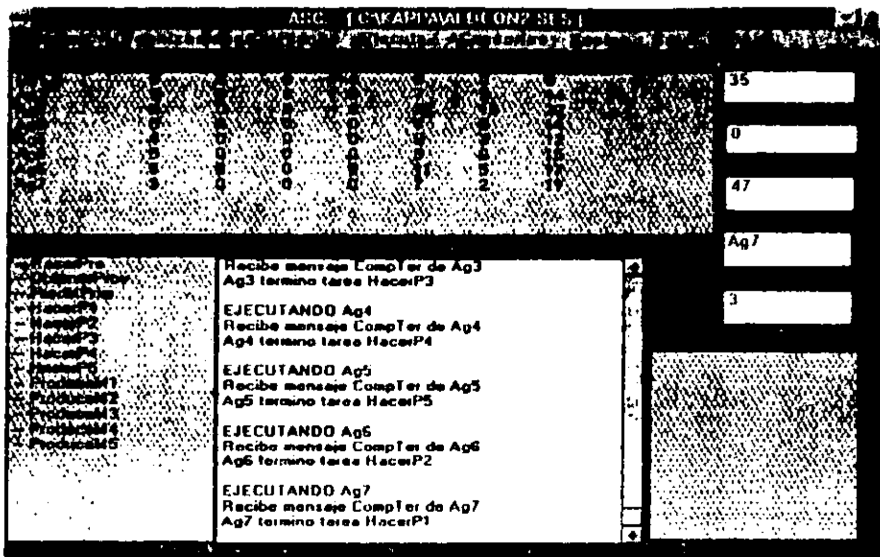

fig. 5.6 Aldeas con conocimiento previo

Con estas dos corridas del ejemplo de aldeas de producción, para el desarrollador de la aplicación sería fácil observar el desempeño de su sistema al dotarlo de características tales como el modelado, la interacción implícita y explícita, la planeación, etc. Podría tener una idea más clara de esto al comparar los resultados (fig. 5.7). Y podría además, analizar el desempeño particular de cada agente en el sistema.

	Sin conocimiento	Con conocimiento
Tráfico	191	47
Reloj	4	3
Metas	35	35

fig. 5.7 Comparación del desempeño de las aldeas en situaciones distintas

Una de las primeras ventajas encontradas en el sistema que usa modelos sería la reducción en los costos de comunicación y, aunque no es muy notorio en este ejemplo, también se reduce el tiempo en alcanzar las metas.

5.2 Redes de Contrato

5.2.1 Descripción

Se ha desarrollado un sistema de agentes que trabajan en el esquema de redes de contrato, donde los agentes toman uno de dos roles: *manejador* o *trabajador*. Se asume una división de tareas dada explícitamente usando planes.

Tareas	PreRequisitos	Recursos	Subtareas
T1	Nada	R1, R2	T2, T3
T2	Nada	R3, R4	T4, T5, T6
T3	T2	R5, R6, R7	T7, T8
T4	Nada	R8	T8, T9
T5	T4	R9	Indivisible
T6	T4	R10	Indivisible
T7	Nada	R11	Indivisible
T8	Nada	R12	Indivisible
T9	T8	R13	Indivisible

fig. 5.8 División de tareas

Se cuenta además con 8 agentes con un estado mental inicial mostrado en fig. 5.8. En donde se puede observar que el agente Ag1 tiene una meta a conseguir: *hacer la tarea T1*, por lo cual será quien inicie la actividad de la red de contratos.

Agente	Habilidades	Recursos	Metas	Conocidos
Ag1	T1	R1, R2	TareaT1	--
Ag2	T2,T7, T8	R11, R12	--	--
Ag3	T2, T5, T9	R3, R4, R9, R13	--	--
Ag4	T3, T8, T9	R5, R6, R7, R12, R13	--	--
Ag5	T4, T6, T7	R8, R10, R11	--	--
Ag6	T2, T6, T7	R2, R4, R1, R9	--	--
Ag7	T3, T4 T9	R5, R6 R7, R8, R13	--	--
Ag8	T2, T5	R4, R9	--	--

fig. 5.9 Estado mental inicial de los agentes

5.2.2 Implementación

La representación de un agente de la red de contratos está dada de la siguiente forma.

```
MakeInstance( Ag1, Agentes ).
MakeInstance( Md1, MDAg ).                 /* Modelo de si mismo */
Md4 Nombre = Ag1.
Md4 Relacion = mio.
Md4 Recursos = (R1, R2 ).
Md4 Capacidades= (TareaT1 );
Md4 Compromisos  = ( Ag1, TareaT1);        /*Meta inicial */
```

Para conseguir una tarea es necesario un plan que permita asignar a otros agentes, o a si mismo si es el caso, las subtareas en las cuales está subdividida, **(AsignTi)**. Una vez que las subtareas estén concluidas se procederá a sintetizar los resultados parciales para concluir la tarea, **(SintTi)**. Veamos el ejemplo del plan para obtener la tarea T1.

```
MakeClass( Tarea, PLANES);
MakeClass( TareaT1, Tarea ).
SetValue( TareaT1 Fases, AsigT2, AsigT3, SintT1 ).
MakeMethod( Tarea, Iniciar, [agent ].
   { Monitor("Iniciando plan TareaT1");} ).
MakeMethod( Tarea, Terminar, [agent ].
   { Monitor("Termina Plan TareaT1");} );
```

Para poder iniciar la asignación de una tarea, es necesario que se cumplan las precondiciones de la misma. Por ejemplo para efectuar la tarea T3 es necesario que la

tarea T2 esté completamente terminada. En la asignación de la tarea, primero el agente verifica si él mismo la puede realizar, en cuyo caso genera un compromiso consigo mismo para realizar la subtarea. Si el agente no está capacitado, genera un compromiso, también consigo mismo, para ofertar la tarea y obtener un proveedor.

```
MakeInstance( AsigT3, AsignarTar );
MakeMethod( AsigT3, Precond, [plan agent ],
   { ClearList(plan:listtmp);
    AppendToList(plan:listtmp, T2);
    If( SendMessage(agent:MiMod, TengoRecursos, plan:listtmp))
    Then TRUE
    Else FALSE;} );
MakeMethod( AsignarTar, Accion, [plan agent ],
   { Monitor(" Asignando T1");
    ClearList(plan:listtmp);
  /* Verifica que la tarea no este hecha */
   AppendToList(plan:listtmp, T1);
   If Not ( SendMessage(agent:MiMod, TengoRecursos, plan listtmp))
   Then    { /* Si puede, hace tarea */
     If ( Member?(agent:MiMod:Capacidades, Tarea1 ))
     Then
      {Monitor("Se compromete consigo mismo");
       Let [ comp   SendMessage(CMPS, Crea)]
       { SendMessage(comp, Actualiza, agent, Tarea1,   EjAg Reloj+20, EjAg Reloj,-1),
        SendMessage(agent:MiMod, Agrega, Compromisos comp);    }   }
    Else /* Se compromete a ofertar tarea, (buscando proveedor) */
     { Monitor("Se compromete a buscar quien "),
      Let [ comp   SendMessage(CMPS, Crea)]
       {  SendMessage(comp, Actualiza, agent, ObtenerProv,
          EjAg.Reloj+20, EjAg:Reloj,-1);
         SendMessage(agent MiMod, Agrega, Compromisos.comp);
         AppendToList(comp Result, Tarea1, R1, R2).
           }.  }.  }.    } ).
```

La síntesis de resultados está condicionada a la terminación de todas las subtareas

```
MakeClass( SintT1, FASES );
MakeMethod( SintT1, Precond, [plan agent ],
   {/* Verifica que todas las subtareas esten completas */
   ClearList(plan:listtmp),
   AppendToList(plan listtmp, T2, T3);
   If (SendMessage(agent:MiMod, TengoRecursos,plan listtmp))
   Then TRUE
   Else FALSE;} );
MakeMethod( SintT1, Accion, [plan agent ],
   {/* Sintetiza resultados y hace la tarea general */
    Monitor("Haciendo T1");
    SendMessage(agent:MiMod, Borra, Recursos, T2);
    SendMessage(agent:MiMod, Borra, Recursos, T3),
    AppendToList(plan:Comp:Result, T1, tarea, 1); } );
```

El plan para obtener un proveedor de una tarea (i.e. ofertar tareas), es similar al del ejemplo con aldeas de producción, sólo se ha aumentado una fase más para monitorear o hacer un seguimiento de los contratos establecidos. En la *primera etapa* del plan, **BuscaQuien**, se busca un conocido y si no existe se manda un mensaje en *broadcast* para conocer a alguien. Esta fase es similar al ejemplo anterior y no se incluye en este.

```
MakeClass( ObtenerProv, PLANES ),
MakeMethod( ObtenerProv, Iniciar, [agent ],
```

```
{/* Instancia Meta y recursos a conseguir por el proveedor */
 MakeSlot(Self:Meta);                    MakeSlot(Self:Recursos),
 SetSlotOption(Self:Recursos, MULTIPLE);        ClearList(Self:Recursos);
 MakeSlot(Self:Prov);                    MakeSlot(Self:CmpProv);
 MakeSlot(Self:listtmp);                        SetSlotOption(Self:listtmp, MULTIPLE);
 ClearList(Self:listtmp);                Self:Meta = GetNthElem(Self:Comp:Result,1);
 AppendToList(Self:Recursos, Self:Comp:Result);         RemoveFromList(Self:Recursos, Self:Meta);
 ClearList(Self.Comp:Result); Monitor(" Iniciando busqueda de proveedor para "#Self:Meta);} );
MakeMethod( ObtenerProv, Terminar, [agent ],
  { Monitor("Se ha conseguido proveedor de la tarea "#Self:Meta),} ),
SetValue( ObtenerProv:Fases, BuscarQuien, HacerComp, MonitResult );
```

La *segunda etapa* del plan, **HacerComp**, se encarga de elegir un proveedor de entre todos los ofertantes y establecer un contrato entre él y el agente. Las precondiciones son, que existan ofertantes para la subtarea. Para simplificar el ejemplo la regla de elección del proveedor es: *"primero que encuentra que tenga los recursos suficientes"*, pero el desarrollador de aplicaciones podría establecer una política más compleja para ello.

```
MakeInstance( HacerComp, FASES );
MakeMethod( HacerComp, Precond, [plan agent ],
  { If (LengthList( SendMessage(agent,
        Selecciona, Capacidades,plan:Meta) ) != 0 )
  Then TRUE
  Else FALSE,} ),
MakeMethod( HacerComp, Accion, [plan agent ],
{ClearList(plan:listtmp),
 AppendToList(plan:listtmp,SendMessage(agent, Selecciona,
          Capacidades, plan:Meta)),
 MakeSlot(plan:lista2),
 SetSlotOption(plan:lista2, MULTIPLE);
 ClearList(plan:lista2),
 plan:Prov = NADA,
/* Hacer compromiso con el primer proveedor encontrado
 que tenga todos los recursos */
 EnumList(plan:listtmp, x,
  { Let [ mod  SendMessage(agent, Modelo?, x) ]
   If (SendMessage(mod, TengoRecursos, plan:Recursos))
   Then plan:Prov = x;  });
 If (plan:Prov #= NADA)
 Then
/* Hacer compromiso con primer proveedor encontrado al  que pueda enviar recursos */
  { plan:Prov = GetNthElem(plan:listtmp, 1);
   ClearList(plan:listtmp);
   Let [mod SendMessage(agent, Modelo?, plan:Prov)]
   AppendToList(plan:listmp, SendMessage(mod, Recursos?)),
   AppendToList(plan:lista2, plan:Recursos);
   EnumList(plan:listtmp, x,
     { If (Member?(plan:lista2, x))
       Then RemoveFromList(plan:lista2,x);      }),  }.
 Monitor("Haciendo el CONTRATO con "#plan:Prov);
 ClearList(plan:listtmp);
 AppendToList(plan:listtmp, plan:Meta, EjAg:Reloj +20, -1, plan:lista2 );
 SendMessage(agent, Envia, plan:Prov, Comp,0, plan:listtmp),
/* Actualizando el modelo del proveedor */
Let [comp     SendMessage(CMPS, Crea)]
  [modprov  SendMessage(agent, Modelo?, plan:Prov)]
{  SendMessage(comp, Actualiza,  agent, plan:Meta, EjAg:Reloj+20,EjAg:Reloj,-1);
 SendMessage(modprov, Agrega, Compromisos, comp),
 plan:CmpProv = comp,  };  } );
```

La *tercera etapa* del plan, **MonitorearResult**, permite al agente manejador hacer un seguimiento continuo del contrato y en caso de que el agente proveedor falle establecer un compromiso consigo mismo para buscar un nuevo proveedor. Esta parte del plan se activará cuando el compromiso (contrato) se termine, o cuando se cancele por el proveedor (cuando le es imposible terminarlo) o bien cuando el tiempo permitido por el contrator para terminar la tarea haya expirado.

```
MakeInstance( MonitResult, FASES ).
MakeMethod( MonitResult, Precond, [plan agent ],
  { If (Null?(plan:CmpProv))
    Then FALSE
    Else
     {If ((plan:CmpProv:Estado #= Cancelado) Or
         (plan:CmpProv:Estado #= Terminado) Or
         (plan CmpProv:TMax <= EjAg.Reloj) )
      Then TRUE
      Else FALSE;  }.} ).
MakeMethod( MonitResult, Accion, [plan agent ],
  {/* Si expiró tiempo o proveedor cancela, solicitar nuevo prov */
   If( Not ((plan CmpProv:Estado) #= Terminado) )
   Then
    { Monitor("Falla proveedor");
      Monitor(" Ofertando de nuevo tarea"#plan:Meta).
      Let [ comp  SendMessage(CMPS, Crea)]
       {SendMessage(comp, Actualiza, agent, ObtenerProv,  EjAg Reloj+20, EjAg:Reloj,-1).
       SendMessage(agent:MiMod, Agrega, Compromisos,comp).
       ClearList(comp Result);
       AppendToList(comp Result, plan:Meta, plan:Recursos);      }. }.} ).
```

5.2.3 Resultados

El estado mental de los agentes después de correr los ejemplos es el siguiente.

Agente	Habilidades	Recursos	Metas	Conocidos
Ag1	T1	R1, R2	--	Ag3, Ag6, Ag8, Ag4, Ag7
Ag2	T2,T7, T8	R11, R12	--	--
Ag3	T2, T5, T9	R3, R4, R9, R13	--	Ag5, Ag7, Ag6
Ag4	T3, T8, T9	R5, R6, R7, R12, R13	--	--
Ag5	T4, T6, T7	R8, R10, R11	--	--
Ag6	T2, T6, T7	R2, R4, R1, R9	--	--
Ag7	T3, T4, T9	R5, R6, R7, R8, R13	--	Ag2, Ag4, Ag5
Ag8	T2, T5	R4, R9	--	--

fig. 5.10 Estado mental final de los agentes

Al igual que en el ejemplo anterior se mostrarán las ventanas con los resultados obtenidos y listados parciales de la sesión de trabajo.

```
□ SESION: C:\KAPPA\REDES.SES                          21/03/98
□==========================================================
□ RELOJ: 21      METAS: 17      FRACASOS: 0    TRAFICO: 110
□==========================================================
  AGENTE  Ag1
  HABILIDADES  Tarea11
  RECURSOS
```

R1	material 10		R2	material 10			
COMPROMISOS:							
(Agente, Meta, TFirma, TMax, TInic, TTerm, Estado)							
Ag1	TareaT1	0	23	0	20	Terminado	
Ag1	ObtenerProv	0	20	1	14	Terminado	
Ag1	ObtenerProv	14	34	15	20	Terminado	
CONOCIDOS:							
Ag2	Conocido		Ag3	Conocido		Ag6	Conocido
Ag4	Conocido		Ag7	Conocido			

fig. 5.11 Listado parcial de la sesión de trabajo del ejemplo de redes de contratos

A partir de los listados de la sesión obtenidos con ayuda del sistema **ASC** o bien revisando las ventanas del modificador de agentes, podría derivarse la red de contratos que se formó en este ejemplo, si se siguen la lista de compromisos de cada uno de los agentes.

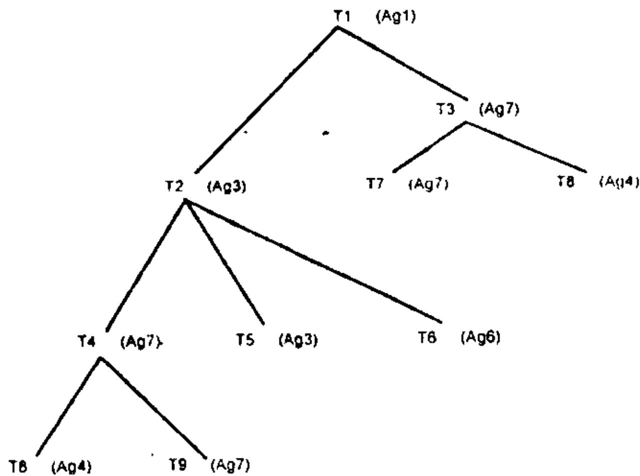

fig. 5.12 Red de contratos formada en el ejemplo

Nuevamente el usuario desarrollador podrá medir el desempeño de los sistemas que diseñó.

	Sin conocimiento	**Con conocimiento**	**Cambiando ambiente**
Tráfico	110	26	29
Reloj	21	15	17
Metas	17	17	18

fig. 5.15 Comparación del desempeño de redes de contrato en ambientes distintos

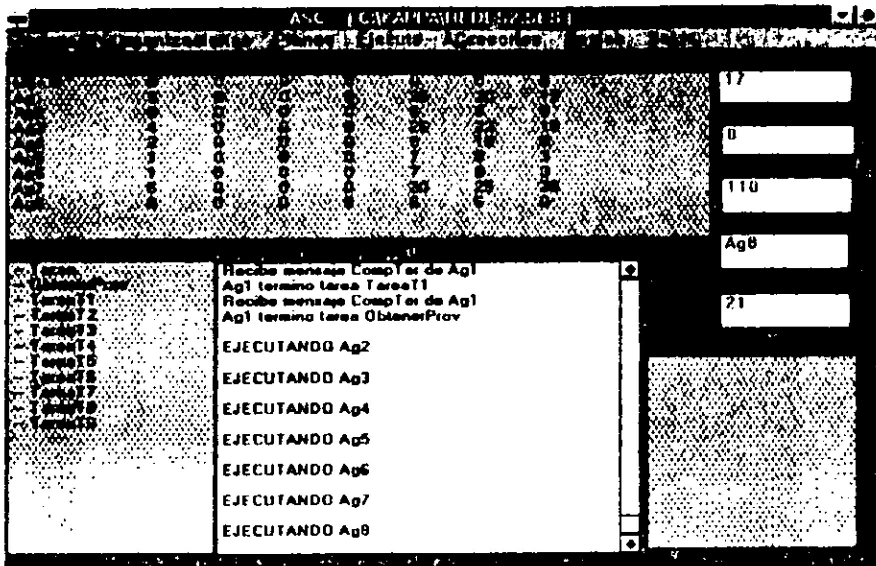

fig. 5.13 Redes de contrato sin conocimiento previo.

