UNIVERSIDAD DE LEÓN

Departamento de Ingeniería Eléctrica
y de Sistemas y Automática

Tesis doctoral

Técnicas para despliegue de arquitectura distribuida en sistemas expertos basados en reglas empleando el paradigma multiagente

Francisco José Aguayo Canela

Dirigida por:
Dr. ***Isaías García Rodríguez***

Septiembre ~ 2017

Doctorando: Francisco José Aguayo Canela
Director de tesis: Dr. Isaías García Rodríguez

Título tesis (es_ES): Técnicas para despliegue de arquitectura distribuida en sistemas expertos basados en reglas empleando el paradigma multiagente.

Departamento de Ingeniería Eléctrica y de Sistemas y Automática
Escuela de Ingenierías Industrial e Informática
Campus de Vegazana — 24007 León

Universidad de León
http://www.unileon.es

universidad de León

Departamento de Ingeniería Eléctrica
y de Sistemas y Automática

Técnicas para despliegue de arquitectura distribuida en sistemas expertos basados en reglas empleando el paradigma multiagente

Tesis doctoral

Francisco José Aguayo Canela — *Doctorando*

Dr. Isaías García Rodríguez — *Director de tesis*

Escuelas de Ingenierías Industrial e Informática

Septiembre ~ 2017

Doctorando: Francisco José Aguayo Canela
Director de tesis: Dr. Isaías García Rodríguez

Título tesis (es_ES): Técnicas para despliegue de arquitectura distribuida en sistemas expertos basados en reglas empleando el paradigma multiagente.

Departamento de Ingeniería Eléctrica y de Sistemas y Automática
Escuela de Ingenierías Industrial e Informática
Campus de Vegazana — 24007 León

Universidad de León
http://www.unileon.es

Deploying Production Systems on distributed architecture using the Multiagent paradigm: Applied Techniques

Doctoral Thesis

Department of Electric Engineering,
Systems and Automatic

Ph.D. Student:
Francisco José Aguayo Canela

Thesis Director:
Ph.D. Isaías García Rodríguez

September, 2017

School of Industrial Engineering and Information Technology
University of Leon

Doctorando: Francisco José Aguayo Canela
Director de tesis: Dr. Isaías García Rodríguez

Título tesis (es_ES): Técnicas para despliegue de arquitectura distribuida en sistemas expertos basados en reglas empleando el paradigma multiagente.

Departamento de Ingeniería Eléctrica y de Sistemas y Automática
Escuela de Ingenierías Industrial e Informática
Campus de Vegazana — 24007 León

Universidad de León
http://www.unileon.es

A mí buena amiga y esposa, Carmen,
a mis queridos hijos; Currito y Luisito

A mis dos Universidades:
Sevilla (us.es) y León (unileon.es)

Agradecimientos

Quiero dedicar este primer párrafo de agradecimientos a los miembros del *Departamento de Ingeniería Eléctrica y de Sistemas y Automática* por el espíritu que me han transmitido; que ha sido particularmente cálido aún siendo un departamento de *Ingeniería Industrial*. Y expresar también en estas líneas, mi admiración y agradecimiento a *Ángel Alonso Álvarez*, quien puso a mi disposición los recursos humanos y los medios tecnológicos que han permitido llevar adelante y finalmente concluir, esta investigación. Envío además, un abrazo a los miembros más cercanos del departamento; *José M. Alija, Carmen Benavides* (ella, fuera de España en estos momentos), *Francisco J. Rodríguez* y *Ramón A. Fernández*; por sus palabras llenas de ánimo que resultaron ser el elemento imprescindible para afrontar el reto que nos plateamos hace ya, más de cuatro años.

Este párrafo central es para *Isaías García Rodríguez* y *Héctor Aláiz Moretón*, por compartir conmigo sus objetivos de incorporar en los *agentes-software* las capacidades de inferencia que exponen los *sistemas expertos basados en reglas*. A ambos, les remito este abrazo por escrito y mi agradecimiento por la guía técnica y el apoyo humano que con tanta excelencia y generosidad me han brindado estos años.

Y como *ingeniero mecánico*, me gustaría concluir con un agradecimiento lleno de complicidad y de admiración a los autores de los *paquetes de software* empleados en la elaboración de las *Técnicas* implementadas en esta tesis: *profesionales*, *profesores*, *expertos* e incluso *Doctores* que han permanecido ahí, en la distancia pero conectados y accesibles por *correo* o por el *Blog*. Son un ejemplo vivo de discreción, conocimientos enciclopédicos y generosidad para aquellos que deseamos incorporarnos a esta rama del conocimiento: *Fabio Bellifemine*, *Giovanni Rimassa*, *Ernest Friedman-Hill*, *Giovanni Caire*, *Georgios Meditskos* y *Gary Riley*. Sus indicaciones han permitido lograr integraciones versátiles de *CLIPS* y *Jess*, dentro del modelo de ejecución de los *agentes* en la plataforma *JADE*.

Francisco José Aguayo Canela

Alumno de doctorado
Universidad de León

Miembros del tribunal:

_____	_____
_____	_____
_____	_____
_____	_____
_____	_____

Metadatos:

Thesis Deposit Number: **2012.0000.55**

Title (en_US):	Deploying Production Systems on distributed architecture using the Multi-Agent Paradigm: Applied Techniques.
Title (es_ES):	Técnicas para despliegue de arquitectura distribuida en sistemas expertos basados en reglas empleando el paradigma multiagente.
PhD Program:	Intelligent Systems at Engineering Science (Cod. 212) Department of Electric Engineering, Systems and Automatic
Ph.D. Student:	Francisco José Aguayo Canela
Thesis Director:	Ph.D. Isaías García Rodríguez

School of Industrial Engineering and Information Technology
University of León (Spain)

ÍNDICE GENERAL

ÍNDICE GENERAL i
ÍNDICE DE FIGURAS v
ÍNDICE DE TABLAS vii
ÍNDICE DE SCRIPTS viii
ACRÓNIMOS ix
Introducción 1