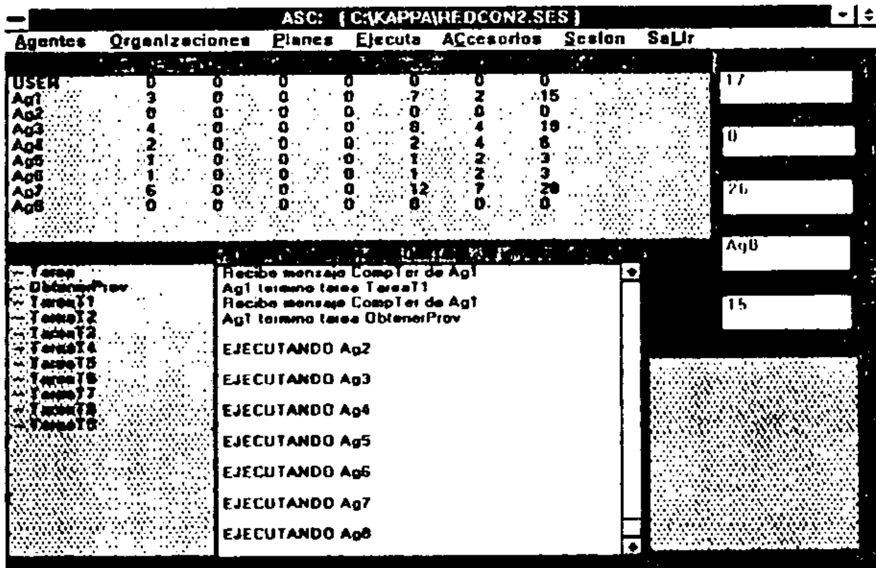

fig. 5.14 Redes de contrato con conocimiento previo

De los resultados mostrados por el sistema podrían sacarse conclusiones inmediatas. Por ejemplo el desempeño del sistema cuando ya se conocen a otros agentes en el medio ambiente es bueno porque reduce costo de comunicación y tiempo en encontrar la solución. Otra conclusión podría ser, que no son necesarios los agentes Ag2 y Ag8 pues dentro de la red no aportaron algo para la solución global del problema.

Una situación interesante que el desarrollador podría modelar en el sistema ASC, sería la *simulación de cambio de condiciones* en el medio ambiente. Por ejemplo de la fig. 5.12 se puede observar que la tarea T7 estuvo asignada al agente Ag5, así que para la segunda corrida del ejemplo Ag7, quien es encargado de asignar la tarea T7, sabe que puede asignársela. Si el desarrollador cambia el estado mental de Ag5 con ayuda del modificador de agentes y le quita la habilidad de realizar la tarea T7, existirá un compromiso fallido por lo cual Ag7 intentará de ofertar nuevamente la tarea. En esta reasignación la red queda como en la figura 5.16

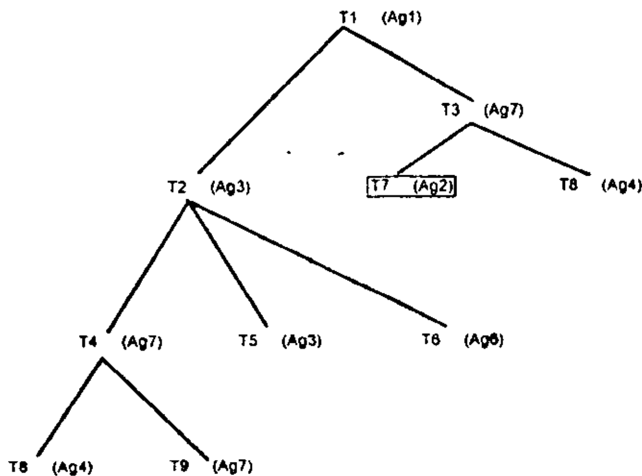

fig. 5.16 Reasignación en redes de contratos

El desempeño del sistema no se vio afectado en gran medida, porque el Ag7 ya conocía otro agente en el sistema (Ag2) que le pudiera hacer la tarea. Así que sólo aumento el número de metas al tener que buscar un nuevo proveedor. Un ciclo de reloj extra y tres mensajes más: el de hacer compromiso y su aceptación correspondiente además del mensaje de compromiso terminado (fig. 5.17).

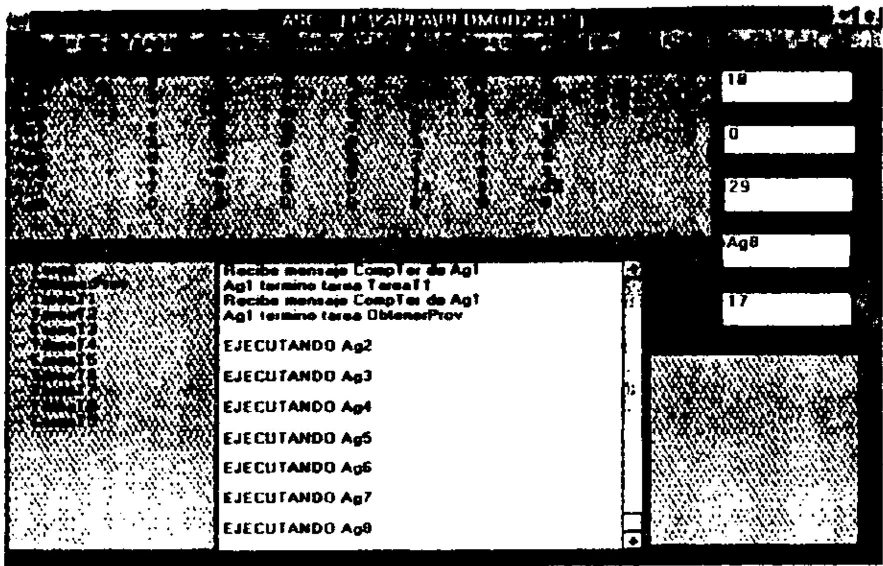

fig. 5.17 Redes de contrato cuando cambia el ambiente

5.3 Organizaciones Jerárquicas

5.3.1 Descripción

Se tiene una organización jerárquica de dos niveles, la cual representa una división encargada de la producción de automóviles. La organización representada tiene como estado inicial al descrito en la figura 5.18

Organización	Habilidades	Recursos	Metas
ProdVW	HacerVW	Pintura, Llantas, Carroceria Tela, Motor, Asientos	HacerVW

Roles	Habilidades	Numero	Miembros
Supervisor	Supervisar, HacerVW	1	--
Tapicero	Tapizar	2	--
Mecanico	PonerMotor	1	--
Ensamblador	Ensamblar	1	--
Pintor	Pintar	1	--

Agente	Habilidades	Recursos	Metas	Conocidos
Ag1	Supervisar, HacerVW	--	--	--
Ag2	Tapizar	--	--	--
Ag3	Tapizar	--	--	--
Ag4	PonerMotor	--	--	
Ag5	Ensamblar	--	--	
Ag6	Pinta	--	--	

fig. 5.18 Estado inicial de la organización

La labor del *agente manejador*, es asignar a cada uno de los *agentes trabajadores*, y de acuerdo a su rol, la tarea adecuada para la consecución de las metas. Después de instanciar la organización con el procedimiento descrito en el capítulo anterior, el estado de la organización cambia al descrito en la figura 5.18.

Organización	Habilidades	Recursos	Metas
ProdVW	HacerVW	--	HacerVW

Roles	Habilidades	Numero	Miembros
Supervisor	Supervisar, HacerVW	1	Ag1
Tapicero	Tapizar	2	Ag2, Ag3
Mecanico	PonerMotor	1	Ag4
Ensamblador	Ensamblar	1	Ag5
Pintor	Pintar	1	Ag6

Agento	Habilidades	Recursos	Metas	Conocidos
Ag1	Supervisar, HacerVW	Pintura, Llantas, Carroceria, Tela, Motor, Asientos	HacerVW	Ag2, Ag3, Ag4 Ag5, Ag6
Ag2	Tapizar	--	--	Ag1
Ag3	Tapizar	--	--	Ag1
Ag4	PonerMotor	--	--	Ag1
Ag5	Ensamblar	--	--	Ag1
Ag6	Pinta	--	--	Ag1

fig. 5.18 Estado depués de instanciar la organización

Nótese que el agente manejador, en este caso **Ag1**, adquirió conocimiento de todos sus trabajadores, de las metas organizacionales y de los recursos de la organización En cambio los agentes trabajadores sólo adquirieron conocimiento de su Jefe.

5.3.2 Implementación

La actividad del manejador está determinada por el plan, dicho plan contempla la búsqueda de entre sus conocidos de aquellos cuyo rol les permita ejecutar determinada tarea.

```
MakeClass( HacerVW, PLANES );
MakeMethod( HacerVW, Iniciar, [agent ],
   { Monitor("Iniciando plan HacerVW"); } );
MakeMethod( HacerVW, Terminar, [agent ],
   {Monitor("Termina Plan HacerVW");} );
HacerVW:Fases, AsignaTapiz, AsignaPonerMotor, AsignaEnsamblado, AsignaPintura, Supervisar ;
```

Las fases correspondientes a la asignación de tareas son similares entre si, asi que sólo se describirá una de ellas

```
MakeClass( AsignarTar, FASES );
MakeMethod( AsignarTar, Precond, [plan agent ],
   { ClearList( plan:listtmp );
     AppendToList( plan:listtmp, SendMessage( agent, Selecciona, Roles, Mecanico ) );
     ClearList( plan:lista2);
```

```
AppendToList(plan:lista2, Motor, Carroceria);
If ( (LengthList(plan:listtmp)>=1) And
    (SendMessage( agent:MiMod, TengoRecursos?, plan:lista2)) });
MakeMethod( AsignarTar, Accion, {plan agent },
 { Monitor( " Asignando mecanica" );
   ClearList( plan:listtmp );
 /* Buscando el trabajador adecuado */
   AppendToList( plan:listtmp, SendMessage( agent, Selecciona, Roles, Mecanico ) );
   Let {mec1 GetNthElem(plan:listtmp,1)}
     {com1 SendMessage( CMPS, Crea )}
   { Let {mod1 SendMessage( agent, Modelo?, mec1 )}
    { ClearList( plan:listtmp );
    AppendToList( plan.listtmp, PonerMotor, EjAg:Reloj + 5, -1, Motor, Carroceria);
 /*Asignando Tarea y Recursos */
    SendMessage( agent, Envia, mec1, Comp, 0, plan:listtmp ),
    SendMessage( com1, Actualiza, agent, PonerMotor, EjAg:Reloj + 5, EjAg:Reloj, -1 );
    SendMessage( mod1, Agrega, Compromisos, com1 );   }}; } );
```

5.3.3 Resultados

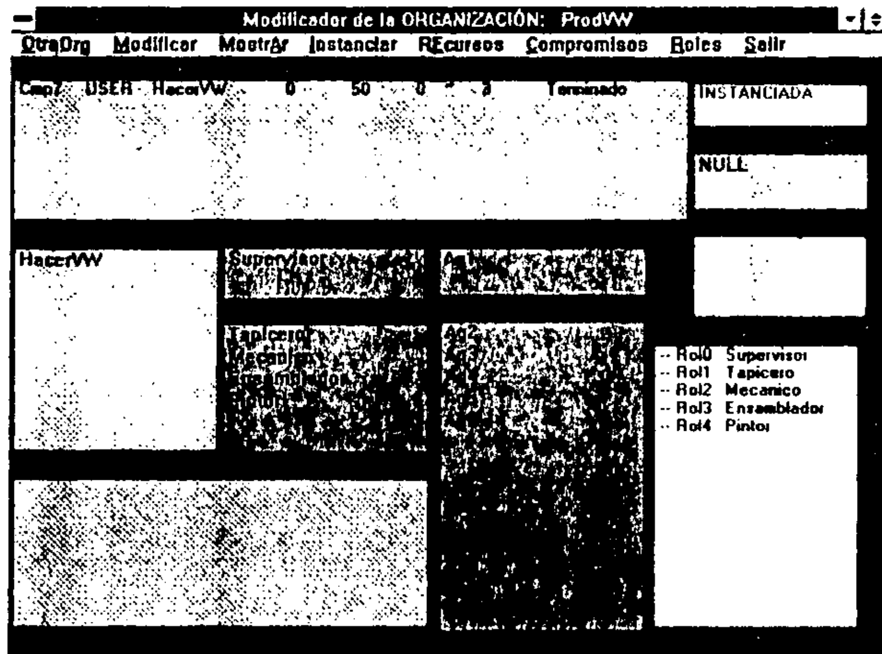

fig. 5.19 Organización jerárquica

Una vez instanciada la organización y ya que se eligieron los miembros adecuados a cada rol, éstos pueden empeza su trabajo. Ésto se logra al insertar metas organizacionales y empezar la ejecución del sistema, en este punto el agente manejador hará suya la meta HacerVW y empezará la ejecución del plan adecuado hasta conseguirla. En la figura 5.19 puede observarse como la meta organizacional fué cumplida por los miembros de la organización

ASC (Ambiente para desarrollo de Sistemas Cooperativos)
SESION: C:\KAPPA\ORG.SES 28/04/98

RELOJ: 5 METAS: 6 FRACASOS: 0 TRAFICO: 15

AGENTE: **Ag1**
HABILIDADES: Supervisar HacerVW
RECURSOS:

Asientos	material	5	Tela	material	50
Carroceria	material	1	Pintura	material	5
Motor	material	4	AsientosTapizados	producto	2
AsientosTapizados	producto	2	AutoConMotor	producto	1
AutoEnsamblado	producto	1	AutoPintado	producto	1

COMPROMISOS:

(Agente,	Meta,	TFirma,	TMax,	TInic,	TTerm,	Estado)
Ag1	HacerVW	0	50	0	3	Terminado

CONOCIDOS:

Ag2	MiTrab	Ag3	MiTrab	Ag4	MiTrab
Ag5	MiTrab	Ag6	MiTrab		

AGENTE: **Ag2**
HABILIDADES: Tapizar
RECURSOS:
COMPROMISOS:
CONOCIDOS: Ag1 Jefe

PLAN: **HacerVW**
FASES: AsignaTapiz AsignaPonerMotor AsignaEnsamblado AsignaPintura Supervisar

PLAN: **Tapizar**
FASES: Tapizando

PLAN: **PonerMotor**
FASES: PoniendoMotor

ORGANIZACION: **ProdVW**
HABILIDADES: HacerVW
RECURSOS:
COMPROMISOS:

(Agente,	Meta,	TFirma,	TMax,	TInic,	TTerm,	Estado)
USER	HacerVW	0	50	0	0	NoIniciado

MANEJADOR: Supervisor Ag1
MIEMBROS:

Tapicero	Ag2	Tapicero	Ag3
Mecanico	Ag4	Ensamblador	Ag5
Pintor	Ag6		

fig. 5.20 Listado parcial de la sesión con organizaciones

CONCLUSIONES

El desarrollo de este trabajo ha permitido la construcción de un ambiente de experimentación mediante la aplicación de diferentes teorías acerca de los sistemas cooperativos.

En particular, los agentes **ASC** incorporan en su arquitectura mecanismos que permiten la cooperación , tales como el modelado, la comunicación, el intercambio de conocimiento entre agentes, la planeación, la representación del estado mental y del conocimiento organizacional.

Estos mecanismos permitieron al sistema **ASC** dotar a sus agentes con los siguientes atributos: ***deliberativos, cooperativos, interactivos, reactivos y sociales.*** Y por lo tanto lo hacen útil para el desarrollo de varias arquitecturas de sistemas cooperativos y con diferentes niveles de complejidad.

Con respecto a las organizaciones, se ha trabajado con el enfoque en donde la actividad de las organizaciones la realizan los propios agentes que la forman y aunque se restringió a la representación de organizaciones jerárquicas, se dejan las bases para representación futura de otro tipo de organizaciones. En particular esta es un área con un gran potencial de investigación

Como herramienta de experimentación, en el ambiente **ASC**, los usuarios desarrolladores de sistemas cooperativos pueden auxiliarse para la obtención de prototipos de sistemas cooperativos y a tener una idea general de su desempeño antes de construirlo, debido a que en **ASC** pueden encontrar las facilidades necesarias para:

- Modelar sistemas ya desarrollados (redes de contrato, pizarrones, organizaciones, etc.)
- Modelar nuevos sistemas de agentes cooperantes.
- Modelación de sistemas tanto MultiAgente como DPS (Distributed Problem Solving).
- Simular situaciones de falla y experimentar con resultados.
- Medir el desempeño del sistema ante condiciones ideales o de fallas.
- Medir desempeño de un agente en particular.
- Usar las funciones de ASC para implementación de planes.
- Definición de organizaciones jerárquicas.
- Ejecución paso a paso de los agentes en el sistema.
- Disponer de herramientas de depuración que le faciliten el desarrollo de sus modelos.

Es importante mencionar que en el desarrollo de nuevos sistemas cooperativos, el usuario sólo tiene que programar los planes adecuados al comportamiento de sus agentes.

El diseño orientado a objetos empleado en el desarrollo del sistema, lo hacen flexible al mantenimiento evolutivo. Por ejemplo el usuario puede añadir funciones interpretadoras de mensajes fácilmente; o bien definir nuevos atributos a los modelos de los agentes; o bien agregar nuevas características a los agentes, las organizaciones y los planes.

Debido a que KAPPA usa manejo dinámico de memoria, la capacidad de los sistemas cooperativos representados depende únicamente del tamaño de la memoria disponible en la computadora. ASC fue probado con 1000 agentes; usando una computadora personal con procesador 486, 8 Mb de memoria, trabajando a 100 Mhz y no presentó más problema que el tiempo requerido para crearlos (cerca de un minuto y medio) y una baja en el desempeño general del sistema, debido a los intercambios de memoria (*swap)* realizados por Windows.

Lamentablemente, KAPPA almacena todos sus programas en forma de instrucciones del lenguaje KAL que puedan después ser interpretadas para crear las clases, objetos y métodos utilizados en el sistema. Por lo anterior, no pudo obtenerse una medida del tamaño aproximado del código binario para la representación de agentes u organizaciones. El tamaño del código fuente del programa es cerca de 220 Kb y el tamaño de las sesiones usadas en los ejemplos descritos en el capítulo 5, oscila entre los 40 y 70 Kb. Además, mientras que la representación de un agente sin conocimiento ocupa 1.5K, la de un agente con conocimiento ocupa alrededor de 7Kb. Sin embargo, como se mencionó anteriormente, estas cifras no revelan mucho acerca del código binario

RESTRICCIONES

La restricción más fuerte para el sistema es que está desarrollado totalmente en KAPPA, por lo cual los planes y funciones interpretadoras de mensajes deben programarse en el lenguaje KAL de KAPPA. Aunque el lenguaje no es difícil de aprender por su sintaxis semejante a C, se calcula una semana para aquellos usuarios con las nociones básicas de programación orientada a objetos y en programación basada en reglas.

Otra desventaja es que la concurrencia está simulada y puede ser un problema cuando se modelen sistemas distribuidos prácticos, porque los resultados pueden no coincidir con los del sistema.

Comparando **ASC** con otros ambientes de desarrollo disponibles en el mercado, tal como **JATLine** (Java Agent Template, Lite) de la Universidad de Stanford, **ABS** (Agent Building Shell) del Enterprise Integration Laboratory de la Universidad de Toronto, **ABE** (Agent Building Environment) de la IBM, es fácil notar que la gran ventaja de estos sistemas es que funcionan de facto sobre una red y al estar realizados en JAVA, permiten que los agentes definidos puedan ejecutarse en diferentes plataformas de hardware. Sin embargo las facilidades para definir agentes y construir y depurar sistemas cooperativos pueden encontrarse tanto en esos grandes sistemas como en **ASC**. Más aún, **ASC** dota a sus agentes de la capacidad de modelar a otros agentes, cosa que en otros ambientes debe ser programada por el usuario. Otra característica adicional la proporcionan los parámetros para medir el desempeño de los agentes en particular y del sistema en general.

PERSPECTIVAS

En base a los atributos de los agentes podría diseñarse e implementarse un *Lenguaje de definición de agentes*, para usuarios con más experiencia en programación que requieran una implementación rápida de los agentes, las organizaciones y los planes. Este lenguaje sería un traductor para el lenguaje de KAPPA.

Puede ampliarse el conjunto de *funciones interpretadoras de mensaje* a fin de contar con más elementos que permitan la comunicación entre agentes.

Existe mucho más trabajo en cuanto a organizaciones se refiere. Las últimas investigaciones tratan a las organizaciones como adaptivas a los cambios en el medio ambiente, organizaciones que aprenden de sus experiencias propias y de la experiencia de otras, es decir necesitan del mecanismo de la **reorganización**. Sin embargo este es un buen avance para la representación y el estudio de las mismas.

El sistema **ASC**, que funciona adecuadamente tal como se ha mostrado en los ejemplos, puede tomarse como un prototipo para el desarrollo de una herramienta más potente que no dependa del lenguaje de programación empleado. Lo ideal sería la representación de planes en lenguajes ampliamente conocidos, un buen candidato sería el lenguaje C en cualquiera de sus versiones disponibles en el mercado, o cualquier otro lenguaje que permita la creación dinámica de objetos.

Apéndice A

Interfaz de usuario

El Sistema **ASC** cuenta con una interfaz de usuario basada en ventanas, en imágenes, en menús pop-up y de cascada, en botones y en accesos a los "browsers" de KAPPA. Se tienen 4 ventanas principales con sus respectivos menús: *la interfaz principal, la interfaz del modificador de agentes, la interfaz del modificador de planes y la interfaz del modificador de organizaciones.* Asociadas a cada una de ellas, se disponen de *imágenes activas*, visualizadas siempre que se tenga acceso a la interfaz, y de *imágenes ocultas* las cuales sólo se visualizan a petición expresa en algún menú de las interfaces principales, y que pueden volver a ocultarse presionando la tecla de escape (**ESC**).

A.1 Interfaz principal del sistema

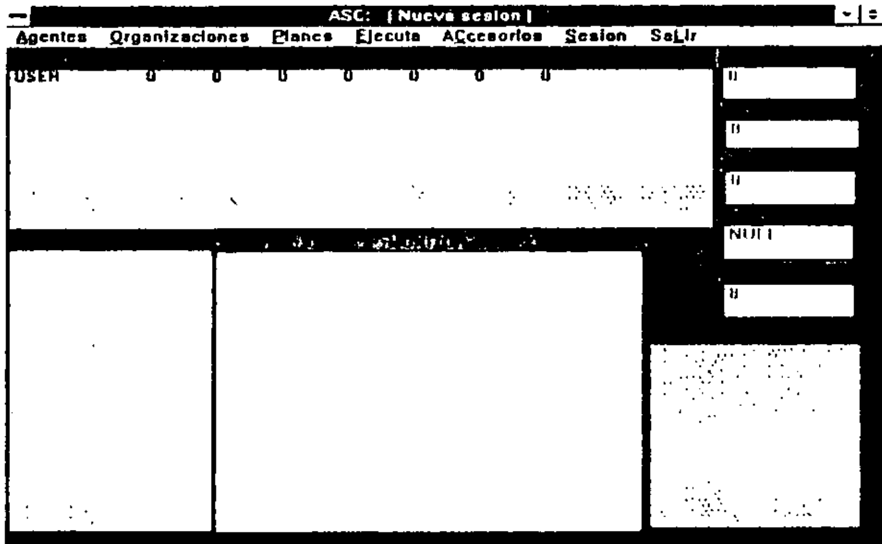

fig. A.1 Ventana principal del depurador

La ventana principal del sistema (fig. A.1) permite accesar directamente los depuradores, el ejecutor de agentes y el manejador de archivos del sistema. Cuenta con nueve imágenes activas, cuatro de ellas despliegan los elementos disponibles en el sistema (agentes, planes, y organizaciones) y cinco para visualizar el desempeño del sistema. Se tiene además dos imágenes ocultas y un menú principal.

IMÁGENES ACTIVAS

Directorio. Visualiza los agentes en el sistema, así como también el desempeño individual de cada uno. El formato de la información desplegada en esta ventana es: nombre del agente, número de metas alcanzadas (éxitos), número de metas fracasadas (canceladas o suspendidas), número de planes en activo (metas en proceso), número de metas aún no iniciadas (pendientes), número de mensajes recibidos, número de mensajes enviados, y finalmente número de acciones atómicas ejecutadas por el agente (fases de planes). Esta ventana es actualizada automáticamente cada vez que alguna de la información desplegada cambia Por ejemplo cuando se crean o destruyen agentes y cuando se ejecuta el ciclo de control de alguno de ellos

Librería de Planes. Muestra los nombres de los planes disponibles en el sistema. Si antes del nombre aparece -- significa que el plan está sin usar por algún agente. Si lo que aparece antes del nombre es ** significa que se han generado algunas instancias del plan y se han asignado a algún agente Si aparece ii, entonces significa que el plan es una instancia de algún plan y está actualmente ocupada por un agente. Al igual que la ventana anterior, esta ventana es actualizada en cuanto existen más planes o instancias de planes en el sistema.

MONITOR. Esta ventana despliega cualquier información enviada desde los agentes al usuario mediante el uso de la función ***Monitor(texto);***

ORGANIZACIONES. Muestra los nombres de las estructuras organizacionales actualmente disponibles en el ambiente.

METAS. Despliega el número total de metas conseguidas en el sistema

FRACASOS. Despliega el número total de metas fracasadas en el sistema

TRAFICO. Visualiza el número total de mensajes hasta el momento enviados en el sistema

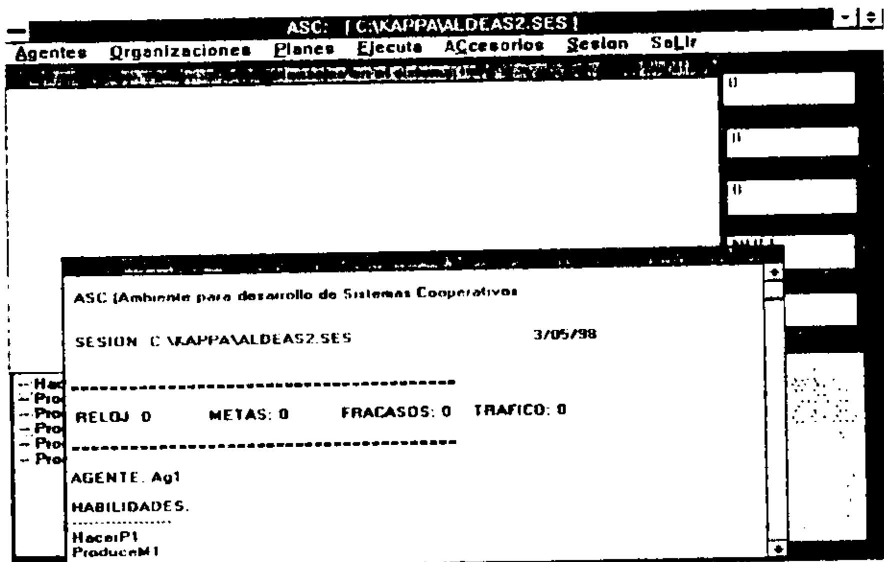

fig. A.2 Imágenes ocultas de la ventana principal.

AGENTE. Nombre del agente actualmente ejecutándose
RELOJ. Visualiza el reloj del sistema.

IMÁGENES OCULTAS

Estas ventanas son visibles sólo a través de la opción ***Mostrar*** del menú de ***Accesorios***
Mensajes en el sistema. Visualiza todos los mensajes circulando actualmente. (fig A 2)
Impresiones. Visualiza la impresión de una sesión de trabajo.

MENÚ PRINCIPAL

Agentes, **Organizaciones**, **Planes** y **Fases**, dan acceso al menú del depurador correspondiente (**Modificar**, **Renombrar**, **Leer** de disco, **Escribir** a disco, **Crear** o **Destruir** un elemento. (fig A 3)
Ejecuta. Acceso a las opciones para ejecución de agentes . (**Agente**) Ejecuta un ciclo de control para un agente específico. (**Todos**) ejecuta una vez el ciclo de control para todos los agentes en el sistema. (**Sistema**) ejecuta el ciclo de control de todos los agentes en el sistema hasta que no existan compromisos pendientes, ni mensajes en espera de ser procesados por algún agente Además en este menú se encuentran disponibles dos opciones extras (**ResetAG**) Reset a los agentes del sistema borra cualquier mensaje y compromiso pendiente de los agentes en el sistema (**KAPPA**) Acceso a las ventanas de KAPPA, esto es con el fin de aprovechar las facilidades de depuración y selección de puntos de ruptura que el sistema ofrece mediante su *KALDebugger* (fig A 4)

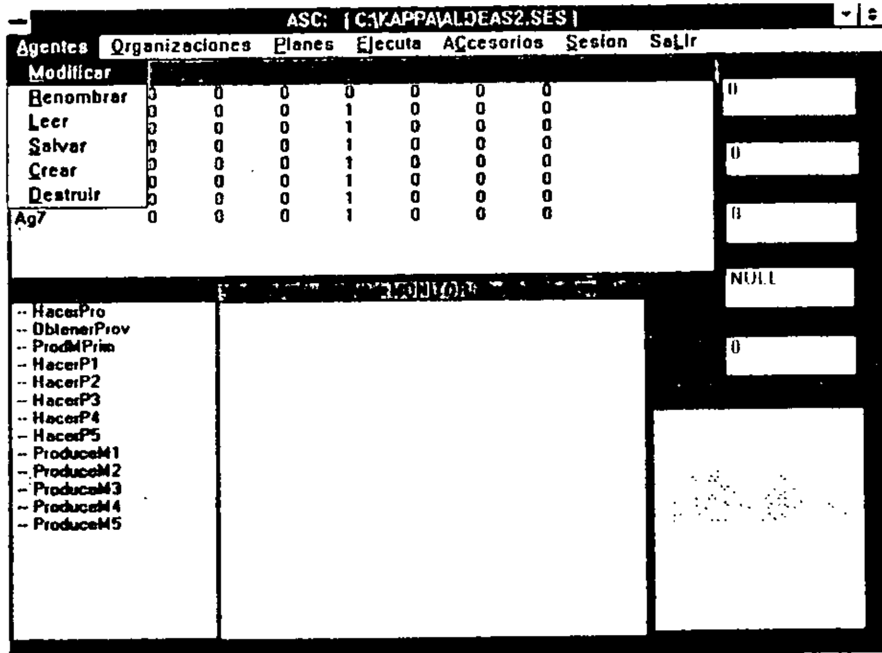

fig. A.3 Acceso a los depuradores de objetos

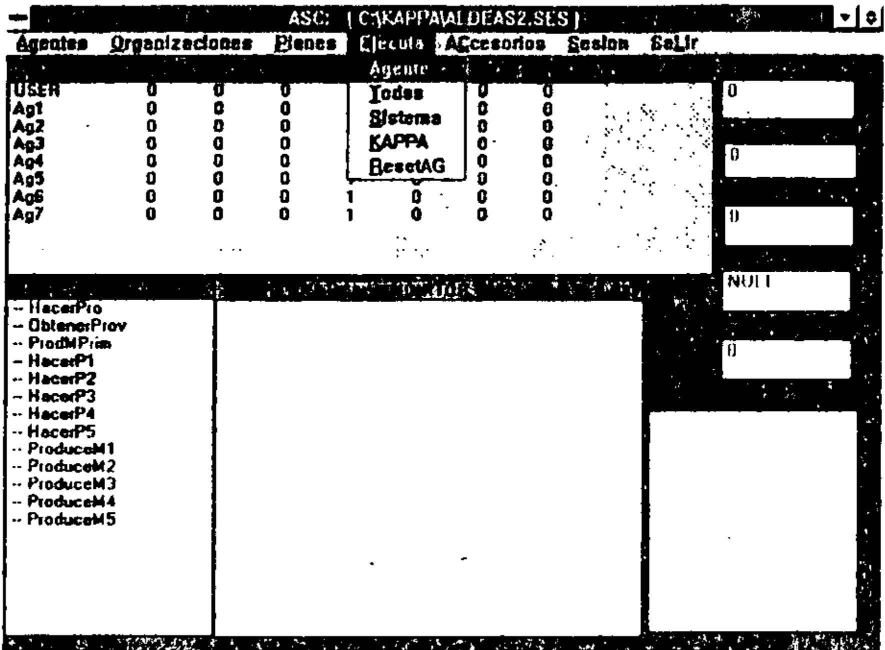

fig. A.4 Acceso al ejecutor de agentes

ACcesorios. Opciones para crear y borrar accesorios en el sistema Permite **Crear** y **Borrar** mensajes desde la interfaz de usuario Además cuenta con una opción para desplegar en una ventanas (**Mostrar**) los mensajes circulando en el sistema También se tiene una opción para **Imprimir** (a un archivo) la descripción de los agentes, los planes y las organizaciones ocupadas en la sesión actual. (fig A 5)
Sesión. Acceso al manejador de archivos para. Iniciar (**Nueva**), abrir (**Abre**) y salvar (**Salva**) una sesión de trabajo Además se tiene la opción de cargar nuevamente en memoria desde el disco una sesión de trabajo (**Reset**).. (fig A 6)
SAlir. Termina el sistema

Aunados a esta ventana principal se tienen varios menús pop-up que emergen cuando el sistema solicita alguna información al usuario (fig A 7) Por ejemplo cuando es necesario elegir un agente, plan u organización; o bien cuando el sistema necesita que el usuario desarrollador elija una opción.

En la parte superior de la ventana principal aparecerá además, el nombre del archivo de donde se ha tomado la sesión de trabajo actual