1 Justificación, objetivos y estructura
1.1 Justificación 5
1.2 Objetivos de la investigación 6
1.3 Comprobación de resultados 8
1.4 Organización capitular 9

2 Sistemas expertos y sistemas DPS
2.1 Introducción 11
2.2 Sistemas expertos 12
2.2.1 Clasificación de los sistemas expertos 13
2.2.2 Representación de datos y resolución de problemas 14
2.2.3 Sistemas de producción 16
2.2.4 Encadenamiento hacia delante 18
2.2.5 El conjunto-conflicto 20
2.2.6 Sistemas de producción distribuidos 20
2.3 Sistemas DPS 21
2.3.1 Software-colaborativo 24
2.3.2 Sistemas de pizarra 25
2.4 Sistemas multiagente 27
2.5 Resumen 30

3 La plataforma de agentes
3.1 Introducción 31
3.1.1 Fundación para la estandarización de *agentes inteligentes* 32
3.2 Agentes software 33
3.2.1 Modelo de referencia para la gestión del agente 35
3.2.2 Lenguaje de comunicación de Agentes 37
3.2.3 Estructura del mensaje según FIPA 38
3.2.4 Librería de protocolos de interacción de FIPA 43

3.2.5 Protocolos de Interacción 44
3.3 Sistemas multiagente con JADE....44
3.3.1 Arquitectura del sistema de ejecución....46
3.3.2 Una plataforma distribuida....47
3.3.3 Subsistemas de envío de mensajes....48
3.4 Resumen....49

4 Gestión de la concurrencia y de los comportamientos 51
4.1 Introducción....51
4.2 Sistema JADE en tiempo de ejecución....51
4.2.1 Plataforma distribuida de agentes....52
4.2.2 Transiciones de estado en el ciclo de vida del *agente*....53
4.2.3 Hilo de ejecución de un agente sobre JADE....53
4.2.4 Cola privada de mensajes del agente....55
4.2.5 Ciclos de vida de los Threads en *Java*....56
4.3 Gestión de la concurrencia en el *agente*....57
4.3.1 Comportamientos básicos....58
4.3.2 Operaciones programadas....59
4.3.3 Comportamientos compuestos....59
4.3.4 Comportamientos especiales....60
4.4 Resumen....61

5 Resultados de la investigación
5.1 Introducción....63
5.2 Modelo de integración de *motor de inferencia* en *agente*....66
5.3 Diagrama: 'Modelo del dominio'....66
5.3.1 Entidad: Constructor de Directorios de aplicación....68
5.3.2 Entidad: Framework o el Directorio de aplicación....68
5.3.3 Entidad: Motores basados en reglas....69
5.3.4 Entidad: Plataforma Multiagente....69
5.3.5 Entidad: Agente—Nodo instancia del sistema de producción....69
5.3.6 Entidad: Herramientas y utilidades....69
5.3.7 Entidad: Entorno Tecnológico de software....69
5.4 Relación de *utilidades-software* aportadas....70
5.4.1 Utilidad: framework o [dps-frameworkBuilder.jar]....70
5.4.2 Utilidad: psnode (Production System Node)....71
5.4.3 Utilidad: launcher (Lanzadera genérica)....72
5.4.4 Utilidad: shell (Entornos de ejecución de comandos)....74
5.5 Relación de *productos-instrumentales* aportados....75
5.5.1 Instrumento: Ontología *Problem-Solvers*....75
5.5.1.1 Esquema de la ontología Problem-Solvers....76
5.5.1.2 Concepto 'Dewey' en la ontología....77
5.5.1.3 Concepto 'Phase' en la ontología....78
5.5.1.4 Concepto 'Ticket' en la ontología....78
5.5.1.5 Predicado 'ProblemFor' en la ontología....78

5.5.1.6 Acción 'Advertise' en la ontología....79
5.5.1.7 Acción 'Repl' en la ontología....79
5.5.1.8 Acción 'Process' en la ontología....80
5.5.2 Instrumento: Entorno de desarrollo (agente `pocket`)....82
5.5.2.1 Pantalla: Shell JADE....83
5.5.2.2 Pantalla: Editor de comportamientos y configuraciones....84
5.5.2.3 Pantalla: Editor y Traza de mensajes ACL....84
5.5.2.4 Pantalla: Shell Asíncrona....85
5.5.2.5 Pantalla: Editor de archivos del sistema de producción-local....86
5.5.2.6 Pantalla: Shell Síncrona....87
5.5.3 Instrumento: *Protocolos de interacción*....88
5.5.3.1 Protocolo de Interacción Interna de doble secuencia....89
5.5.3.2 Protocolo: FIPA-Request aplicado al Servicio de Tickets....91
5.5.4 Instrumento: Documentación en formato JavaDoc....93
5.6 Relación de *técnicas* aportadas....94
5.6.1 Técnica: Integración de un *Intérprete de Java*....94
5.6.2 Técnica: Externalización de los comportamientos....95
5.6.3 Técnica: Control de inicialización de los comportamientos....96
5.6.4 Técnica: Elaboración de *pruebas-unitarias*....98
5.6.5 Técnica: Integración del motor de inferencia en el agente....98
5.6.6 Técnica: Migración de '*agentes-nodo*'....98
5.7 Relación de *propuestas* y clases de agentes aportadas....100
5.7.1 Propuesta: Directorio de aplicación....100
5.7.2 Propuesta: Convenios en el Directorio de aplicación....102
5.7.3 Propuesta: Agente-Nodo....103
5.7.3.1 Agente-Nodo: Distribución de componentes....103
5.7.3.2 Agente-Nodo: capa de propiedades 'Node-Layer'....104
5.7.3.3 Agente-Nodo: Interfaz Gráfica de Usuario....104
5.7.4 Propuesta: Agente-TicketBoard....105
5.7.4.1 Agente-TicketBoard: Distribución de componentes....105
5.7.4.2 Agente-TicketBoard: Interfaz Gráfica de Usuario....106
5.8 Síntesis de los resultados: factores críticos de éxito....107
5.8.1 Comunicación agente-motor....107
5.8.2 La estructura Nodo....109
5.8.3 La estructura Framework....110
5.8.4 La externalización de comportamientos....110
5.8.5 Un Entorno de Desarrollo para los agentes....111
5.9 Comparativa con modelos de integración....111
5.9.1 Integración: JADE - Jess....111
5.9.2 Integración: JADE - CLIPS....113
5.9.3 Integración: JADE - Jess - Protégé....114
5.10 Resumen....115