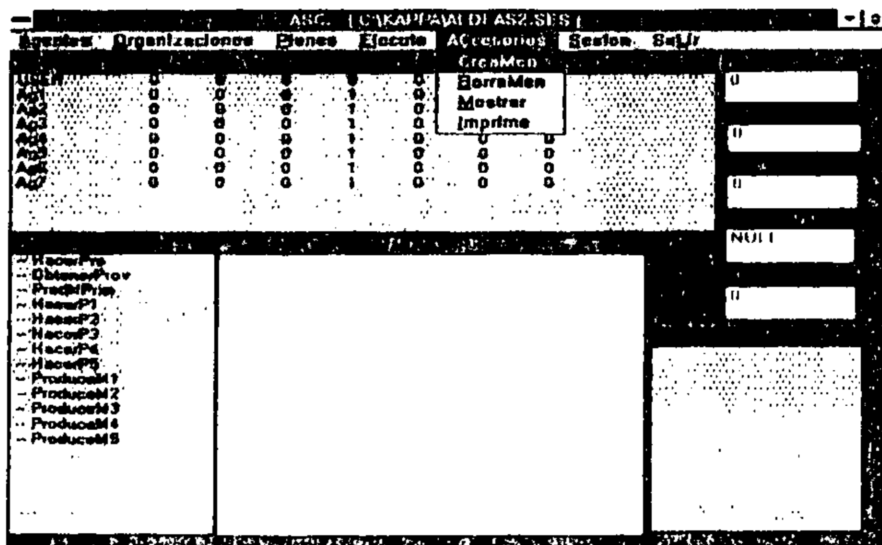

fig. A.5 Menú de accesorios del sistema

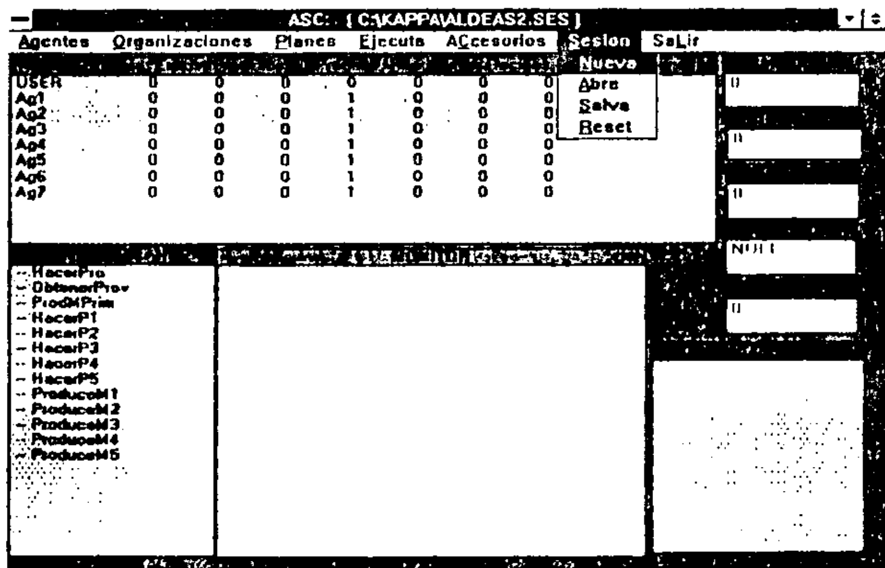

fig. A.6 Acceso al manejador de archivos

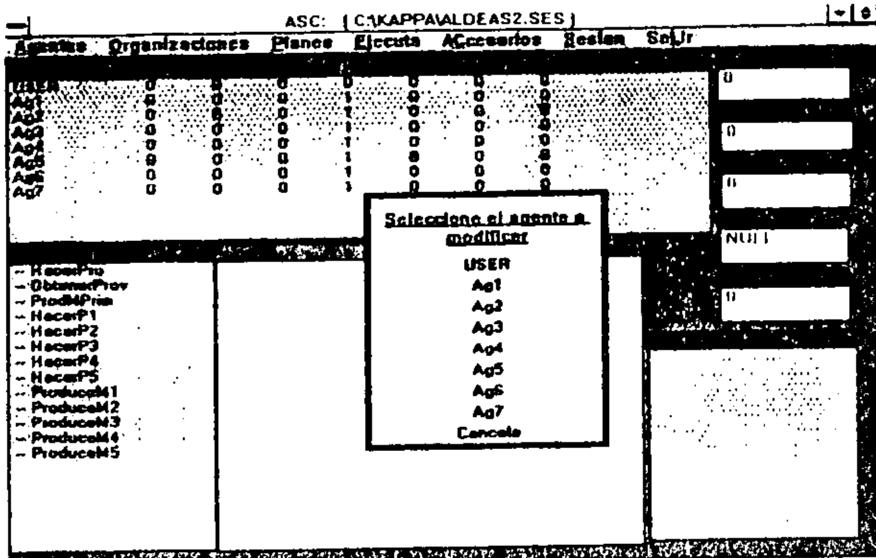

fig. A.7 Menús pop-up de elección múltiple

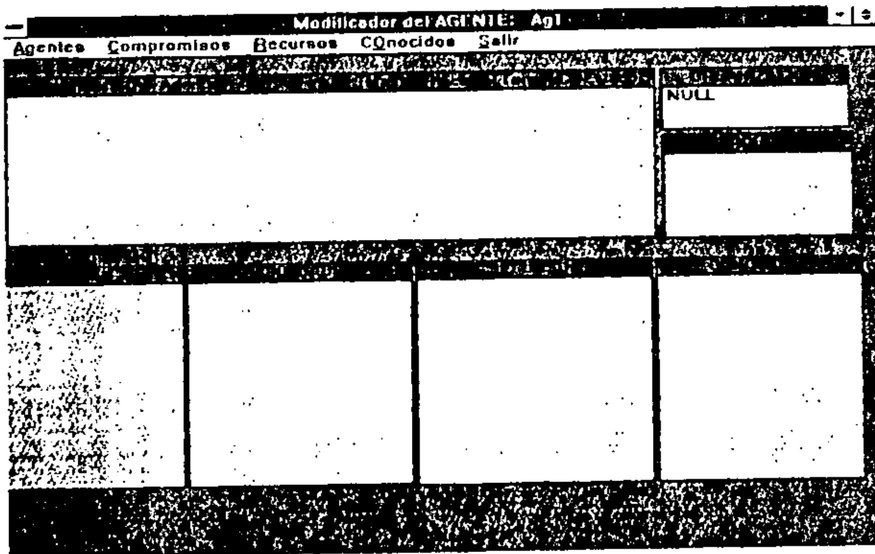

fig. A.8 Ventana principal del modificador de agentes

A.2 Interfaz del modificador de agentes

Esta interfaz da acceso a los menús que permiten modificar los atributos de un agente. Cuenta con un menú principal, seis imágenes activas para desplegar la información de los atributos del agente y dos imágenes ocultas para desplegar todos los compromisos y recursos generados en el sistema (fig A 8)

IMAGENES ACTIVAS

Organización. Despliega el nombre de la organización a la cual pertenece el agente
Roles. Visualiza los roles asignados al agente dentro de su organización
Habilidades. Despliega las habilidades del agente
Recursos. Visualiza los recursos asignados al agente
Pendientes. Visualiza las acciones o fases de un plan pendientes para ejecutarse por el agente
Conocidos. Despliega los nombres de los agentes conocidos y las relaciones con ellos
Compromisos. Despliega los compromisos que el agente actual ha firmado El formato para la información desplegada en esta ventana es: nombre del agente con el cual se esta comprometido, meta a conseguir con el compromiso, hora en que se firmó el compromiso, tiempo máximo para alcanzar la meta, tiempo en que se inició el plan para conseguir la meta, tiempo en que se terminó y estado actual del compromiso

IMÁGENES OCULTAS

RecSistema Muestra todos los recursos (fig A 9)
CmpSistema. Despliega todos los compromisos.

Modificador del AGENTE: Ag1

Agentes Compromisos Recursos COnocidos Salir

0 0 NoIniciado NULL

** Cmp0	Ag1	HacerP1	0	50	0	0	NoIniciado
** Cmp1	Ag2	HacerP2	0	50	0	0	NoIniciado
** Cmp2	Ag3	HacerP3	0	50	0	0	NoIniciado
** Cmp3	Ag4	HacerP4	0	50	0	0	NoIniciado
** Cmp4	Ag5	HacerP5	0	50	0	0	NoIniciado
** Cmp5	Ag6	HacerP2	0	50	0	0	NoIniciado
** Cmp6	Ag7	HacerP1	0	50	0	0	NoIniciado

fig. A.9 Imágenes ocultas del modificador de agentes

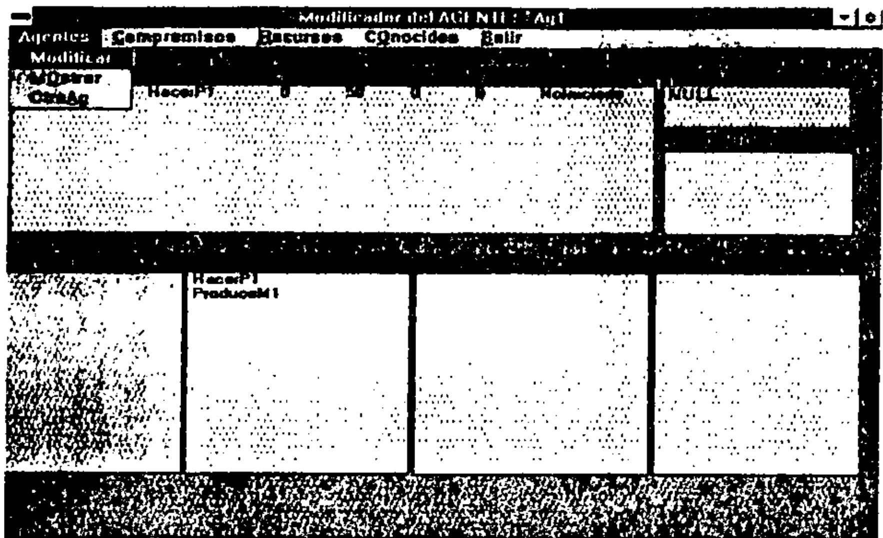

fig. A.10 Menú para modificar agentes

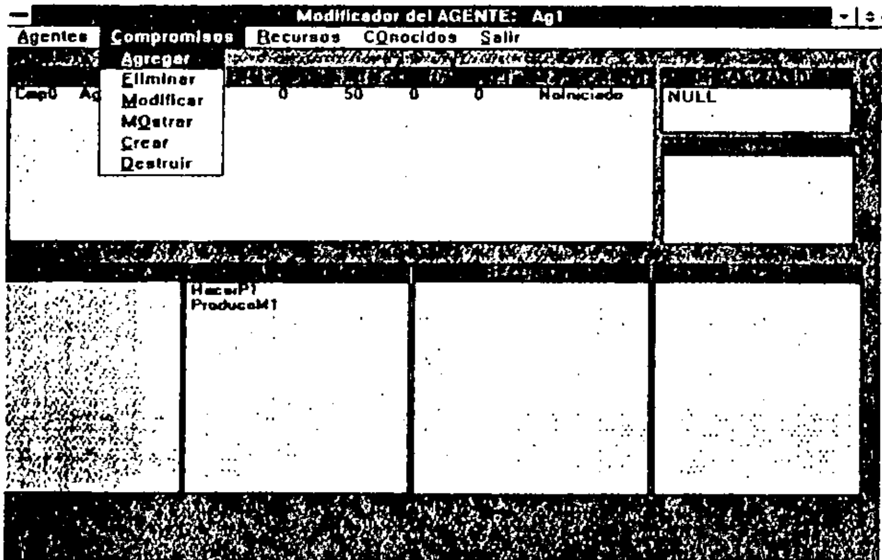

fig. A.11 Menú de modificación de compromisos y recursos

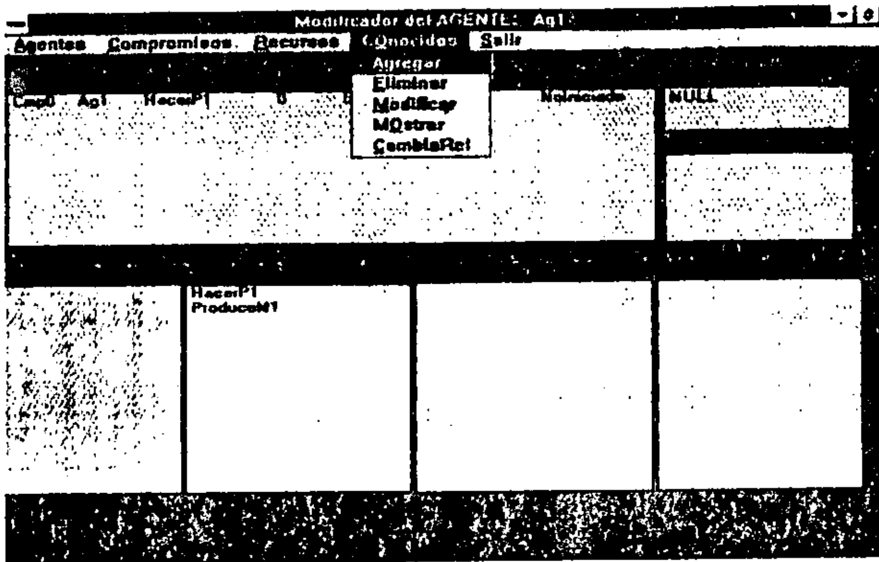

fig. A.12 Menús de modificación de conocidos

MENÚ PRINCIPAL

Agente. Permite **Modificar** interactivamente los atributos de una agente, **MOstrar** el estado mental del agente y seleccionar otro agente a modificar (**OtroAg**). (Ver fig A 10)

Compromisos, Recursos y COnocidos. Permiten **Agregar**, **Eliminar**, **Modificar** o **MOstrar** un compromiso, un recurso o un conocido del agente (Ver fig A 11) Las menús de compromisos y recursos cuentan además con opciones para **Crear** o **Destruir** un elemento En el menú de **COnocidos**, además se puede cambiar el tipo de relación mantenida con otro agente (**CambiaRel**) (fig A 12)

Salir. Regreso a la ventana principal

A.3 Interfaz del modificador de planes

Esta interfaz, si bien es la más sencilla en cuanto al número de imágenes y menús contenidos, es una de las más potentes, pues nos permite acceso al "browser" del sistema de KAPPA, a fin de que se puedan crear y modificar nuevos métodos que podrán asignarse a las funciones *Iniciar* y *Terminar*, o bien como *Fases* de un plan. Consta de un menú principal, dos imágenes activas y la ventana de acceso al browser.

IMÁGENES ACTIVAS

Fases. Despliega la lista de fases asignadas al plan

Librería de Fases. Muestra todas las fases disponibles en el sistema (fig A 13)

Browser. Ventana del sistema KAPPA que permite modificar los métodos de los objetos planes o fases (fig A14)

ESTA TESIS NO DEBE
SALIR DE LA BIBLIOTECA

MENÚ PRINCIPAL.

Plan. Permite el acceso al **Browser** de KAPPA para modificar las funciones relativas a un plan; **AgregaFase** o **EliminaFase** al plan, definir una jerarquía de planes usando herencia para heredar propiedades del plan padre (**CambiaPadre**), y permite además seleccionar **OtroPlan** a modificar (fig A 15)

Fases. Permite el acceso al **Browser** de KAPPA para modificar las funciones relativas a las *precondiciones y acciones* de una fase: Da acceso al *depurador de fases* para **Renombrar**, **Leer**, **Salvar**, **Crear** y **Destruir** una fase, además permite definir una jerarquía de fases y aprovechar la herencia (**CambiaPadre**) (fig A 16)

Salir. Regreso a la ventana principal

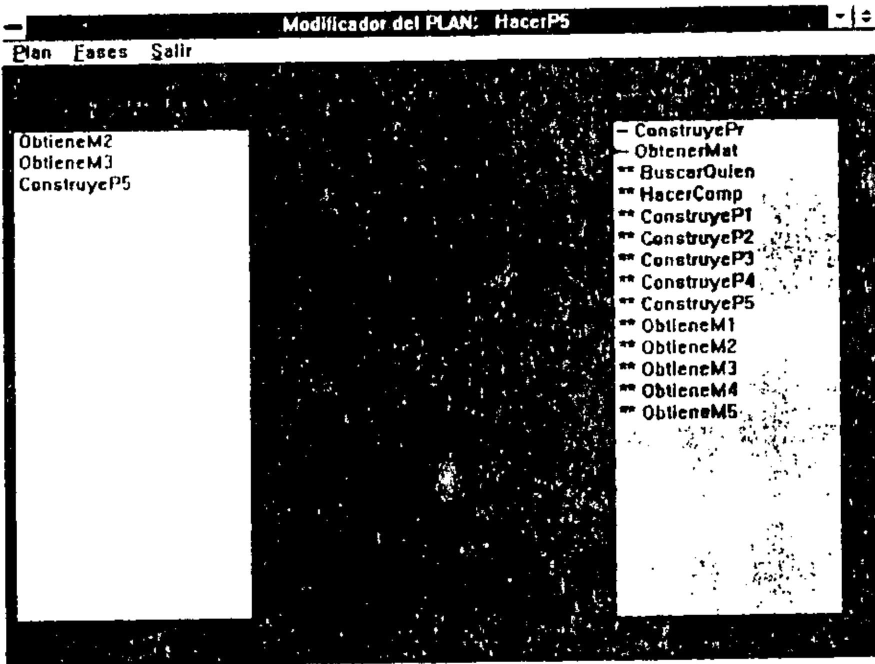

fig. A.13 Ventana principal del modificador de planes

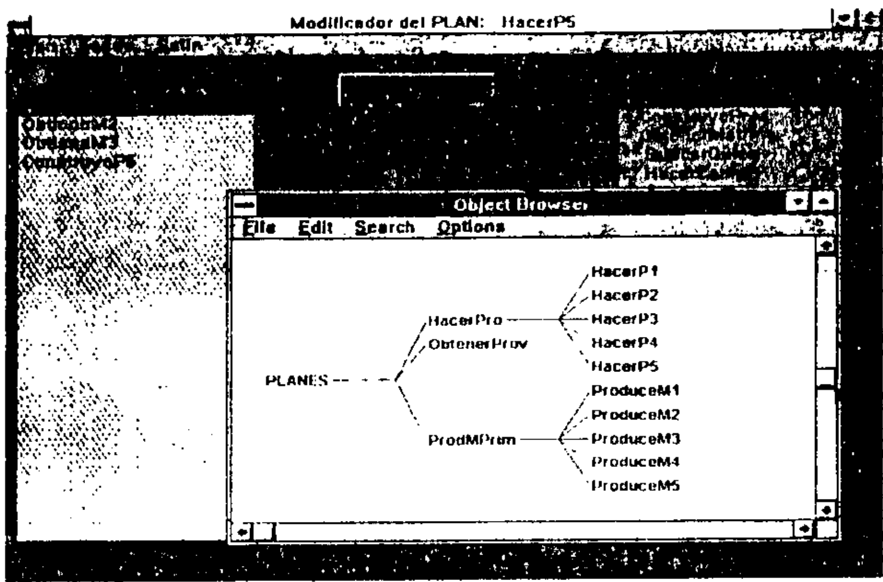

fig. A.14 Ventana de acceso al "Browser" de KAPPA

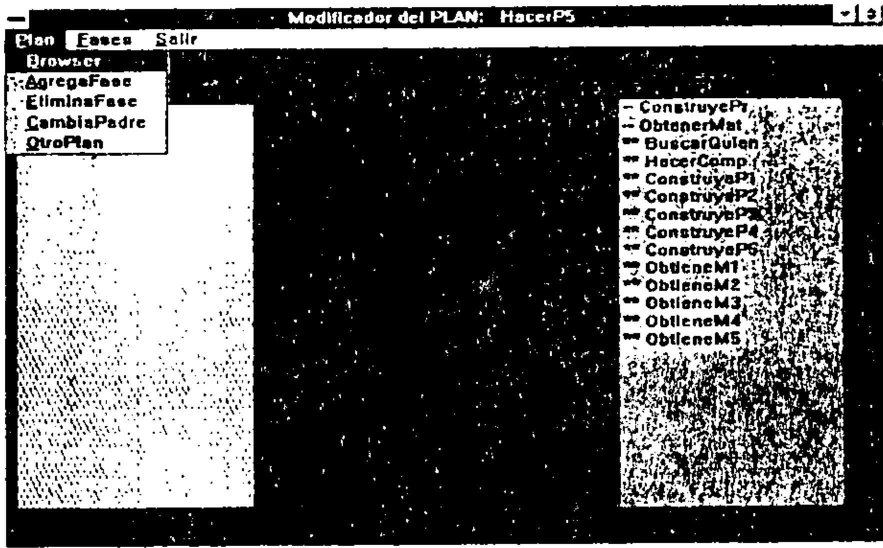

fig. A.15 Menú de modificación de planes

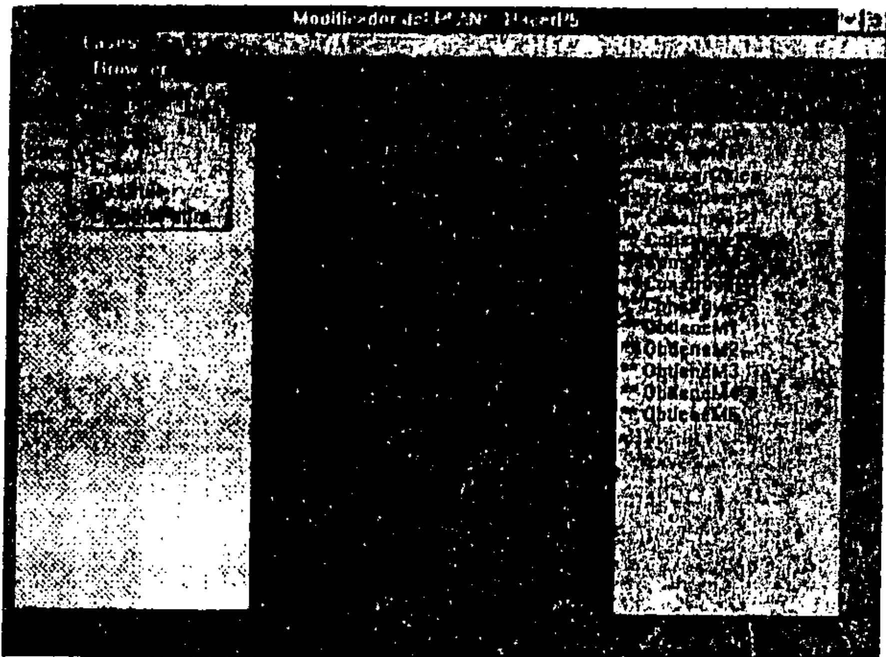

fig. A.16 Menú de modificación de fases

A.4 Interfaz del modificador de organizaciones

Esta interfaz permite definir estructuras organizacionales pertenecientes a las estructuras jerárquicas. Consta de un menú principal, once imágenes activas (fig A.17) y 4 imágenes ocultas.