6 Pruebas de concepto

6.1 Introducción....117
6.2 Configuración en *Modo-infraestructura*....118
6.3 Prueba de concepto: Puzzles de tipo Sudoku....121

6.3.1 Introducción: una primera aproximación desde la *Shell* 122
6.3.2 Creación de nuevos *nodos* para los agentes 122
6.3.3 Resumen de los procesos realizados..... 124
6.3.4 Simplificación del directorio: /engine/ 125
6.3.5 Incorporar capacidades de inferencia a los agentes..... 125
6.3.6 Comprobación de acceso remoto al agente [Robot] 126
6.3.7 Crear un *nuevo mensaje* para remitir crucigramas..... 126
6.3.8 *Protocolo de interacción* entre [AlumnoN] vs. [Robot]..... 128
6.3.9 Cómo cambiar la tecnología de motor de inferencia..... 129
6.3.10 Resumen de la integración de resolución de Sudokus 130
6.4 Prueba de concepto: Sistema O-DEVICE 130
6.4.1 Requisitos para desplegar O-DEVICE en: *java shell clips*..... 132
6.4.2 Creación de nuevos *nodos* para los agentes ODEVnnn..... 133
6.4.3 Asignación dinámica del directorio: ?*odevice-folder* 133
6.4.4 Resumen y evaluación de procesos realizados..... 135
6.4.5 Gestión de *WM + KB* mediante comportamientos..... 136
6.4.6 Recarga de las ontologías OWL/RDF en el agente..... 139
6.4.7 Resumen de la integración de O-DEVICE..... 140
6.5 Conclusiones preliminares 141

7 Conclusiones
7.1 Conclusiones finales 143
7.2 Relación de futuros trabajos de investigación 144

Bibliografía 145

Índice alfabético 151

Anexo A: prontuarios 159
A.1 Nomenclatura 159
A.2 Hoja de Referencia..... 166

ÍNDICE DE FIGURAS

Figura 2.1: Componentes e interfaces en un sistema experto [Merritt, 2012] 12
Figura 2.2: Regla de encadenamiento hacia delante en OPS5 [Forgy, 1981] 17
Figura 2.3: Esquema simplificado de un sistema de producción 18
Figura 2.4: Motor de un sistema de producción [Friedman-Hill, 2003] 19
Figura 2.5: Áreas de investigación en software-cooperativo [Corkill, 2003] 25
Figura 2.6: Conexión de módulos en arquitectura de pizarra [Corkill, 2003] 26
Figura 2.7: Componentes de la pizarra [Corkill, 2003] 27
Figura 2.8: Sistemas colaborativos formados por agentes [Corkill, 2003] 29
Figura 3.1: Ciclo de vida del agente conforme con FIPA. [FIPA00023] 33
Figura 3.2: Modelo de referencia de la gestión del agente. [FIPA00023] 36
Figura 3.3: Visión de la Plataforma de Agentes FIPA [Greenwood, 2004] 37
Figura 3.4: Proceso de transformación de un Mensaje [FIPA00001] 38
Figura 3.5: FIPA estructura del mensaje [Bellifemine et al., 2001] 40
Figura 3.6: Modelo de referencia de transporte de mensajes [FIPA00067] 41
Figura 3.7: Implementación JADE del modelo de referencia FIPA 45
Figura 3.8: Arquitectura modular de la plataforma JADE 47
Figura 3.9: Relación entre elementos arquitectónicos [Bellifemine et al., 2010] 49
Figura 4.1: Paradigma de paso asíncrono de mensajes [Bellifemine et al., 2001] 56
Figura 4.2: Ciclo de vida de los hilos de ejecución en Java [Escoí et al., 2013] 56
Figura 4.3: Modelo jerárquico de reparto de tareas en JADE [Rimassa, 2003] 57
Figura 4.4: Jerarquía de comportamientos en JADE [Bellifemine et al., 2007] 60
Figura 5.1: Modelo del dominio de las entidades participantes 67
Figura 5.2: Ontología para comunicación entre agentes del Directorio de aplicación 77
Figura 5.3: Pantalla de Shell – JADE en el Entorno de desarrollo 83
Figura 5.4: Pantalla de Editor de comportamientos y configuraciones 84
Figura 5.5: Pantalla de Gestión y Traza de mensajes ACL 85
Figura 5.6: Pantalla de Shell Asíncrona 86
Figura 5.7: Pantalla de gestión de archivos del sistema experto local 87

Figura 5.8: Primer boceto de la interfaz gráfica de agente con motor integrado 88
Figura 5.9: Pantalla de Shell síncrona a motor CLIPS 89
Figura 5.10: Diagrama protocolo de interacción PII2S 90
Figura 5.11: Diagrama de protocolo de interación con agentes TicketBoard 92
Figura 5.12: Portada inicial de la documentación incorporada al Framework 93
Figura 5.13: Prompt de consola empleados en Shell-JADE por agentes 95
Figura 5.14: Detalle del nivel de ejecución en barras; de botones y de estado 97
Figura 5.15: PsNodeAgent, esquema general de distribución de componentes 103
Figura 5.16: Interfaz gráfica de usuario de agentes [PsNodeAgent] 104
Figura 5.17: PsBoardAgent, esquema de distribución de componentes 105
Figura 5.18: Interfaz gráfica de usuario de agentes [PsBoardAgent] 106
Figura 5.19: Pestaña de edición de archivos de agentes [PsBoardAgent] 106
Figura 5.20: Shell-JADE en agentes [PsBoardAgent] 107
Figura 5.21: Comunicación agente-motor mediante Protocolo de Interacción 108
Figura 5.22: Entorno de desarrollo, Shell Asíncrona de un agente sobre sí mismo 109
Figura 5.23: Evolución propuesta para integración JADE-Jess-Protégé 115
Figura 6.1: Arquitectura de O-DEVICE [Meditskos & Bassiliades, 2008] 131