IMÁGENES ACTIVAS

Compromisos. Visualiza los compromisos organizacionales (metas de la organización).
Habilidades. Muestra las habilidades de la organización.
Recursos. Despliega los recursos disponibles en la organización.
Organización. Organización de nivel superior a la cual pertenece.
RolesOrg. Roles asignados a la organización actual por su organización de nivel superior.
RolManejador. Rol desempeñado por el agente manejador de la organización
Manejador. Miembro designado como manejador de la organización.
RolTrabajadores. Roles desempeñados por los agentes trabajadores de la organización.
Trabajadores. Miembros designados como trabajadores de la organización.
ROLES. Despliega los roles definidos actualmente en el sistema.
ESTADO. Muestra el estado actual de la organización **INSTANCIADA** o **NoINSTANCIADA**.

IMÁGENES OCULTAS

ROL. Despliega el nombre del rol descrito (fig. A18).
Cantidad. Muestra el número de miembros que deberán desempeñar el rol.
Habilidades. Despliega las habilidades requeridas para los miembros que ocuparán el rol.
Miembros. Lista de agentes que han sido elegidos para desempeñar el rol.

MENÚ PRINCIPAL.

OtraOrg. Permite seleccionar una organización distinta para modificarla.
Modifica. Acceso al modificador de atributos de la organización.
MostrAr. Visualiza los atributos de la organización.
Instanciar. Permite encontrar los miembros adecuados a cada rol en la organización.
REcursos y **Compromisos.** Acceso a los modificadores respectivos (igual en agentes).
Roles. Permite el acceso al modificador para **Modificar**, **MOstrar**, **Crear**, o **Destruir** un rol. (fig. A.19)
Salir. Salida a la ventana principal.

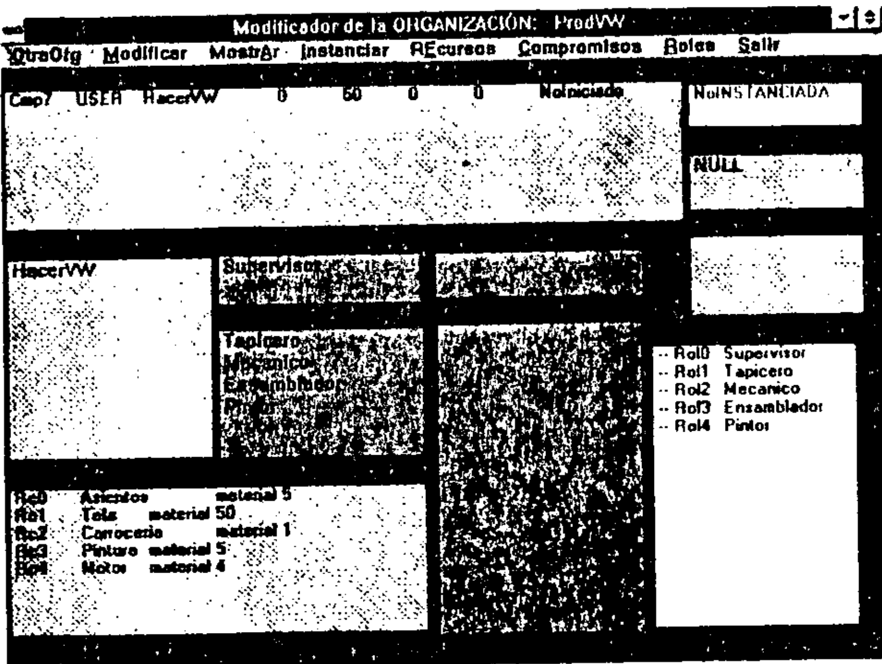

fig. A.17 Ventana principal del modificador de organizaciones

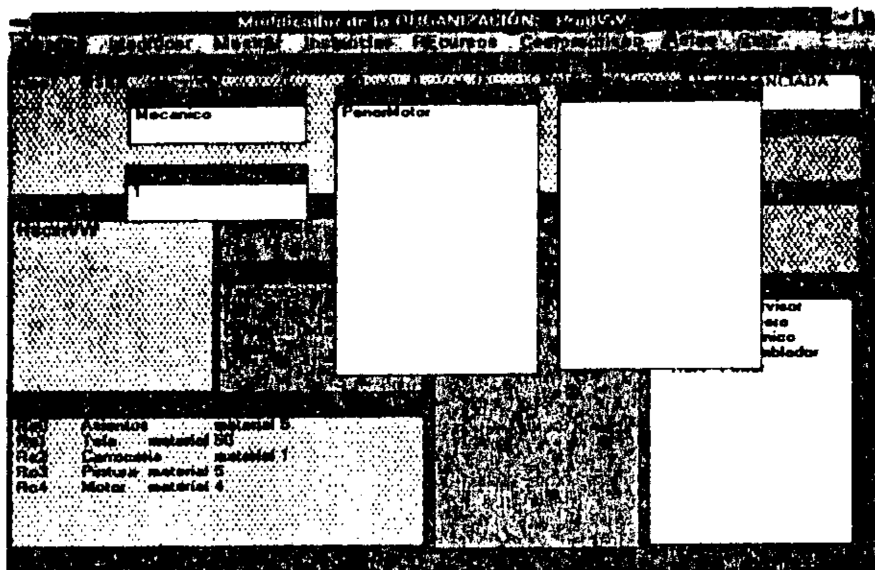

fig. A.18 Imágenes ocultas del modificador de organizaciones

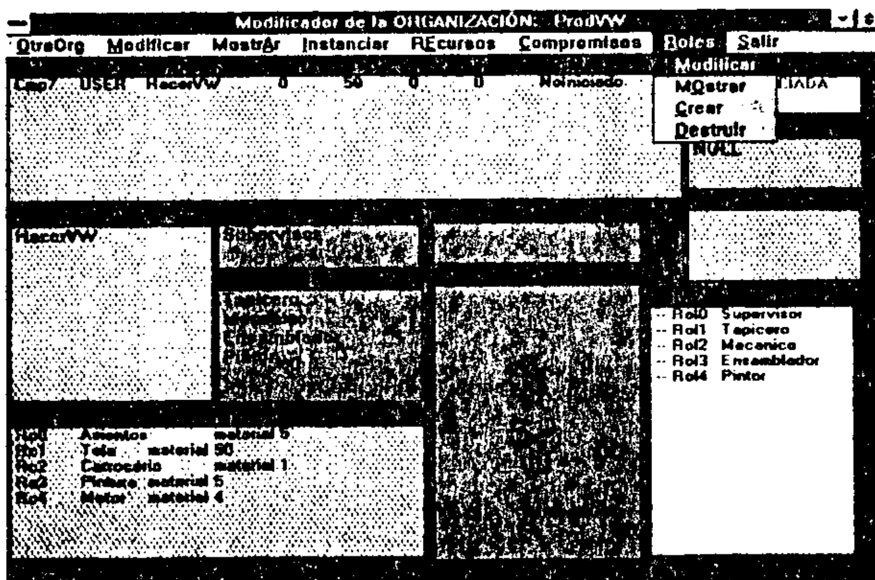

fig. A.19 Menú del modificador de roles

Apéndice B.

Código de las clases más importantes en el sistema

B.1 Clase: Modelos de agentes

```
/************************************
    **** CLASS: MDAg
    **** Representación del conocimiento de agentes
conocidos
    **** por el propietario de este modelo.
    ************************************/
MakeClass( MDAg, Root );
/************** METHOD: Actualiza
**************/
/* "Permite actualizar los valores de los atributos.
valor es una lista. Borrando los valores anteriores." */
MakeMethod( MDAg, Actualiza, [atributo valor ],
{If GetSlotOption(Self:atributo,SINGLE)
 Then SetValue(Self:atributo, valor)
 Else { ClearList(Self:atributo);
    AppendToList(Self:atributo,valor);} } );

/************** METHOD: Agrega **************/
/* "Permite agregar un nuevo valor a un atributo del
modelo" */
MakeMethod( MDAg, Agrega, [atributo valor ],
{If GetSlotOption(Self:atributo,SINGLE)
 Then SetValue(Self:atributo,valor)
 Else If Not (Member?(Self:atributo,valor)) Then
      AppendToList(Self:atributo,valor);} );

/************** METHOD: Informa
**************/
/* "Regresa el valor de un atributo o FALSE cuando el
atributo no existe " */
MakeMethod( MDAg, Informa, [atributo ],
{If Slot?(Self, atributo) Then Self:atributo;} );

/************** METHOD: Destruye
**************/
/* "Destruye la información del modelo. Invoca
destruccion
recursiva para los objetos compuestos. Sólo llamar
desde la clase MdAg o subclases." */
MakeMethod( MDAg, Destruye, [nomb ],
{ If ( Instance?(nomb) And (IsAKindOf?(nomb, Self)))
 Then {
   ClearList(Self:xlisttmp);
   GetSlotList(Self, Self:xlisttmp);
   EnumList(Self:xlisttmp, x,
   {If GetSlotOption(nomb,x,VALUE_TYPE) #=
OBJECT
   Then Let [ClasAtrib
GetNthElem(GetSlotOption(nomb, x,
ALLOWABLE_CLASSES),1)]
      EnumList( nomb:x, y,  SendMessage(
ClasAtrib, Destruye, y) )}};
   DeleteInstance(nomb);
},} );

/************** METHOD: Crea **************/
/* "Crea un modelo del agente "nomag" cuya relación
es "vinculo"
Regresa el nombre del modelo. Sólo invocar desde
MdAg." */
MakeMethod( MDAg, Crea, [nomag vinculo ],
{If Class?(Self) Then {
  Let [ modelo  Md#MDAg:nmod]
   { MDAg:nmod +=1;
     MakeInstance( modelo, Self);
     modelo:Nombre = nomag;
     modelo:Relacion = vinculo;
     modelo; } };} );

/************** METHOD: Modifica
**************/
/* "Modifica interactivamente los atributos de un
agente." */
MakeMethod( MDAg, Modifica, [],
{ PostInputForm("Estado mental de "#Self:Nombre,
     Self:Capacidades, "Habilidades:",
     Self:Recursos, "Recursos",
     Self:Compromisos,"Compromisos:");
  PostInputForm("Organizacion de "#Self:Nombre,
     Self:Org, "Nombre:",
     Self:Roles, "Roles"); } );

/************** METHOD: Borra **************/
/* "Elimina el valor de un atributo." */
MakeMethod( MDAg, Borra, [atributo valor ],
{ If (atributo #= Recursos)
 Then
 {  EnumList(Self:Recursos, x,
    If (x:Nombre #= valor)
    Then
      {RemoveFromList(Self:Recursos, x);
       DeleteInstance(x);} ),  }
Else
 { If GetSlotOption(Self:atributo,SINGLE)
  Then ResetValue(Self:atributo)
  Else If (Member?(Self:atributo,valor)) Then
       RemoveFromList(Self:atributo,valor); },} );
```

```
/************** METHOD: Es? **************/
/* "Pregunta si el atributo tiene un valor determinado.
Si el atributo es lista, pregunta si es miembro de ella.
Para recursos y metas toma solo nombres." */
MakeMethod( MDAg, Es?, [atributo valor ],
{ Self:xtest = TRUE;
 If (atributo #= Recursos)
 Then
 {  If Not(Member?(SendMessage(Self,Recursos?),
valor))
    Then Self:xtest = FALSE;  }
 Else
 { If (atributo #= Metas)
   Then
   {  If Not(Member?(SendMessage(Self,Metas?),
valor))
      Then Self:xtest = FALSE;   }
  Else
  {If GetSlotOption(Self, atributo, SINGLE)
    Then
     { If Not(Self:atributo #= valor)
       Then Self:xtest =FALSE; }
     Else
      {If Not(Member? ( Self:atributo, valor))
       Then Self:xtest = FALSE;}   }; };
Self:xtest;     } );

/************** METHOD: Recursos?
**************
/* "Regresa una lista con los nombres de los recursos
disponibles del agente." */
MakeMethod( MDAg, Recursos?, [],
{ClearList(Self:xlisttmp);
 EnumList(Self:Recursos,
          x,AppendToList(Self:xlisttmp,x:Nombre));
 Self:xlisttmp;} );

/************** METHOD: Metas?
**************/
/* "Regresa una lista con las metas de los agentes." */
MakeMethod( MDAg, Metas?, [],
{ ClearList(Self:xlisttmp);
 EnumList(Self:Compromisos, x,
   If ((x:Estado #= NoIniciado) Or
      (x:Estado #= EnProceso))
   Then AppendToList(Self:xlisttmp,x:Meta));
 Self:xlisttmp;} );

/************** METHOD: Mostrar
**************
/* "Despliega en ventanas la informacion del modelo."
*/
MakeMethod( MDAg, Mostrar, [],
{  ClearTranscriptImage(OrgAg);
   DisplayText(OrgAg,
FormatValue("%s\r",Self:Org));
   DespList( Self:Roles, RolAg);
   DespList( Self:Capacidades, HabAg);
   DespObj ( Self:Recursos, RecAg);
   DespObj ( Self:Compromisos, CompAg);
   ClearList(Self:xlisttmp);
   EnumList(Self:Compromisos, x,
    {If (x:Estado #=EnProceso)
     Then
     {If (Method?(Self:Nombre, x:Plan))
      Then AppendToList(Self:xlisttmp, x:Plan)
      Else AppendToList(Self:xlisttmp, x:Plan:Fases)
}     });
   DespList( Self:xlisttmp, AcPe);  } );

/************** METHOD: TengoRecursos
**************/
/* "Revisa si cuenta con los recursos requeridos." */
MakeMethod( MDAg, TengoRecursos, [recursos ],
{ClearList(Self:lista1);
 AppendToList(Self:lista1,  SendMessage(Self,
Recursos?));
 If SubConj?(recursos, Self:lista1)
 Then TRUE
 Else FALSE;} );

/************** METHOD: Salva  **************/
/* "Salva el modelo en un archivo previamente abierto "
*/
MakeMethod( MDAg, Salva, [],
{ WriteInstance(Self);
 EnumList( Self:Compromisos, x, WriteInstance(x));
 EnumList( Self:Recursos, x, WriteInstance(x));  } );

/************** METHOD: Copia  **************
/* "Copia este modelo en un modelo destino " *
MakeMethod( MDAg, Copia, [dest ],
{ If( IsAKindOf?(dest, MDAg))
 Then
 { dest:Org = Self:Org;
   AppendToList(dest:Capacidades, Self:Capacidades);
   AppendToList(dest:Roles, Self:Roles);
   EnumList(Self:Compromisos, x,
      Let [comp  SendMessage(CMPS, Crea)]
        { SendMessage( comp, Actualiza, x:Agent,
x:Meta,
                x:TMax, x:TFirm, x:TInic);
         SendMessage(dest, Agrega, Compromisos,
comp);       } );
   EnumList(Self:Recursos, x,
      Let [rec  SendMessage(RCS, Crea)]
       { SendMessage( rec, Actualiza, x:Nombre,
          x:Tipo, x:Cantidad);
         SendMessage(dest, Agrega, Recursos, rec);
} );  }
 Else
  Advertencia("No se puede copiar el modelo ");} );

MakeSlot( MDAg:Relacion );
SetSlotComment( MDAg:Relacion, "Relación de este
agente con el agente modelado." );
SetSlotOption( MDAg:Relacion,
ALLOWABLE_VALUES, Mio, MiTrab, JefeOrg,
Colega, Conocido );
SetSlotOption( MDAg:Relacion, PROMPT, "Relacion
con el agente:" );
MakeSlot( MDAg:Roles );
```

```
SetSlotComment( MDAg:Roles, "Papel desempeñado
por el agente dentro de su organización." );
SetSlotOption( MDAg:Roles, MULTIPLE );
ClearList( MDAg:Roles );
SetSlotOption( MDAg:Roles, PROMPT, "Roles: " );
MakeSlot( MDAg:Org );
SetSlotComment( MDAg:Org, "Mantiene el nombre de
la organización de la cual
es parte el agente." ),
SetSlotOption( MDAg:Org, PROMPT, "Organización "
);
MakeSlot( MDAg:Capacidades );
SetSlotComment( MDAg:Capacidades, "Lista de
acciones directamente realizables por el agente.
" ),
SetSlotOption( MDAg:Capacidades, MULTIPLE );
ClearList( MDAg:Capacidades );
SetSlotOption( MDAg:Capacidades, PROMPT,
"Habilidades " ),
MakeSlot( MDAg:Recursos ),
SetSlotComment( MDAg:Recursos, "Recursos del
agente " ),
SetSlotOption( MDAg:Recursos, MULTIPLE ),
SetSlotOption( MDAg:Recursos, VALUE_TYPE,
OBJECT ),
SetSlotOption( MDAg:Recursos,
ALLOWABLE_CLASSES, RCS ),
ClearList( MDAg:Recursos ),
SetSlotOption( MDAg:Recursos, PROMPT,
"Recursos " ),
MakeSlot( MDAg:Compromisos );
SetSlotComment( MDAg:Compromisos, "Agenda de
compromisos" ),
SetSlotOption( MDAg:Compromisos, MULTIPLE ),
SetSlotOption( MDAg:Compromisos, VALUE_TYPE,
OBJECT ),
SetSlotOption( MDAg:Compromisos,
ALLOWABLE_CLASSES, CMPS );
ClearList( MDAg:Compromisos );
SetSlotOption( MDAg:Compromisos, PROMPT, "Lista
de compromisos." );
MakeSlot( MDAg:nmod );
SetSlotComment( MDAg:nmod, "Numero de modelo
actual." ),
SetSlotOption( MDAg:nmod, VALUE_TYPE,
NUMBER );
SetSlotOption( MDAg:nmod, MINIMUM_VALUE, 0
).
MDAg:nmod = 1;
MakeSlot( MDAg:Nombre ),
SetSlotComment( MDAg:Nombre, "Nombre del agente
modelado" );
SetSlotOption( MDAg:Nombre, PROMPT, "Modelo
del agente: " ),
```

B.2 Clase: Agentes

```
/*************************************
**** CLASS: Agentes
**** Plantilla para definición de agentes
*************************************/
MakeClass( Agentes, Comunicador );

/************** METHOD: Planificador
**************/
/* "Planificador del agente." */
MakeMethod( Agentes, Planificador, {},
{  /* Revisando por nuevos compromisos */
EnumList(Self:MiMod:Compromisos, x,
 If (( x:Estado #= NoIniciado) And
     ((x:TInic == -1) Or
      (x:TInic == EjAg:Reloj)))
 Then
  {/* La meta requiere un plan */
    Self:xmod = NULL;
    EnumSubClasses(PLANES, y,
     If ( y#=x:Meta )
     Then   Self:xmod = y);
    If (Not ( Null?(Self:xmod)))
    Then
      { Let [ instplan SendMessage(Self:xmod, Crea,
INST)]
        { instplan:Comp = x,
          x:Plan = instplan;
          SendMessage(instplan, Iniciar, Self),
          Self:Acciones +=1,
          x:Estado = EnProceso;
          x:TInic = EjAg:Reloj;  } }
/* No existe un plan para alcanzarla */
    Else
      { x:Estado = Cancelado,
        x:TInic  = EjAg:Reloj;
        x:TConc  = EjAg:Reloj;
        ClearList(Self:xlisttmp);
        AppendToList(Self:xlisttmp, x:Meta),
        AppendToList (Self:xlisttmp, x:TConc),
        SendMessage(Self, Envia, x:Agente, LoSiento,
0,Self:xlisttmp).
      };  });
/* Generando la agenda actual*/
EnumList(Self:MiMod:Compromisos, x,
  If (x:Estado #= EnProceso)
  Then
   {    EnumList(x:Plan:Fases, y,
        If (SendMessage(y, Precond, x:Plan, Self))
        Then
        {AppendToList(Self:Agenda,y);
          AppendToList(Self:AgendList, x:Plan),
          RemoveFromList(x:Plan:Fases,y);
        });   });      } );

/************** METHOD: Ejecutor
**************/
MakeMethod( Agentes, Ejecutor, {},
{/* Cumpliendo agenda */
Let [ nacc  LengthList(Self:Agenda)]
 For n From 1 To nacc  Do
  { Let [ fase GetNthElem(Self:Agenda,n)]
      [ inst GetNthElem(Self:AgendList,n)]
    { SendMessage( fase, Accion, inst, Self),
      Self:Acciones +=1;    }, };
```

```
ClearList(Self:Agenda);
ClearList(Self:AgendList);
/*Checando por metas alcanzadas */
EnumList(Self:MiMod:Compromisos, y,
   If ((y:Estado #= EnProceso) And
      ((LengthList(y:Plan:Fases) == 0)))
   Then
   { y:Estado = Terminado;
     y:TConc = EjAg.Reloj;
     SendMessage(y:Plan, Terminar, Self);
     Self:Acciones +=1;
     DeleteInstance(y:Plan);
     MakeSlot(Self:lista2),
     SetSlotOption(Self:lista2, MULTIPLE);
     ClearList(Self:lista2),
     AppendToList(Self:lista2, y:Meta, y:TConc,
y:Result);
     SendMessage(Self, Envia, y:Agente, CompTer,
0,Self:lista2),
     DeleteSlot(Self:lista2),    });    } );

/************** METHOD: Crea **************/
MakeMethod( Agentes, Crea, [nomb ],
{If Not (Instance?(nomb))
Then {
   MakeInstance(nomb,Self),
   ClearList(nomb:Agenda),
   ClearList(nomb:AgendList);
   nomb:nmen =0,
   nomb:recibos = 0,
   nomb:envios = 0,
   Let [ClasMod GetNthElem(GetSlotOption(Agentes,
Modelos, ALLOWABLE_CLASSES),1)]
   AppendToList(nomb:Modelos, nomb:MiMod =
            SendMessage(ClasMod, Crea, nomb, mio));
   nomb:xmod = nomb:MiMod;      },     } ).

/************** METHOD: Destruye
**************
MakeMethod( Agentes, Destruye, [nomb ],
{ Let [ClasMod GetNthElem(GetSlotOption(Agentes,
Modelos, ALLOWABLE_CLASSES),1)]
 EnumList(nomb:Modelos, x, SendMessage(ClasMod,
Destruye,x));
 DeleteInstance(nomb),   } ).

/************** METHOD: ElimConoc
**************,
/* "Elimina el modelo asociados a un conocido del
agente." */
MakeMethod( Agentes, ElimConoc, [conocido ],
{Let [ y  SendMessage(Self, Modelo?, conocido) ]
 {RemoveFromList(Self:Modelos, y );
 Let [ClasMod GetNthElem(GetSlotOption(Agentes,
Modelos, ALLOWABLE_CLASSES),1)]
 SendMessage(ClasMod, Destruye,y)     };      } );

/************** METHOD: AgregaConoc
**************
/* "Agrega un conocido al agente " */
MakeMethod( Agentes, AgregaConoc, [conocido rel ],
{ Let [ClasMod GetNthElem(GetSlotOption(Agentes,
         Modelos, ALLOWABLE_CLASSES),1)]
   AppendToList(Self:Modelos,
      SendMessage(ClasMod, Crea,conocido,rel))  :  }
);

/************** METHOD: Mostrar
**************/
/* "Visualiza en pantalla el modelo del agente actual."
*/
MakeMethod( Agentes, Mostrar, [],
{ SetWindowTitle(Session1, "Modificador del
AGENTE:  "#Self);
 ShowImage(AcPe);
 SendMessage(Self:MiMod, Mostrar);
 ShowImage(Conocidos);
 SendMessage(Self, MostrarConoc),     } ).

/************** METHOD: Informa
**************/
/* "Informa valor del atributo de un conocido " */
MakeMethod( Agentes, Informa, [conocido atributo ],
{ SendMessage(
     SendMessage(Self,Modelo?,conocido),
          Informa, atributo) ;     } );

/************** METHOD: Actualiza
**************/
/* "Actualiza atributo de un conocido " */
MakeMethod( Agentes, Actualiza, [conocido atributo
valor ],
{  SendMessage(
   SendMessage(Self,Modelo?,conocido),
      Actualiza, atributo, valor);} );

/************** METHOD: Agrega **************
/* "Agrega un valor al atributo de un conocido " *
MakeMethod( Agentes, Agrega, [conocido atributo
valor ],
{  SendMessage(
   SendMessage(Self,Modelo?,conocido),
      Agrega, atributo, valor),  } );

/************** METHOD: Modifica
**************/
/* "Modifica un conocido." */
MakeMethod( Agentes, Modifica, [conocido ],
{SendMessage(SendMessage(Self,Modelo?,conocido),
              Modifica);    } );

/************** METHOD: Modelo?
**************/
/* "Regresa modelo asociado a un conocido " */
MakeMethod( Agentes, Modelo?, [conocido ],
{ Self:xmod = NULL;
  EnumList(Self:Modelos, x,
    If ((x:Nombre) #= conocido ) Then Self:xmod = x),
  Self:xmod;    } );

/************** METHOD: Borra **************
/* "Borra el valor del atributo de un conocido " *.
```

```
MakeMethod( Agentes, Borra, [conocido atributo valor
],
{ SendMessage ( SendMessage( Self, Modelo?,
conocido),
     Borra, atributo, valor);    } );

/************** METHOD: Conocidos?
**************/
/* "Regresa nombres de los conocidos." */
MakeMethod( Agentes, Conocidos?, [],
 { ClearList(Self:xlisttmp);
 EnumList(Self:Modelos, x,
 AppendToList(Self:xlisttmp,x:Nombre));
 RemoveFromList(Self:xlisttmp,
Self:MiMod:Nombre);
 Self:xlisttmp;    } );

/************** METHOD: MostrarConoc
**************/
MakeMethod( Agentes, MostrarConoc, [],
{ ClearTranscriptImage(Conocidos);
 EnumList(SendMessage(Self,Conocidos?), x,
    DisplayText(Conocidos, FormatValue("%s\t%s\r",
       x, SendMessage(Self,Informa,x,Relacion))));
} ).

/************** METHOD: Selecciona
**************/
/* "Selecciona un subconjunto de conocidos tal que
tengan entre
sus atributos un valor específico " */
MakeMethod( Agentes, Selecciona, [atributo valor ],
{ ClearList(Self:xlisttmp),
 EnumList( Self:Modelos, x,
   If( SendMessage(x,Es?, atributo, valor))
   Then
     AppendToList(Self:xlisttmp, x:Nombre)),
 Self:xlisttmp;   } ).

/************** METHOD: Ejecuta
**************/
/* "Ejecuta un ciclo en el agente " */
MakeMethod( Agentes, Ejecuta, [],
{  Monitor(" ");
   Monitor("EJECUTANDO "#Self).
   SendMessage(Self, Recibe);
   SendMessage(Self, Planificador);
   SendMessage(Self, Ejecutor);
   SendMessage(Self, Transmite);  } );

/************** METHOD: Salva   **************/
MakeMethod( Agentes, Salva, [],
{WriteInstance(Self);
 EnumList(Self:Modelos, x, SendMessage(x, Salva));
 EnumList(Self:BuzonR, x, WriteInstance(x));
 EnumList(Self:BuzonT, x, WriteInstance(x)),  } );

/************** METHOD: Performance?
**************/
/* "Regresa una lista con las medidas del desempeño
del agente
( Exitos, Fracasos, EnProceso, Pendientes) " */
MakeMethod( Agentes, Performance?, [],
{MakeSlot(Self:exito);
SetSlotOption(Self:exito, VALUE_TYPE, NUMBER);
Self:exito = 0;
MakeSlot(Self:frac);
SetSlotOption(Self:frac, VALUE_TYPE, NUMBER).
Self:frac = 0;
MakeSlot(Self:pend);
SetSlotOption(Self:pend, VALUE_TYPE, NUMBER);
Self:pend = 0;
MakeSlot(Self:enpro);
SetSlotOption(Self:enpro, VALUE_TYPE, NUMBER);
Self:enpro = 0;
EnumList(Self:MiMod:Compromisos, x,
 { If (x:Estado #= Terminado) Then Self:exito += 1
  Else
   { If (x:Estado #= Suspendido) Then Self:frac += 1
    Else
     { If (x:Estado #= NoIniciado) Then Self:pend += 1
      Else
       { If (x:Estado #= EnProceso) Then Self:enpro
+= 1
       }      }     } });
ClearList(Self:xlisttmp).
AppendToList(Self:xlisttmp,Self:exito,
         Self:frac, Self:enpro, Self:pend);
DeleteSlot(Self:exito);
DeleteSlot(Self:frac),
DeleteSlot(Self:enpro).
DeleteSlot(Self:pend),
Self:xlisttmp;  } ).

/************** METHOD: FDAR  **************
/* "Funcion de acuerdo al rol desempeñado " *
MakeMethod( Agentes, FDAR, [agent ],
{TRUE;  } );

/************** METHOD: FDAT  **************/
MakeMethod( Agentes, FDAT, [agent ],
{ TRUE;} );

MakeSlot( Agentes:Modelos );
SetSlotComment( Agentes:Modelos, "Modelos de
agentes conocidos por el agente" ).
SetSlotOption( Agentes:Modelos, MULTIPLE ).
SetSlotOption( Agentes:Modelos, VALUE_TYPE,
OBJECT );
SetSlotOption( Agentes:Modelos,
ALLOWABLE_CLASSES, MDAg ).
ClearList( Agentes:Modelos );
SetSlotOption( Agentes:Modelos, PROMPT,
"Modelos:" );
MakeSlot( Agentes:Acciones );
SetSlotComment( Agentes:Acciones, "Número total de
acciones ejecutadas por este agente." );
SetSlotOption( Agentes:Acciones, VALUE_TYPE,
NUMBER );
SetSlotOption( Agentes:Acciones,
MINIMUM_VALUE, 0 );
Agentes:Acciones = 0;
```

```
SetSlotOption( Agentes:Acciones, PROMPT, "Total de
trabajos:" );
MakeSlot( Agentes:MiMod );
SetSlotComment( Agentes:MiMod, "Nombre del
modelo asociado al agente" );
SetSlotOption( Agentes:MiMod, VALUE_TYPE,
OBJECT );
SetSlotOption( Agentes:MiMod,
ALLOWABLE_CLASSES, MDAg ).
MakeSlot( Agentes:AgendList );
SetSlotOption( Agentes.AgendList, MULTIPLE ),
ClearList( Agentes:AgendList );
MakeSlot( Agentes:Agenda ),
SetSlotOption( Agentes:Agenda, MULTIPLE );
ClearList( Agentes:Agenda ).
```