ÍNDICE DE TABLAS

Tabla 1.1: Objetivos 1º, 2º y 4º clasificados por campo temático y cuestiones a resolver ... 7
Tabla 2.1: Niveles y modelos de la Representación de datos ... 15
Tabla 2.2: Matriz para distinguir Líneas de Investigación [Durfee & Rosenschein] ... 23
Tabla 3.1: Frame y Ontology empleados para definir Mensajes ACL [FIPA00061] ... 39
Tabla 3.2: Parámetros en mensaje ACL categorizados [FIPA00061] ... 42
Tabla 3.3: Campos en el Envelope de un mensaje ACL [FIPA00067] ... 43
Tabla 4.1: Privilegios necesarios para modificar el estado de agentes en JADE ... 53
Tabla 5.1: Distribución de los Productos aportados por Llínea de trabajo ... 65
Tabla 5.2: Términos asignados a entidades en el Modelo del Dominio ... 67
Tabla 5.3: Descripción del Entorno Tecnológico: librerias y versiones empleadas ... 70
Tabla 5.4: Códigos empleados en la definición de Acciones Process ... 81
Tabla 5.5: Nomeclatura empleada en archivos de carga de comportamientos ... 96
Tabla 5.6: Nomeclatura y téminos propuestos en el Directorio de apliación ... 103
Tabla 5.7: Comparación gráfica de *modelos de integración* JADE-Jess ... 113
Tabla 5.8: Comparación gráfica de *modelos de integración* JADE-CLIPS ... 113
Tabla 5.9: Comparación gráfica de *modelos de integración* JADE-Jess-Protégé ... 114
Tabla 6.1: Estructura por defecto de un Directorio de aplicación ... 119
Tabla 6.2: Función de las librerías incluidas en el directorio ./lib/ ... 121
Tabla 6.3: Comportamientos adicionales en agentes con O-DEVICE integrado ... 139
Tabla 6.4: Comportamientos activados en agentes con O-DEVICE integrado ... 141

ÍNDICE DE SCRIPTS

Código 4.1: Instancia de Thread según [Lea, 1999]....53
Código 4.2: Inicialización de Threads en JADE [CSELT & TILab, 2014]....55
Código 5.1: Creación del Directorio de aplicación con dps-frameworkBuilder....71
Código 5.2: Opciones de la utilidad de construcción de Directorios de aplicación....71
Código 5.3: Utilidad 'psnode' para construcción de la estructura de un nodo....72
Código 5.4: Ejemplo de ejecución de la utilidad 'psnode'....72
Código 5.5: Despliegue de agente en plataforma JADE con utilidad launcher....72
Código 5.6: Utilidad 'launcher' para lanzamiento de agentes (ayuda)....73
Código 5.7: Alias utlizados habitualmente con la utilidad 'launcher'....74
Código 5.8: Utilidad 'shell' para acceder a intérpretes del Framework....74
Código 5.9: Métodos de cierre de Shell: CLIPS, Jess, Java y Hsql....75
Código 5.10: Ejemplo de Mensaje ACL con objetos de ontología Problem-Solvers....76
Código 5.11: Comprobación de ontología Problem-Solvers desde consola....77
Código 5.12: Ejemplo de mensaje ACL con predicado ProblemFor....79
Código 5.13: Fragmento de mensaje ACL con acción Repl....80
Código 5.14: Volcado de un mensaje ACL con acción Process en su contenido....80
Código 5.15: Técnica empleada para conectar/desconectar la Shell Síncrona....88
Código 5.16: Cargador de comportamiento externalizado en tiempo de ejecución....96
Código 5.17: Propiedades por defecto de un Directorio de aplicación (vista parcial)....101
Código 5.18: Archivo board.properties de un agente PsBoardAgent (vista parcial)....105
Código 5.19: Integración JADE-Jess de [Kamyab & Bellifemine, 2004] (parcial)....111
Código 5.20: Integración JADE-Jess de [Cardoso, 2007] (parcial)....112
Código 5.21: Nuevo modelo para la integración JADE-Jess-Protégé....114

ACRÓNIMOS

ACL	Agent Communication Language
AI	Artificial Intelligence
AMS	Agent Management System
AOP	Agent Oriented Programming
AP	Agent Platform
API	Application Program Interface
BBS	Blackboard Systems
BDI	Architecture Believe Desire and Intention
C/S	Client-Server
CLIPS	C++ Language Integrated Production System
CN	Contract Net Protocol
CORBA	Common Object Request Broker Architecture
DL	Descriptive Logic
DF	Directory Facilitator (FIPA/JADE)
DAI	Distributed Artificial Intelligence
ES	Expert System. (véase RBES)
FIPA	Foundation for Intelligent Physical Agents
GPRS	General Packet radio System
HTTP	Hyper Text Transfer Protocol
HW	Hardware
IIOP	Internet Inter-ORB Protocol
IP	Internet Protocol
J2EE	*Java* 2 Enterprise Edition
J2ME	*Java* 2 Micro Edition
J2SE	*Java* 2 Standard Edition
Jess	*Java* Expert System Shell
JICP	JADE Internal Communication Protocol
JADE	*Java* Agent Development Platform
JSA	JADE Semantic Add-On
JVM	*Java* Virtual Machine
KB	Knowledge Base
LGPL	Lesser GNU Public License
M2M	Machine to Machine
MAS	Multiagent Systems
MIDP	Mobile Internet Device Profile
MTS	Message Transfer System (FIPA/JADE)

OMG	Object Management Group
ORB	Object Request Broker
OS	Operating System
OWL	Web Ontology Language
P2P	Peer to Peer
PC	Personal Computer
PDA	Personal Digital Assistant
PM	Production Memory (Rules Database)
R&D	Research and Development
RBES	Rule-Based Expert Systems
RDF	Resource Description Framework
RPC	Remote Procedure Call
RMA	Remote Management Agent (JADE)
SMTP	Simple Mail Transfer Protocol
SOAP	Simple Object Access Protocol
SSL	Secure Socket layer
TCP	Transmission Control Protocol
URI	Uniform Resource Locator
W3C	World Wide Web Consortium
W-LAN	Wireless Local Area Network
WM	Working Memory (Facts Database)
WSDL	Web Services Description Language
XML	Extensible Markup Language

Resumen

Esta tesis doctoral propone modelos y técnicas, y aporta herramientas, para facilitar la transformación ó la creación de aplicaciones basadas en *sistemas de producción*, en entornos compuestos por *agentes* que se despliegan en una plataforma multiagente.

Este objetivo principal requiere la integración de un motor de inferencia en cada uno de los agentes software que componen la aplicación, teniendo como premisa fundamental no modificar ni alterar la arquitectura intrínseca de la plataforma multiagente, ni las características que son propias de los agentes software, lo cual diferencia la presente investigación de otros trabajos similares.

Como resultado de la investigación se han obtenido una serie de modelos: integración desacoplada entre agente y motor de inferencia, comunicación entre motor de inferencia y el agente conforme a las especificaciones de la fundación FIPA (Foundation for Intelligent Physical Agents), y un mecanismo de migración de los agentes con motor de inferencia integrado.

Se han diseñado y construido una serie de elementos y funcionalidades destinadas a facilitar el proceso de despliegue de estas aplicaciones distribuidas: se ha ideado e implementado un entorno de desarrollo que extiende a cualquier agente de la aplicación con el fin de facilitar el diseño, análisis y depuración de sus comportamientos; se ha creado un mecanismo para modelar el conocimiento mediante diferentes *shells* que permiten interactuar con el código del agente y con el motor de inferencia tanto en tiempo de diseño como en tiempo de ejecución; y se han ampliado las capacidades del agente para interpretar, en tiempo de ejecución, comportamientos externalizados en archivos.

Para comprobar la validez de los modelos, elementos y funcionalidades mencionados se han realizado una serie de pruebas de concepto que permiten afirmar que el conjunto de aportaciones realizadas en esta investigación ha logrado conseguir el objetivo principal de la misma.

Escuelas de Ingenierías Industrial e Informática
Universidad de León

Abstract

This Ph.D. Thesis proposes models and techniques, and provides tools, for easing the adaptation or creation of production system applications, in environments composed of a number of software agents deployed over a multi-agent platform.

This main objective requires the integration of an inference engine in the software agents that make up the application, having as a fundamental prerequisite not to change or to alter the intrinsic architecture of the multi-agent platform, nor the defining characteristics of the software agents, what differentiates this research from other, related works.

As some results from this research, a number of models have been obtained: uncoupled integration of an agent with an inference engine, communication process among the inference engine and the agent in accordance with the specifications from FIPA (Foundation for Intelligent Physical Agents), and a mechanism for the migration of agents having an integrated inference engine.

A number of elements and functionalities have been built in order to ease the deployment process for these distributed applications: a development environment has been designed and built, extending any agent in the application with the aim to facilitate the processes for designing, analyzing and debugging of behaviours; a mechanism has been built in order to model knowledge by integrating a number of shells that allow interacting with the source code of the agent and with the inference rule engine, both at design and run times; also, agent capabilities have been extended for it to be able to incorporate, at runtime, behaviours that have been externalized into files.

In order to test the validity of the different mentioned models, elements and functionalities, a number of tests have been carried out. The results from these tests allow concluding that the set of contributions obtained from this research has achieved the initial main objective.

Escuelas de Ingenierías Industrial e Informática
Universidad de León

INTRODUCCIÓN

A principios de la década de los cincuenta, los estudios en inteligencia artificial (**AI**) se centraban en el razonamiento, en los métodos de *representación del conocimiento* y en la *resolución de problemas.* En los años setenta, ya se había alcanzado la tecnología necesaria para desarrollar *sistemas expertos* con capacidad de razonamiento y de resolución de problemas, en algunos casos, comparable a las capacidades de los especialistas humanos que participaban en sus construcciones [Garijo, 2005]. Para *Francisco Garijo*, fueron tales las expectativas creadas por aquellos resultados iniciales, que provocaron inversiones millonarias en la década de los ochenta y líderes de la industria como Japón, mantenían múltiples líneas de producción de mainframes lanzando, en 1983, la quinta generación de computadores con Lenguaje *Prolog* como elemento central empleado para desarrollar funcionalidades inteligentes basadas en procesamiento simbólico.

La búsqueda permanente para mejorar el rendimiento y la capacidad de proceso en las aplicaciones dio lugar a la creación de arquitecturas especiales para permitir a los *sistemas expertos* operar de manera distribuida. El nacimiento de los sistemas de *resolución de problemas* y la arquitectura de pizarra enunciada por el grupo de investigación de *Amherst*, en la *Universidad de Massachusets* [Lesser, 1989], finalizaron —entre otros tantos éxitos— con la construcción del sistema de reconocimiento del habla (Hearsay-II) bajo el patrocinio de DARPA que utilizaba un modelo de cooperación entre sistemas de *resolución de problemas* independientes [Erman et al., 1980]. Durante más de tres décadas los investigadores en *Inteligencia Artificial*, han empleado el paradigma de la colaboración aplicado al software para solventar problemas complejos mediante la explotación de la *arquitectura de pizarra.*

Como establecieron [Erman et al., 1980] y [Gasser & Ishida, 1991], «...era lógico que el siguiente paso en la evolución de los *sistemas expertos* fuese la aglomeración de sus capacidades en *nodos de resolución de problemas*[1] y, con la llegada de las facilidades para la interconexión entre ordenadores fuese la *arquitectura de pizarra* [Hayes-Roth, 1985] el ingrediente final de aquella receta necesaria para alcanzar la capacidad y la eficiencia en la *resolución de problemas*» dando así lugar al nacimiento de los sistemas distribuidos de resolución de problemas o, de forma abreviada, *Sistemas DPS*.

[1] ***Problem-Solvers nodes***: Nodos de resolución de problemas.