B.3 Clase: Actualizador

```
/************************************
     **** CLASS Actualizador
     ************************************/
MakeClass( Actualizador, Root );

'************** METHOD: Procesa
**************/
MakeMethod( Actualizador, Procesa, [men ].
{ If(Method?(Actualizador, men Tipo) )
  Then
    DelegateMessage(Self, Actualizador.men Tipo,
men)
Else
   {Advertencia("No existe el tipo de mensaje!!"),
     SendMessage(Self, Envia, men:Remitente, Error,0,
         men RespondaCon, men:Contenido).    }. } ).

/************** METHOD Comp **************/
/* "Comisiva Haz compromiso
Regresa confirmación Hecho " */
MakeMethod( Actualizador, Comp, [men ].
{  If( Member?(Self:MiMod:Capacidades,
GetNthElem(men.Contenido,1)))
  Then
  {/* Crea compromiso */
    Let [ meta    GetNthElem(men:Contenido,1)]
       [ tmax    GetNthElem(men:Contenido,2)]
       [ tinic   GetNthElem(men:Contenido,3)]
       [ cmp     SendMessage(CMPS, Crea)]
    { Monitor("Comprometiendose con
"#men:Remitente#" a realizar"#meta);
    SendMessage(cmp, Actualiza, men:Remitente,
              meta, tmax, EjAg:Reloj,tinic);
    SendMessage(Self:MiMod, Agrega, Compromisos,
cmp),
 * Revisa si le enviaron Recursos */
    If (LengthList(men.Contenido) > 3 )
    Then
     { ClearList(Self:xlisttmp);
       AppendToList(Self:xlisttmp, men:Contenido),
       RemoveFromList(Self:xlisttmp, meta);
       RemoveFromList(Self:xlisttmp, tinic);
       RemoveFromList(Self:xlisttmp, tmax);
       EnumList( Self:xlisttmp, r,
         { Self:xmod = SendMessage(RCS, Crea):
           SendMessage(Self:xmod, Actualiza, r,
RecEnviado, 1);
           SendMessage(Self:MiMod, Agrega, Recursos,
Self:xmod);
         });       };            };
/* Agrega hora de firma y contesta afirmativo *
    AppendToList(men:Contenido, EjAg:Reloj),
    SendMessage(Self, Envia,  men Remitente, Hecho,
           men:RespondaCon, men:Contenido).  }
 Else
  {Monitor("Negando un compromiso");
   SendMessage(Self, Envia, men Remitente, LoSiento,
          men:RespondaCon, men Contenido).  }.   } ).

/************** METHOD: Dile **************
/* "Informativa Respuesta de una pregunta " *
MakeMethod( Actualizador, Dile, [men ].
{ Monitor(men Remitente#"Dando información "),
 Let [ conocido    men Remitente]
    [ modconoc
SendMessage(Self,Modelo?,conocido)]
    [ atributo     GetNthElem( men Contenido,1)]
    [ valor        GetNthElem( men Contenido, 2) ]
 If( atributo #  Recursos )
 Then
  { If
Not(Member?(SendMessage(modconoc,Recursos?),val
or))
   Then
    { Self:xmod    SendMessage(RCS, Crea),
      SendMessage( Self:xmod, Actualiza, valor, NULL,
0),
      SendMessage(Self, Agrega,conocido,Recursos,
Self:xmod).    }   }
  Else
   If (atributo #    Metas)
   Then
    {If
Not(Member?(SendMessage(modconoc.Metas?),valor))
      Then
       { Self:xmod = Sendmessage(CMPS, Crea),
         SendMessage(Self:xmod, Actualiza, NULL,
valor, 0,0,0),
         SendMessage(Self,
Agrega,conocido,Compromisos,
          Self:xmod);        }.    }
   Else
     SendMessage(Self, Agrega,conocido,atributo,
valor).    } ).

************** METHOD: Preg **************
/* "Pregunta Pregunta si el atributo tiene un valor
determinado " *
MakeMethod( Actualizador, Preg, [men ].
{ Monitor(men:remitente#" Pregunta");
 Let [ atributo    GetNthElem( men Contenido,1)]
    [ valor      GetNthElem( men Contenido, 2) ]
{If (Slot?(MDAg, atributo) )
```

```
  Then
   {If ( SendMessage(Self:MiMod, Es?, atributo,
 valor))
     Then
       SendMessage(Self, Envia, men:Remitente,
       Dile,men:RespondaCon, men:Contenido)
     Else
       SendMessage(Self, Envia, men:Remitente,
        Niega, men:RespondaCon, men:Contenido);   }
  Else
   SendMessage(Self, Envia, men:Remitente, Error,
        men:RespondaCon, men:Contenido);
  };  } );

/************** METHOD: PuedeHacer
**************/
/* "Pregunta. Puede hacer la tarea que necesita los
recursos " */
MakeMethod( Actualizador, PuedeHacer, [men ],
{ Monitor(men:Remitente# "Pregunta
PUEDE_HACER? ");
 Let [ tarea    GetNthElem(men:Contenido, 1)]
  { MakeSlot(Self:recursos),
   SetSlotOption(Self:recursos, MULTIPLE);
   ClearList(Self:recursos);
   AppendToList(Self:recursos, men:Contenido),
   RemoveFromList(Self:recursos,tarea);
/* Revisa que pueda hacer tarea */
   If (Member?(Self:MiMod:Capacidades,tarea))
   Then
    {  If  Not (SendMessage(Self:MiMod,
              TengoRecursos, Self:recursos))
      Then
      { /* Regresa sólo recursos disponibles en el
agente */
        EnumList(Self:recursos, x,
         If Not (SendMessage(Self:MiMod, Es?,
Recursos, x))
          Then  RemoveFromList(men:Contenido, x));
};
      SendMessage(Self, Envia, men:Remitente,
        PuedoHacer, men:RespondaCon,
men:Contenido);}
   Else
      SendMessage(Self, Envia, men:Remitente,
        NoPuedo, men:RespondaCon, men:Contenido);
   DeleteSlot(Self, recursos) ;  }; } );

/************** METHOD: PuedoHacer
**************/
/* "Informativa. Puede hacer tarea." */
MakeMethod( Actualizador, PuedoHacer, [men ],
{  Monitor(men:Remitente#" Contesta
PUEDO_HACER ").
 If ( Not (Member?(SendMessage(Self,
Conocidos?),men:Remitente)))
 Then SendMessage(Self, AgregaConoc,
men:Remitente, Conocido);
 Let [modrem SendMessage(Self, Modelo?,
men:Remitente)]
   [tarea GetNthElem(men:Contenido,1)]
   { ClearList(Self:xlisttmp);
    AppendToList(Self:xlisttmp, men:Contenido);
    RemoveFromList(Self:xlisttmp, tarea);
    SendMessage(modrem, Agrega, Capacidades, tarea),
     EnumList(Self:xlisttmp, x,
      If Not(SendMessage(modrem, Es?, Recursos, x))
      Then
       { Self:xmod = SendMessage( RCS, Crea);
        SendMessage(Self:xmod, Actualiza, x,
material,0);
        SendMessage(modrem, Agrega, Recursos,
Self:xmod);
      } );      };     } );

/************** METHOD: CualEs  **************
/* "Pregunta.  Información del valor de un atributo " *
MakeMethod( Actualizador, CualEs, [men ],
{ Monitor(men:Remitente#"Preguntando valores "),
 Let [ atributo   GetNthElem( men:Contenido,1)]
   [conocido    men:Remitente]
 If ( Slot?(GetValue(Self:MiMod), atributo) )
 Then
  { ClearList(Self:xlisttmp),
   AppendToList(Self:xlisttmp, atributo),
   If( atributo #=Recursos )
   Then
    AppendToList(Self:xlisttmp,
         SendMessage(Self:MiMod,Recursos?))
     Else
    If (atributo #= Metas)
    Then
     AppendToList(Self:xlisttmp,
         SendMessage(Self:MiMod,Metas?))
    Else
     AppendToList(Self:xlisttmp,
         SendMessage(Self:MiMod, Informa, atributo))
 SendMessage(Self, Envia, conocido,
       ValorEs, men:RespondaCon  Self:xlisttmp),  }
Else
 SendMessage(Self, Envia, men:Remitente,
       Error, men:RespondaCon, men:Contenido),} ),

/************** METHOD: Niega  **************
/* "Informativa. Niega información" */
MakeMethod( Actualizador, Niega, [men ],
{ Monitor(men:Remitente#"Negando Información "),
Let [ atributo    GetNthElem( men:Contenido,1)]
  [ valor       GetNthElem( men:Contenido, 2) ]
 SendMessage(Self, Borra, men:Remitente,
      atributo, valor);  } );

/************** METHOD: ValorEs
**************/
/* "Informativa. Valor de un atributo " */
MakeMethod( Actualizador, ValorEs, [men ],
{ Monitor(men:Remitente#" Informa VALORES
"#Self);
 Let [modconoc   SendMessage(Self,
Modelo?,men:Remitente)]
   [atributo    GetNthElem( men:Contenido,1)]
   [infor      RemoveFromList(men:Contenido,
```

```
            GetNthElem(men:Contenido,1))]
 If( atributo #=Recursos )
 Then
  { EnumList(infor, x,
    If Not(SendMessage(modconoc, Es?, Recursos, x))
    Then
     { Self:xmod = SendMessage(RCS, Crea);
      SendMessage( Self:xmod, Actualiza, valor,
material, 0),
      SendMessage(modconoc, Agrega, Recursos,
Self:xmod);
    }      );    }
 Else
  {If (atributo #= Metas)
   Then
   {EnumList(infor, x,
      If Not(SendMessage(modconoc, Es?, Metas, x))
      Then
       { Self:xmod = SendMessage(CMPS, Crea);
        SendMessage( Self:xmod  Actualiza, NULL, x,
0,0,0),
        SendMessage(modconoc, Agrega,
Compromisos, Self:xmod),
      }    ),       }
   Else
   {If GetSlotOption(modconoc, atributo, SINGLE)
    Then
     SendMessage(modconoc, Actualiza, atributo,  ,
            GetNthElem( infor,1))
    Else
     SendMessage(modconoc, Actualiza, atributo,
infor),
   }.     }.    } ).

/************** METHOD  Hecho  **************
/* "Informativa  Hecho " */
MakeMethod( Actualizador, Hecho, [men ].
{ Monitor(men Remitente#"Confirmando
compromiso"),
Let [meta  GetNthElem(men Contenido, 1)]
  [modcon SendMessage(Self, Modelo?,
men Remitente)]
 { EnumList(modcon Compromisos, x,
  If (x:Meta #= meta)
  Then
   { x TFirm = GetNthElem(men Contenido,4),
    x:Estado = EnProceso,
   });      },      } ).

/************** METHOD  Error  **************
/* "Error  Mesaje erroneo." */
MakeMethod( Actualizador, Error, [men ].
{Monitor("ERROR "#men Remitente),} ).

/************** METHOD  CompTer
**************/
MakeMethod( Actualizador, CompTer, [men ].
{ /* Terminaron tarea asignada y envio recursos
resultados */
 Monitor(men Remitente#" termino tarea
"#GetNthElem(men Contenido,1)),
```

```
 Let [conoc  men:Remitente]
   [meta  GetNthElem(men:Contenido, 1)]
   [tconc  GetNthElem(men:Contenido, 2)]
  (If Not(conoc #= Self)
   Then
    {/* Actualiza modelo del conocido */
     Let [modcon SendMessage(Self, Modelo?, conoc)]
     EnumList(modcon:Compromisos, x,
      If (x:Meta #= meta)
      Then
       { x:TConc = tconc;
        x:Estado = Terminado;
        Self:xmod =x;        });     }.
/* Si hay resultados, genera recursos nuevos para ellos
*/
  If (LengthList(men.Contenido) > 2 )
  Then
   { Let [recur  GetNthElem(men:Contenido, 3)]
      [trec  GetNthElem(men Contenido, 4)]
      [cant  GetNthElem(men:Contenido, 5)]
      [contrec  SendMessage(RCS, Crea) ]
    { If Not(conoc #= Self)
      Then AppendToList(Self:xmod Result, recur, trec,
cant),
      SendMessage(contrec, Actualiza, recur, trec,
cant),
      SendMessage(Self:MiMod, Agrega, Recursos,
- contrec);
    },     },        },  } ).

************** METHOD  CancelaComp
**************/
 * "Comisiva  Indica la cancelación de un compromiso
previamente
pactado." */
MakeMethod( Actualizador, CancelaComp, [men ].
{ Monitor(men Remitente#" Cancelando
compromiso"),
 Let [conoc  men.Remitente]
   [meta  GetNthElem(men Contenido, 1)]
{  EnumList(Self MiMod Compromisos, x,
  If ( (x:Agente #= conoc) And (x Meta #= meta))
  Then
   { x:TConc = EjAg:Reloj,
    x:Estado = Suspendido,
   }),    };    } );

************** METHOD  NoPuedo
**************/
/* "Indica que no puede hacer la tarea." */
MakeMethod( Actualizador, NoPuedo, [men ].
{ Monitor(men:Remitente#" Contesta NoPuedo ");
 If (Member?(SendMessage(Self,
Conocidos?),men Remitente))
 Then
   Let [modrem SendMessage(Self, Modelo?,
men:Remitente)]
      [tarea GetNthElem(men:Contenido,1)]
    SendMessage(modrem, Borra, Capacidades, tarea).}
);
```

```
/************** METHOD: LoSiento
**************/
/* "El remitente avisa que no pudo concluir con éxito el
compromiso pactado." */
MakeMethod( Actualizador, LoSiento, [men ],
{ /* No pudo terminar un compromiso */
 Monitor(men:Remitente#" No termino tarea
"#GetNthElem(men:Contenido,1)),
 Let [conoc  men:Remitente]
    [meta  GetNthElem(men:Contenido, 1)]
    [tconc  GetNthElem(men:Contenido, 2)]
  {    If Not(conoc #= Self)
   Then
    {/* Actualiza modelo del conocido */
     Let [modcon SendMessage(Self, Modelo?, conoc)]
      {SendMessage(modcon, Borra, Capacidades,
meta);
       EnumList(modcon:Compromisos, x,
         If (x:Meta #= meta)
         Then
          { x:TConc = tconc,
           x:Estado   Cancelado,
          }):      },      },      },      } ),
```

B.4 Clase: Comunicador

```
/**********************************
     **** CLASS Comunicador
     **********************************
MakeClass( Comunicador, Root ),

/************** METHOD: Recibe  **************/
MakeMethod( Comunicador, Recibe, [],
{ /* Función que depende del ROL asigando *
 SendMessage( Self, FDAR, Self),
 Let [ nmen  LengthList(Self:BuzonR)]
 For n From 1 To nmen  Do
  Let [men GetNthElem(Self:BuzonR,1)]
   {RemoveFromList(Self:BuzonR, men),
    If(IsAKindOf?(men,MSG))
    Then
     { Monitor( "Recibe mensaje "# men:Tipo#
          " de "# men:Remitente );
      Self:recibos += 1,
      DelegateMessage(Self, Actualizador, Procesa,
men),
      SendMessage(MSG,Destruye,men);},   };} ),

/************** METHOD: Transmite
**************/
MakeMethod( Comunicador, Transmite, [],
{ Let [ nmen  LengthList(Self:BuzonT)]
 For n From 1 To nmen  Do
  Let [men GetNthElem(Self:BuzonT,1)]
   {RemoveFromList(Self:BuzonT, men),
    If (SendMessage(DIR,Existe, men:Receptor) )
    Then
     { AppendToList(men:Receptor:BuzonR, men);
      If Not( men:Receptor #= Self)
       Then Self:envios+=1;    }
     Else
      { Advertencia("Agente desconocido ");
       SendMessage(MSG,Destruye,men);
      }:  }:
 SendMessage(Self, FDAT, Self);    } );

/************** METHOD: Envia  **************/
MakeMethod( Comunicador, Envia, [agente tipo idres
contenido ],
{ If (agente #= Todos)
 Then
  { MakeSlot(Self, Todos);
   SetSlotOption(Self, Todos, MULTIPLE);
  ClearList(Self:Todos);
  GetInstanceList(EstOrg, Self:Todos) ;
  AppendToList(Self:Todos, DIR:agentes),
  RemoveFromList(Self:Todos, Self);
  RemoveFromList(Self:Todos, USER);
  EnumList( Self:Todos, x,  SendMessage(Self,
   EnviaUno,  x, tipo, Self:nmen, idres, contenido)),
  Self:nmen += 1,
  DeleteSlot(Self,Todos);  }
 Else
  {/* Envía a un grupo de agentes definidos por una
relación */
   If(Member?(DepAg:relaciones, agente))
   Then
    {EnumList(SendMessage(Self, Selecciona,
Relacion,agente) ,
      x, SendMessage(Self, EnviaUno, x,
              tipo,Self:nmen, idres, contenido)),    }
  Else
    SendMessage(Self, EnviaUno, agente,
              tipo,Self:nmen, idres, contenido),
  },    } );

 ************** METHOD: EnviaUno
**************
MakeMethod( Comunicador, EnviaUno, [agente tipo
idmen idres contenido ],
{Let [nmen MSG:nmen]
  [ modelo SendMessage(MSG, Crea)]
  { SetValue(modelo, Receptor, agente),
   ClearList(modelo:Contenido);
   AppendToList(modelo:Contenido, contenido);
   SetValue(modelo, Tipo, tipo);
   SetValue(modelo, Remitente, Self);
   SetValue(modelo, RespondaCon, idmen);
   SetValue(modelo, EnRespA, idres);
   AppendToList(Self:BuzonT, modelo);
  nmen;    };} );

MakeSlot( Comunicador:BuzonR ),
SetSlotOption( Comunicador:BuzonR, MULTIPLE ),
SetSlotOption( Comunicador:BuzonR, VALUE_TYPE,
OBJECT );
SetSlotOption( Comunicador:BuzonR,
ALLOWABLE_CLASSES, MSG );
ClearList( Comunicador:BuzonR ),
MakeSlot( Comunicador:BuzonT );
```

```
SetSlotOption( Comunicador:BuzonT, MULTIPLE );
SetSlotOption( Comunicador:BuzonT, VALUE_TYPE,
OBJECT );
SetSlotOption( Comunicador:BuzonT,
ALLOWABLE_CLASSES, MSG );
ClearList( Comunicador:BuzonT );
MakeSlot( Comunicador:envios );
SetSlotOption( Comunicador:envios, VALUE_TYPE,
NUMBER );
SetSlotOption( Comunicador:envios,
MINIMUM_VALUE, 0 );
Comunicador:envios = 0;
MakeSlot( Comunicador:recibos );
SetSlotOption( Comunicador:recibos, VALUE_TYPE,
NUMBER ),
SetSlotOption( Comunicador:recibos,
MINIMUM_VALUE, 0 ),
Comunicador:recibos = 0;
```

B.5 Clase: Planes

```
/***********************************
**** CLASS PLANES
**** Representación de los planes.
***********************************/
MakeClass( PLANES, Root );
SetClassComment( PLANES, "Representación de los
planes " );

/************** METHOD: Crea **************
/* "Crea un nuevo plan. Distingue la creación de planes
y de instancias de planes. Requiere como argumento
el nombre de la instancia y del padre
Regresa nombre del plan creado." */
MakeMethod( PLANES, Crea, [tarea ],
{ If( tarea#=INST )
 Then
 { Let[ inst = Self#PLANES:nplan]
  {PLANES.nplan+=1;
   MakeInstance(inst, Self);
   inst;  }  }
Else
 { MakeClass( tarea,PLANES);
 tarea.    }.} ).

/************** METHOD: Iniciar **************/
/* "Metodo para ejecutarse antes de iniciar el plan " */
MakeMethod( PLANES, Iniciar, [agent ],
{TRUE. } ).

/************** METHOD: Terminar
**************/
/* "Metodo para ejecutar cuando se termina un plan.
agent es el agente que tiene asignado el plan." */
MakeMethod( PLANES, Terminar, [agent ],
{ TRUE.} ).

/************** METHOD: Destruye
**************/
/* "Distingue entre destrucción de una instancia
y destrucción de una clase." */
MakeMethod( PLANES, Destruye, [plan ],
{ If (Instance?(plan))
 Then DeleteInstance(plan)
 Else
  { MakeSlot(Self:listtmp);
   SetSlotOption(Self:listtmp, MULTIPLE);
   AppendToList(Self:listtmp, plan:Fases);
   DeleteClass(plan);
   EnumList(Self:listtmp, x,
    { If Not (SendMessage(ModPlan, FasePlan?, x))
     Then DeleteClass(x)   });
   DeleteSlot(Self:listtmp);    };    } ).

MakeSlot( PLANES:Fases );
SetSlotComment( PLANES:Fases, "Lista de pasos del
plan" );
SetSlotOption( PLANES:Fases, MULTIPLE ).
SetSlotOption( PLANES:Fases, VALUE_TYPE,
OBJECT );
SetSlotOption( PLANES:Fases,
ALLOWABLE_CLASSES, FASES ).
ClearList( PLANES:Fases );
MakeSlot( PLANES:Comp );
SetSlotComment( PLANES:Comp, "Compromiso
asociado a la instancia del plan " );
- MakeSlot( PLANES:nplan ).
SetSlotOption( PLANES:nplan, INHERIT, FALSE ).
SetSlotComment( PLANES:nplan, "Numero de
instancia de un plan especifico." ),
SetSlotOption( PLANES:nplan, VALUE_TYPE,
NUMBER ),
SetSlotOption( PLANES:nplan, MINIMUM_VALUE,
0 );
PLANES:nplan = 0;
```

B.6 Clase: Fases

```
/***********************************
**** CLASS: FASES
***********************************/
MakeClass( FASES, Root );

/************** METHOD: Precond
**************/
/* "Precondiciones que deben cumplirse antes de iniciar
la acción.
" */
MakeMethod( FASES, Precond, [plan agent ],
{TRUE;} );

/************** METHOD: Accion **************/
/* "Accion ejecutada en el agente (agent) en este paso."
*/
MakeMethod( FASES, Accion, [plan agent ],
{TRUE;} );
```