Los *sistemas expertos* continúan representando hoy por hoy, uno de los logros más notables de la *Inteligencia Artificial* y ofrecen un modelo que permite codificar el conocimiento humano en el lenguaje de los ordenadores. *Ana Mas* [Mas, 2005], indica que fue precisamente la *representación del conocimiento* el elemento clave para alcanzar los primeros razonamientos y, con el paso del tiempo y la mejora de las funcionalidades que han sufrido los *sistemas expertos* con algoritmos avanzados de optimización cada vez más complejos, se ha llegado hasta el presente y se han consolidado como las piezas clave para la obtención de deducciones a partir del conocimiento introducido en forma de *reglas*.

En el ámbito de la Inteligencia Artificial Distribuida se desarrolla el concepto *agente* que, aunque surgido en la década de los setenta, encuentra a lo largo de los años noventa su definición más completa y aceptada. *Michael Wooldridge* [Wooldridge, 2002], destaca que esta definición se construye al establecer a los agentes como: «sistemas informáticos situados en un entorno, capaces de realizar acciones de forma autónoma para conseguir sus objetivos de diseño». La herramienta para definir esas acciones y esos objetivos llevados a cabo por los agentes es la programación orientada a agentes y son las plataformas de agentes (**AP**, *Agent Platforms*) el lugar donde se despliegan y se ejecutan esas entidades de software.

A principios del siglo XXI, cuando los sistemas distribuidos de resolución de problemas ya habían evolucionado hacia el concepto de *agente*, destacaba *Durfee* [Durfee, 2001] que: «el término conocido como *sistema distribuido de resolución de problemas* se aplicaba a una rama específica de la inteligencia artificial distribuida donde el énfasis se situaba en conseguir que los *actores* (en la literatura científica actual conocidos como *agentes*) trabajasen unidos para solucionar problemas que requerían esfuerzos colectivos». Lo que vino a remarcar la separación conceptual existente entre las soluciones basadas en *arquitectura de pizarra* y la nueva aproximación de la *Inteligencia Artificial Distribuida* (DAI) que revelaba, cada vez con más claridad, las capacidades de los *Sistemas Multi-Agente*[2] [Hayes-Roth, 1985], [Corkill, 1991], [Corkill, 2003] y [Hermann et al., 2008].

En esa encrucijada de significados y de especialización de los paradigmas dedicados a la resolución de problemas, recuerdan *Corchado* y *Molina* [Corchado & Molina, 2002] que a esa construcción de significados y asignaciones de conceptos adoptados en la Comunidad de Investigadores en DAI; se unieron los esfuerzos de la fundación **FIPA**[3], la cual lideró en Europa los trabajos de *definición* y de *normalización* en el ámbito de la tecnología de agentes y de las plataformas de agentes. Hoy, el resultado de la unión de ambas líneas de trabajo suponen la clave para comprender cómo se ha alcanzado el estándar *de facto* que define a cada componente de los sistema multiagente y cómo deben dotarse de propiedades a esos nuevos sistemas multiagente para que además, sean `*interoperables*`.

---

[2] ***Sistemas Multi-Agente***: se empleará la expresión en castellano 'sistemas multiagente' actualmente reconocida como sintaxis correcta. Se ha utilizado en este caso, para hacer hincapié en la primera nomenclatura utilizada por los investigadores en DAI al referirse a estos sistemas.

[3] ***FIPA***: (c) 1996-2015 IEEE *Foundation for Intelligent Physical Agents* - http://www.fipa.org/

Los sistemas multiagente más habituales hoy en día consisten en comunidades de agentes que implementan diferentes tipos de comportamientos. Esos comportamientos involucran *acciones*, que suelen describirse por medio de código integrado en el propio agente, para establecer sus conductas desde el momento de su creación.

El uso de sistemas de producción (o *sistemas expertos basados en reglas de encadenamiento hacia delante*) para implementar las *acciones* o los razonamientos en los agentes es menos común debido, en parte, a los problemas que supone la integración de este tipo de sistemas en estas entidades de software. Las soluciones de integración que se encuentran habitualmente están orientadas a lenguajes y plataformas específicos y suelen conllevar la pérdida de algunas de las características intrínsecas exigidas por la *agencia* como por ejemplo: la movilidad, la autonomía o, el *ciclo de vida* que exponen los agentes-software sobre la plataforma.

La investigación llevada a cabo en esta tesis doctoral persigue conseguir la integración mencionada anteriormente sin perder ninguna de las características que definen a los agentes, logrando no interferir en su funcionamiento o en los procesos de comunicación entre agentes dentro de una plataforma multiagente en la que puedan encontrarse. La plataforma multiagente elegida para realizar las comprobaciones y verificaciones sobre los diferentes resultados obtenidos es JADE (*Java Agent Development Environment*) [CSELT & TILab, 2017] debido a su estabilidad, funcionamiento y amplia comunidad de usuarios, así como a su conformidad con las especificaciones FIPA.

1 Justificación, objetivos y estructura

Para mí, el diseño es el alma fundamental de una creación humana y termina manifestándose en las capas externas de esa creación.

Steve Jobs

Objetivos del capítulo:

- *Enumerar qué motivos llevaron a justificar la elaboración de la tesis.*
- *Presentar los objetivos perseguidos.*
- *Mostrar la estructura capitular de la tesis.*

1.1 Justificación

El uso de sistemas de producción para resolver problemas complejos, con información incompleta, o que involucran conocimiento heurístico, tiene sus raíces en los años cincuenta y ha llegado hasta el presente. De hecho, este tipo de sistemas encuentran cada vez más campos de aplicación práctica y exigen plataformas y modelos adecuados para desplegarse y adaptarse a los requisitos y necesidades de software; con capacidad de inferencia, potencia de análisis y escalabilidad adecuadas. Estas premisas llevan a plantear la posibilidad de utilizar las plataformas de sistemas multiagente para conseguir el despliegue de aplicaciones, basadas en *sistemas de producción*, de manera distribuida.

Existen trabajos previos que se han dedicado a conseguir un objetivo similar al presentado anteriormente, aunque se trata de soluciones parciales que no consiguen cumplir totalmente con dos premisas que son irrenunciables para esta investigación:

(A) Mantener el compromiso de no-modificación y no-alteración en forma alguna, de la arquitectura intrínseca aportada por la plataforma multiagente, para el despliegue de agentes, utilizada en la investigación.

(B) Mantener el compromiso de no-modificación y no-alteración en forma alguna, de las capacidades y las características que son propias de los agentes-software, i.e.: sus propiedades de autonomía, sociabilidad, reactividad y proactividad asociada a la culminación de objetivos potenciando la característica de movilidad hacia los contenedores remotos gestionados por la plataforma multiagente seleccionada.

1.2 Objetivos de la investigación

Esta investigación tiene como objetivos el estudio, diseño y creación de arquitecturas, modelos y técnicas para facilitar la incorporación de los avances de la *tecnología multiagente* en las aplicaciones basadas en *sistemas de producción*. El resultado de esta investigación debe especificar con precisión cómo integrar un *motor de inferencia* de manera estable en las entidades de software conocidas como *agentes* que, a su vez, puedan funcionar de forma colaborativa dentro de *plataformas multiagente*, siguiendo las especificaciones y estándares de facto en este dominio. De esta forma, se podrán abordar proyectos para integrar sistemas de producción heredados o bien crear aplicaciones nuevas de acuerdo a este paradigma distribuido.

Se realiza una división en cuatro áreas de los objetivos mencionados anteriormente:

(**1º**) *Estudiar* las tecnologías, arquitecturas y modelos de implementación empleados en el ámbito de la programación orientada a agentes, los sistemas multiagente y las especificaciones y estándares existentes, en particular las generadas por la fundación FIPA.

(**2º**) *Analizar la viabilidad de integración* en un agente software, de un motor de reglas perteneciente a algunas de las tecnologías más comunes y extendidas de construcción de sistemas de producción, como pueden ser CLIPS o *Jess*, estudiando los trabajos previos que se han efectuado con este mismo fin.

(**3º**) *Diseñar un modelo de integración* genérico que permita a un agente software gestionar un motor de inferencia sin alterar ninguna de sus propiedades intrínsecas de agente.

(**4º**) *Aportar artefactos* para corroborar las hipótesis y resultados obtenidos mediante demostraciones de funcionamiento.

De manera más detallada, los objetivos citados involucran (véase Tabla 1.1) a los siguientes campos de conocimiento:

- La *concurrencia en Lenguaje Java*, como área de conocimiento donde encontrar soluciones a la gestión del Hilo de ejecución del agente [Garg, 2005].
- Las *comunicaciones FIPA-ACL entre agentes JADE*, como área de conocimiento para solventar la comunicación entre agente y el *motor de inferencia* local y entre los agentes y los motores de inferencia de agentes remotos.
- Las *especificaciones FIPA*, como área de conocimiento para no apartar los agentes creados, del modelo establecido por FIPA.
- Las tecnologías de *Sistemas Expertos* y Guías de usuario y de programación avanzadas de CLIPS y *Jess*, como área de conocimiento para analizar la posibilidad de crear un módulo de gestión de los agentes de una misma *aplicación*.

- La tecnología de los *Sistemas DPS* y en concreto de la *Arquitectura de pizarra* y su aplicación en los *sistemas cooperativos distribuidos* mediante nodos de resolución de problemas, como área de conocimiento para la búsqueda de posibles adaptaciones de la arquitectura que facilite la transformación de las aplicaciones.

- La *plataforma JADE*, sus paquetes de clases y en concreto `jade.core.Agent`, el *ciclo de vida* del agente y el tratamiento que establece JADE para los hilos de ejecución de la plataforma, como área de conocimiento para el desarrollo de artefactos.

Objetivos de la tesis

Objetivo	*Campo / Disciplina*	*Cuestiones a resolver*
(1º) Estudiar	Concurrencia en *Java* Comunicaciones en JADE	Hilo de ejecución del agente Hilo de ejecución motor de inferencia
	Especificaciones FIPA Sistemas Expertos Sistemas DPS Plataforma JADE	Intercambio de hechos y reglas Programación del módulo de control Arquitectura final aplicada Prototipo de Agente reutilizable
(2º) Analizar viabilidad	Frameworks Agentes cognitivos[4]	Persistencia del Agente y el KB Movilidad en agentes-cognitivos
	CLIPS/Jess en JADE	Integración de Librería CLIPSJNI
	Sociedades de agentes	Marco de operatividad entre agentes
	Algoritmos en DS[5]	Sincronización / Recuperación
(4º) Aportar artefactos	Ejecutables sobre los sistemas operativos	Familia Windows 32 y 64bits Macintosh OS-X 10.6 y superior GNU-Linux 32 y 64bits
	Empleando la Programación orientada a objetos clásica (OO)	Formato de persistencia Técnica de empaquetado / envío. Procedimientos de shutdown / init
	Empleando la Programación orientada a agentes (AOP).	GUI de agente (multilingüe) Arranques desde jade.Boot Interpretación de OWL Publicación de servicios en DF

Tabla 1.1: Objetivos 1º, 2º y 4º clasificados por campo temático y cuestiones a resolver

- La selección de un *frameworks* de desarrollo y despliegue de sistemas multiagentes del mercado, y su utilización para elaborar los artefactos software y las soluciones propuestas que además, permita realizar el desarrollo en Lenguaje *Java*.

- Los *agentes cognitivos* y las plataformas con agentes con capacidades de inferencia ya existentes en la actualidad, como área de estudio preliminar en la investigación, que debe ser acometida antes del diseño de nuevas propuestas.

[4] ***Agentes cognitivos***: Agentes con capacidades de inferencia obtenidas de la utilización de tecnología de *sistemas expertos*.
[5] **DS: *Distributed System*,** (Sistema Distribuido).

- La viabilidad de integración de la pasarela nativa de *Java* de CLIPS denominada *CLIPSJNI* en agentes-software, como posible solución para la interacción desde la *Shell* de CLIPS en la interfaz gráfica de agentes JADE.