B.7 Clase: Organizaciones

```
/***********************************
     **** CLASS: EstOrg
     **** Representando a las organizaciones
*********************************/
MakeClass( EstOrg, Comunicador ).
SetClassComment( EstOrg, "Representando a las
organizaciones" ).

/************** METHOD: Crea  **************/
MakeMethod( EstOrg, Crea, [nombre ],
{ MakeInstance( nombre, EstOrg);
 nombre:MiMod = SendMessage(MDAg, Crea,
nombre, mio);
 nombre,  } ).

/************** METHOD: Destruye
**************/
MakeMethod( EstOrg, Destruye, [nomb ],
{  If( Instance?(nomb) And
(IsAKindOf?(nomb,EstOrg)))
Then
 { SendMessage(ROLES, Destruye, nomb:Manejador),
  EnumList(nomb:Trabajadores, x,
      SendMessage(ROLES, Destruye, x)),
  SendMessage(MDAg, Destruye, nomb:MiMod)
  DeleteInstance(nomb),    },   } );

/************** METHOD: Modifica
**************/
/* "Modifica interactivamente una organización " */
MakeMethod( EstOrg, Modifica, [],
{PostInputForm("Modificando Organizacion "+Self,
   Self:Manejador, "Rol del manejador  ",
   Self:Trabajadores, "Rol de trabajadores  "),
 SendMessage(Self:MiMod, Modifica),} ).

/************** METHOD: Mostrar
**************
MakeMethod( EstOrg, Mostrar, [],
{   ClearTranscriptImage(OrgOrg);
   DisplayText(OrgOrg,
     FormatValue("%s",Self:MiMod:Org)),
   DespList( Self:MiMod:Roles, RolOrg);
   DespList( Self:MiMod:Capacidades, HabOrg);
   ClearTranscriptImage(RecOrg);
   EnumList( Self:MiMod.Recursos, x,
       DisplayText(RecOrg,
          FormatValue("%s\t%s\t%s\t%d\r",
          x,x:Nombre, x:Tipo, x:Cantidad))).
   ClearTranscriptImage(MetOrg);
   EnumList( Self:MiMod Compromisos,x,
     DisplayText(MetOrg,

FormatValue("%s\t%s\t%s\t%u\t%u\t%u\t%u\t%s\n",
     x,x:Agente, x:Meta, x:TFirm, x:TMax,
      x:TInic, x:TConc, x:Estado)) ).
   ClearTranscriptImage(EdoOrg).
   If( SendMessage( Self, Instanciada?))
  Then
    DisplayText(EdoOrg,
       FormatValue("INSTANCIADA\r"))
  Else
    DisplayText(EdoOrg,
       FormatValue("NoINSTANCIADA\r"));
 ShowImage(RolMng);
 ClearTranscriptImage(RolMng).
 ShowImage(MngOrg).
 ClearTranscriptImage(MngOrg);
 If Not (Null?(Self:Manejador))
 Then  SendMessage(Self:Manejador, MMiemb,
RolMng,MngOrg);
 ShowImage(RolTrab);
 ClearTranscriptImage(RolTrab).
 ShowImage(TrabOrg).
 ClearTranscriptImage(TrabOrg).
 EnumList(Self:Trabajadores, x,
   SendMessage(x, MMiemb, RolTrab, TrabOrg)),} ).

/************** METHOD: Instancia
**************/
/* "Instanciando los roles de manejadores y
trabajadores " */
MakeMethod( EstOrg, Instancia, [],
{ MakeSlot(Self, listtmp),
 SetSlotOption(Self:listtmp, MULTIPLE),
 /* Buscando un manejador para la organizacion */
If Not (Null?(Self:Manejador))
 Then SendMessage(Self:Manejador, Instancia, Self).
If(LengthList(Self:Trabajadores)!  0)
Then
/* Buscando trabajadores para la organizacion */
 EnumList(Self:Trabajadores, x,
      SendMessage(x, Instancia, Self)).
/* Actualizando conocimiento en la Organización */
Let [ mang  GetNthElem(Self:Manejador:Miembros, 1)
]
{ EnumList(Self:Trabajadores,x,
   { ClearList(Self:listtmp),
    AppendToList(Self:listtmp, x:Miembros),
    EnumList(Self:listtmp, y,
     { Let [modtrab  SendMessage(MDAg, Crea, y,
MiTrab)]
         [modjefe  SendMessage(MDAg, Crea, mang,
Jefe)]
        {SendMessage(y:MiMod, Copia, modtrab),
         AppendToList(mang:Modelos, modtrab),
         SendMessage(mang:MiMod, Copia, modjefe),
         AppendToList(y:Modelos, modjefe),
        };      });   });
/*Definiendo patron de comportamiento del manejador
*/
/* Toma mensajes, compromisos y recursos de la
Organizacion */
 MakeMethod(mang, FDAR,[agent] ,
 {SendMessage(agent:MiMod:Org, Recibe),
  EnumList(agent:MiMod:Org:MiMod:Compromisos,
y,
   { If(y:Estado == NoIniciado)
     Then
```

```
      {y:Estado = EnProceso;
      Let [comp  SendMessage(CMPS, Crea)]
        { SendMessage(comp, Actualiza, agent, y:Meta,
            y:TMax, y:TFirm, y:TInic);
          SendMessage(agent:MiMod, Agrega,
  Compromisos, comp);
          }     }    } };
  EnumList(agent:MiMod:Org:MiMod:Recursos, y,
    SendMessage(agent:MiMod, Agrega, Recursos, y) );
  ClearList(agent:MiMod:Org:MiMod:Recursos);   }),
/* Revisa por metas cumplidas en la Organización */
 MakeMethod(mang, FDAT,[agent] ,
  {
 EnumList(agent:MiMod:Org:MiMod:Compromisos, y,
  { If (y:Estado #= EnProceso)
    Then
     EnumList(agent:MiMod:Compromisos, z,
     { If ( (z:Meta #= y:Meta) And
         (z:Estado #= Terminado) )
       Then
       { y:Estado = Terminado;
        y:TConc = EjAg:Reloj;
        MakeSlot(agent:lista2);
        SetSlotOption(agent:lista2, MULTIPLE);
        ClearList(agent:lista2);
        AppendToList(agent:lista2, y:Meta, y:TConc,
y:Result);
        SendMessage(agent:MiMod:Org, Envia,
y:Agente, CompTer, 0,agent:lista2);
        DeleteSlot(agent:lista2);
      }       }),   } ),
  SendMessage(agent:MiMod:Org, Transmite);   });
},
DeleteSlot(Self:listtmp);} );

/************** METHOD: Instanciada?
**************/
/* "Verifica que una organización esté completamente
instanciada " */
MakeMethod( EstOrg, Instanciada?, [],
{If (Not (Null?(Self:Manejador))) And
  SendMessage(Self:Manejador, Instanciada?) And
  SendMessage(Self, TodosTrab? )
Then   TRUE
Else   FALSE;} );

/************** METHOD: DesInstancia
**************/
/* "Desinstancia una organización. Borra roles,
organización, y elimina modelos
de los agentes instanciados a esta organización." */
MakeMethod( EstOrg, DesInstancia, [],
{ /* Desinstancia al manejador */
 If Not (Null?(Self:Manejador))
 Then SendMessage(Self:Manejador, DesInstancia);
 If (LengthList(Self:Trabajadores)!= 0)
 Then
  EnumList(Self:Trabajadores, x,
     SendMessage(x, DesInstancia));} );

/************** METHOD: Salva   **************/
 MakeMethod( EstOrg, Salva, [],
 { WriteInstance(Self:MiMod);
 WriteInstance(Self);
 EnumList(Self:BuzonR, x,
        WriteInstance(x));
 EnumList(Self:BuzonT, x,
        WriteInstance(x));   } ).

/************** METHOD: TodosTrab?
**************/
/* "Checa si todos los trabajadores están instanciados "
*/
MakeMethod( EstOrg, TodosTrab?, [],
{ MakeSlot (Self, inst);
 SetSlotOption(Self, inst, VALUE_TYPE,
BOOLEAN),
 Self:inst = TRUE;
 EnumList( Self:Trabajadores, x,
  { If Not (SendMessage( x, Instanciada? ) )
    Then
     Self:inst   FALSE;  }),
Self:inst,    } ),

/************** METHOD: Recibe  **************/
MakeMethod( EstOrg, Recibe, [],
{ Let [ nmen  LengthList(Self:BuzonR)]
  For n From 1 To nmen  Do
  Let [men GetNthElem(Self:BuzonR,1)]
   {RemoveFromList(Self:BuzonR, men),
   If (IsAKindOf?(men,MSG))
   Then
    { Monitor( "Recibe mensaje para la organizacion
"# men:Tipo#
           " de "# men:Remitente#" n" ),
     Self:recibos += 1,
/* Procesa mensajes de la organización */
    If( Member?( Self:MenOrg, men:Tipo))
    Then
     {  DelegateMessage(Self, Actualizador, Procesa,
men);
      SendMessage(MSG,Destruye,men);       }
    Else
/* Transfiere mensajes al manejador */
     { Let [mang
GetNthElem(Self:Manejador:Miembros,1)]
      AppendToList( mang:BuzonR, men);
     };    };       };   } ).

/************** METHOD: Transmite
**************
MakeMethod( EstOrg, Transmite, [],
{   Let [ nmen  LengthList(Self:BuzonT)]
 For n From 1 To nmen  Do
  Let [men GetNthElem(Self:BuzonT,1)]
   {RemoveFromList(Self:BuzonT, men),
    If (SendMessage(DIR,Existe, men:Receptor) )
    Then
     { AppendToList(men:Receptor:BuzonR, men),
      If Not( men:Receptor #= Self)
      Then Self:envios += 1;        ;
    Else
```

```
    { Advertencia("Agente desconocido ");
    SendMessage(MSG,Destruye,men);
    };    };    } );

MakeSlot( EstOrg:MiMod );
SetSlotOption( EstOrg:MiMod, VALUE_TYPE,
OBJECT ),
SetSlotOption( EstOrg:MiMod,
ALLOWABLE_CLASSES, MDAg );
MakeSlot( EstOrg:Manejador );
MakeSlot( EstOrg:Trabajadores ),
SetSlotOption( EstOrg:Trabajadores, MULTIPLE );
ClearList( EstOrg:Trabajadores ).
MakeSlot( EstOrg MenOrg );
SetSlotOption( EstOrg:MenOrg, MULTIPLE );
SetValue( EstOrg:MenOrg, Comp, PuedeHacer,
CualEs, Preg );
```

B.8 Clase: Roles

```
/************************************
     **** CLASS. ROLES
     **** Define un rol asociado a una estructura
organizacional
     ***********************************/
MakeClass( ROLES, Root );

/************** METHOD  Crea  **************/
MakeMethod( ROLES, Crea, [],
  {   Let [nombre Rol # Self.nrol]
     { Self.nrol += 1,
       MakeInstance( nombre, ROLES ),
       nombre;        },    } ).

 ************** METHOD. Destruye
**************/
MakeMethod( ROLES, Destruye, [nomb ],
  {If ( Instance?( nomb ) And IsAKindOf?( nomb,
ROLES ) )
    Then {
       SendMessage( nomb, DesInstancia );
       DeleteInstance( nomb );          };   } );

************** METHOD. Modifica
**************/
MakeMethod( ROLES, Modifica, [],
  { PostInputForm( "Modificando ROL " # Self,
Self:Nombre, "Nombre: ",
     Self:Habilidades, "Habilidades: ", Self:NMmb,
"Número de miembros ",
     Self:Miembros, "Miembros: " ),    } ).

/************** METHOD: Mostrar
**************/
MakeMethod( ROLES, Mostrar, [],
  { MapEscapeKey( QuitaRol );
  ShowImage( HabRol ),
  DespList( Self.Habilidades, HabRol );
  ShowImage( NMbRol ),
    ClearTranscriptImage( NMbRol );
    DisplayText( NMbRol, FormatValue( "%d\r",
 Self:NMmb ) );
    ShowImage( NombRol );
    ClearTranscriptImage( NombRol );
    ShowImage( MbRol );
    ClearTranscriptImage( MbRol );
    SendMessage( Self, MMiemb, NombRol, MbRol );
 } );

/************** METHOD: Instancia
**************/
/* "Instancia agentes a este rol." */
MakeMethod( ROLES, Instancia, [org ],
 {  MakeSlot( Self, miemb );
   SetSlotOption( Self, miemb, VALUE_TYPE,
 NUMBER );
   MakeSlot( Self, listtmp );
   SetSlotOption( Self, listtmp, MULTIPLE ),
   GetInstanceList( EstOrg, Self:listtmp ),
   RemoveFromList( Self:listtmp, org ),
   AppendToList( Self:listtmp, DIR agentes ),
   Self:miemb = Self.NMmb - LengthList(
 Self:Miembros );
   While (( Self:miemb > 0 ) And ( LengthList(
 Self:listtmp )
                               != 0 ) ))
       { Let [y GetNthElem( Self:listtmp, 1 )]
          {If SubConj?( Self.Habilidades,
 y MiMod Capacidades )
            Then {
               y MiMod:Org = org;
               If Not( Member?( y MiMod Roles,
 Self Nombre ) )
                 Then AppendToList( y MiMod Roles,
 Self Nombre ),
               AppendToList( Self Miembros, y ),
               Self:miemb = Self miemb - 1,        },
         RemoveFromList( Self.listtmp, y ),    },
 },
   DeleteSlot( Self:listtmp );
   DeleteSlot( Self:miemb );    } );

/************** METHOD: DesInstancia
**************/
MakeMethod( ROLES, DesInstancia, [],
  { EnumList( Self:Miembros, x,
     { ResetValue( x:MiMod:Org );
      ResetValue( x:MiMod:Roles );        } );   } );

/************** METHOD: MMiemb
**************/
MakeMethod( ROLES, MMiemb, [ventnomb
ventmiemb ],
  { DisplayText( ventnomb, FormatValue( "%s\r",
Self:Nombre ) );
     EnumList( Self:Miembros, x, DisplayText(
ventmiemb,
                       FormatValue( "%s\r", x ) ) ),    }
).
```

```
/************** METHOD: Instanciada?
**************/
MakeMethod( ROLES, Instanciada?, [],
   {  Self:NMmb <= LengthList( Self:Miembros );
   } );

MakeSlot( ROLES:Nombre );
MakeSlot( ROLES:NMmb );
SetSlotOption( ROLES:NMmb, VALUE_TYPE,
NUMBER );
SetSlotOption( ROLES:NMmb, MINIMUM_VALUE,
1 );
ROLES:NMmb = 1;
MakeSlot( ROLES:Habilidades );
SetSlotOption( ROLES:Habilidades, MULTIPLE );
ClearList( ROLES:Habilidades );
MakeSlot( ROLES:Miembros );
SetSlotOption( ROLES:Miembros, MULTIPLE );
SetSlotOption( ROLES:Miembros, VALUE_TYPE,
OBJECT );
SetSlotOption( ROLES:Miembros,
ALLOWABLE_CLASSES, EstOrg, Agentes );
ClearList( ROLES:Miembros );
MakeSlot( ROLES:nrol );
SetSlotOption( ROLES:nrol, VALUE_TYPE,
NUMBER );
SetSlotOption( ROLES:nrol, MINIMUM_VALUE, 0 );
ROLES:nrol = 0;
```

Glosario

Agentes. Entidades con identidad propia que perciben e inciden en su medio ambiente y son capaces de resolver problemas a pesar de no tener un conocimiento total del mismo.
Creencias. Expresan la representación que el agente hace de sí mismo y del estado actual de su medio ambiente.
Coherencia. se refiere a *que "tan bien" el sistema se está comportando como una unidad* y se mide en base a la eficiencia del sistema mismo.
Complejidad. Demanda excesiva de racionalidad.
Comunicación. Mecanismo para establecer un diálogo entre dos o más agentes por medio de una serie de intercambios de mensajes.
Cooperación. *Coordinación de las capacidades individuales de cada agente en el sistema.*
Coordinación. El acto de trabajar juntos armoniosamente o bien el acto de manejar la interdependencia entre actividades desempeñadas por varios agentes para alcanzar una meta común.
Deseos. Son una noción abstracta que expresan las preferencias del agente sobre futuros estados de sí mismo y de su medio ambiente.
FIM. Funciones interpretadoras de mensajes, usadas para reconocer los diferentes tipos de mensajes que pueden llegar a un agente.
Intenciones. Dado que un agente está acotado en recursos, no puede proponerse todas sus metas u opciones a la vez. Aún si el conjunto de metas es consistente, frecuentemente es necesario seleccionar ciertas metas (o un conjunto de metas) para comprometerse con ellas. Así las intenciones actuales de los agentes están descritas por un conjunto de metas seleccionadas.
Interacción. Algún tipo de acciones colectivas donde un agente puede llevar a cabo una acción o crear una decisión en base a la presencia o conocimiento de otros agentes en su medio ambiente.
Medio ambiente. Es el mundo exterior de un agente, puede estar formado por otros agentes que interactúan con él o por cualquier otro factor que afecte su conducta.
Metas. Representan un subconjunto de deseos consistentes que el agente tiene como propósito.
Modelo. Representación mental de los atributos de otros agentes en el sistema.
Organización. Establecimiento de una estructura de interacción en donde los agentes mantienen ciertas responsabilidades y que procesan e intercambian información, en un intento por conseguir una o más metas
Planes. Secuencia de acciones, algunas de las cuales pueden tener restricciones, para lograr un fin.
Procesamiento. Propiedad de los agentes para producir acciones en base a su conocimiento y a la interacción con otros agentes en su medio ambiente.
Sistemas Cooperativos. Sociedad de agentes distribuidos que se unen para encontrar la solución a un problema dado

Bibliografía

[BG89] A. Bond. L. Gasser. An Analysis of Problems and Research in DAI. Readings in Distributed Artificial Inteligence. Pag 3-35. 1989.

[BK96] Robert W. Blanning and David R. King. Organizational Intelligence. AI in organizational Design, Modeling, and Control.Computer Society Press. 1996.

[Bla96] Robert W. Blanning. Organization Design as Knowledge Engineering in Organizational Intelligence. AI in organizational Design, Modeling, and Control.Computer Society Press. 1996.

[Cue91] José Cuena. Arquitecturas Cognitivas. II Escuela Internacional de Invierno en Temas Selectos de la Computación. Xalapa Ver. 1991.

[DAR93] The DARPA Knowledge Sharing Initiative External Interfaces Working Group. DRAFT Specification of the KQML Agent-Communication Language. June 1993.

[DLC89] Durfee, Lesser Corkill. Coherent Cooperation Among Communicating Problem Solvers. Readings in Distributed Artificial Intelligence. pag. 268-284. 1989.

[DS89I] R. Davis and R.G. Smith. Negotiation as a Metaphor for Distributed Problem Solving. Readings in Distributed Artificial Inteligence. pag 333-355. 1989.

[DS89II] R. Davis and R.G. Smith. Framework for cooperation in Distributed Problem Solving. Readings in Distributed Artificial Inteligence. pag 61-70. 1989.

[FF94] Tom Finin. Richard Fritzon. KQML as an Agent Communication Languaje. The Proceedings of the Third International Conference on Information and Knowledge Management. ACM Press. November 1994.

[Fox89] Mark A. Fox. An Organizational View of Distributed System. Readings in Distributed Artificial Inteligence. Pag 140-150. 1989.

[Gar91] Francisco Garijo. Modelos Cooperativos de Resolución de Problemas. II Escuela Internacional de Invierno en Temas Selectos de la Computación. Xalapa Ver. 1991.

[GBH87] Les Gasser, Carl Braganza and Nava Herman. MACE: A flexible Testbed for Distributed AI Research. Distributed Artificial Intelligence. Pitman, London. Pag. 119-152. 1987.

[GGR89] MR. Genesereth, ML. Ginsberg and JS Rosenschein. Cooperation without communication. Readings in Distributed Artificial Intelligence. pag 220-226. 1989.

[Gru91] Grudin, J. CSCW Introduction. CACM v34. 1991.

[HC93] H. Carvajal, F. López, MG. López, L. Zamora y LA. Gónzález. Sistemas cooperativos: Un marco de referencia. Memorias de la X Reunión Nacional de Inteligencia Artificial. 1993.

[Int92] IntelliCorp, Inc. Kappa-PC. 1992

[Kat96] Miguel Katrib. Programación Orientada a Objetos en C++. X view. 1996

[LaFi93] Yannis Labrou. Tim Finin. A semantics approach for KQML a general purpose communications languaje for software agents. University of Maryland.1993

[LHK79] Douglas B.Lenat, Frederick Hayes-Roth, Philip Klarh. Cognitive Economy in Artificial intelligence Systems. Memo HPP-79-15, Heuristic Programming Project, Computer Science Dept., Stanford Univ. Stanford, CA. 1979.

[Mal89] T.W. Malone. Modeling Coordination in Organizations and Markets. Readings in Distributed Artificial Inteligence. Pag 151-158. 1989.

[Mas93] R.M. Mason. Strategic Information System: Use of Information Technology in a Learning Organization. Proceedings 26th Hawaii International Conference on System Sciences, Vol IV. 1993.

[MC90] Thomas W. Malone & Kevin Crowston. What is coordination Theory and How Can It Help Design Cooperative Work Systems?. Readings in GROUPWARE and CSCW. Pag 375-388. 1990.

[MR92] T.W. Malone and J.F. Rockart. Information Technology and the New Organization. Proceedings 25th Hawaii International Conference on System Sciences. Vol IV. 1992.

[Müll96] Jörg P. Müller. The Design of Intelligent Agents. A Layered Approach. Springer. 1996.

[SHH91 M.J. Shaw, B. Harrow, and S. Herman. Proceeding 24th Hawaii International Conference on System Sciences. Vol IV. 1993

[Sho90] Yoav Shoham. Agent-Oriented Programming. Technical Report. Stanford University. 1990.

[TBS89] Tanenbaum, Bal y Steiner. Programming Languajes for Distributed Computing System. ACM Computing Surveys. Vol 21. No 3. September 1989.

[Wer87] Eric Werner. Cooperating Agents: A unified theory of communication and social estructure. Distributed Artificial Intelligence. Pitman, London. Pag. 3-36. 1987.