- La viabilidad de integración de *Jess* en agentes-software, como posible solución para la interacción desde la *Shell* de *Jess* en la interfaz gráfica de agentes JADE.

- El análisis de las *Sociedades de agentes* como solución tecnológica para formar un perímetro de *aplicación* cerrado; un espacio lógico donde los agentes de la aplicación se intercambian *hechos*, *reglas* y *resultados*.

- La comprobación de las *pruebas de concepto* sobre los *sistemas operativos* habitualmente utilizados: Microsoft Windows 7, GNU-Linux y Apple OS-X 10.x.

- La *programación orientada a objetos clásica (OO)* y la programación de comportamientos de los agentes-software, como instrumentos para desarrollar y obtener capacidades de persistencia de la *base de conocimiento* de los agentes.

- La *programación orientada a agentes (AOP)*, como instrumento necesario para lograr la construcción de agentes JADE con versatilidad suficiente para su transformación en módulos de la aplicación desplegada en el escenario multiagente.

1.3 Comprobación de resultados

Para evaluar los resultados obtenidos en esta investigación se procederá con la implementación de un conjunto de artefactos software (utilidades y agentes software) con ejemplos completos para demostrar la viabilidad de utilización de las técnicas de integración de motores de inferencia en los *agentes-software.* Ese material contendrá al menos los siguientes artefactos:

(**a**) Una utilidad de software para generar, en función del sistema operativo donde se ejecute, el escenario de despliegue de la aplicación una vez transformada al paradigma multiagente.

(**b**) Una utilidad de software para crear nuevos *agentes con motor de inferencia integrado*.

(**c**) Una utilidad software que permita utilizar el directorio de servicios de la plataforma multiagente.

(**d**) Una utilidad software para facilitar el desarrollo y el análisis de los agentes creados, e ideada para ser utilizada por el *Ingeniero de conocimiento* o por el *Experto del dominio*.

1.4 Organización capitular

El resto de la tesis está organizado en 6 capítulos. Los tres primeros están dedicados a ofrecer un *'estado de la cuestión'* y los tres siguientes exponen los resultados de la investigación, las pruebas de concepto e integración y las conclusiones finales. A continuación se detalla cada uno de ellos.

El capítulo 2 trata sobre el funcionamiento interno de los *sistemas de producción* y *los sistemas distribuidos de resolución de problemas* así como de las implicaciones que su arquitectura provocan sobre el diseño, las técnicas y las herramientas aportadas como resultado de la investigación.

El capítulo 3 está dedicado al estudio de la plataforma multiagente JADE y a las especificaciones de la fundación FIPA: ambas trascendentes a la hora de mantener los compromisos de no-modificación y no-alteración mencionados en el apartado de justificación.

El capítulo 4 expone una síntesis del funcionamiento y la implementación que adopta JADE en relación a problemas de concurrencia que han influenciado de una manera definitiva tanto el diseño como la implementación posterior de los comportamientos de los agentes.

El capítulo 5 expone con detalle el modelo de integración diseñado. Se comienza el capítulo revisando y analizando anteriores trabajos de integración que perseguían objetivos similares. A continuación se presenta la solución aportada en esta tesis, describiendo el diseño y funcionamiento de las diferentes utilidades, instrumentos, técnicas y propuestas planteadas.

El capítulo 6 está dedicado a exponer dos *pruebas de concepto* que permiten comprobar la validez de los resultados obtenidos, así como la potencialidad de los mismos en el ámbito académico e industrial.

El capítulo 7 contiene las conclusiones de la investigación y la relación de posibles futuros trabajos.

2 Sistemas expertos y sistemas DPS

Se propone la auto-organización
como una de las soluciones posibles para que los sistemas
se gobiernen a sí mismos y puedan adaptarse así, a los cambios.

Toru Ishida

International Joint Conference on Web Intelligence
and Intelligent Agent Technologies, 2013

Objetivos del capítulo:

- *Resaltar propiedades fundamentales de los sistemas de producción.*
- *Exponer las características de los sistemas distribuidos de resolución de problemas, también conocidos como sistemas DPS.*

2.1 Introducción

Este capítulo estudia los conceptos de sistemas expertos (apartado 2.2), sistemas DPS (apartado 2.3) y ofrece una introducción a los sistemas multiagente (apartado 2.4), intentando obtener una aproximación a las diferentes formas y arquitecturas en las que los sistemas de producción se han utilizado a la hora de resolver problemas desde sus orígenes hasta su empleo en escenarios distribuidos.

Los conceptos básicos sobre *sistemas expertos* han sido extraídos de publicaciones de los autores; *Feigenbaum, Giarratano*, *Ishida*, *Gasser, Merritt* y *Friedman-Hill* entre otros. Los *sistemas DPS* (también denominados habitualmente *sistemas de pizarra*) y los *sistemas multiagente* (entendidos como una forma de implementar resolución de problemas distribuida) se han estudiado utilizando los trabajos de: *Corkill*, *Durfee* y *Rosenschein*, autores que muestran con gran claridad la división que existe entre los objetivos perseguidos por ambos sistemas.

Durfee y *Rosenschein* [Durfee & Rosenschein, 1994] coinciden en realizar dicha separación entre los conceptos generados por la tecnología de *sistemas DPS* de aquellos conceptos que son propios de los *sistemas multiagente*. Esos autores muestran en sus publicaciones la vigencia de los *sistemas DPS* y la *arquitectura de pizarra* y, llaman la atención sobre cómo debe ser empleada esa separación conceptual durante las fases de *diseño* e *implementación* de prototipos en las investigaciones que empleen ambas tecnologías.

2.2 Sistemas expertos

Edward Feigenbaum en [Buchanan & Feigenbaum, 1978, Cohen & Feigenbaum, 2014, Lenat & Feigenbaum, 1991], define a los sistemas expertos como «...un programa de computación inteligente que usa el conocimiento y los procedimientos de inferencia para resolver problemas, lo suficientemente complejos, como para requerir una significativa cantidad de experiencia humana para alcanzar su solución».

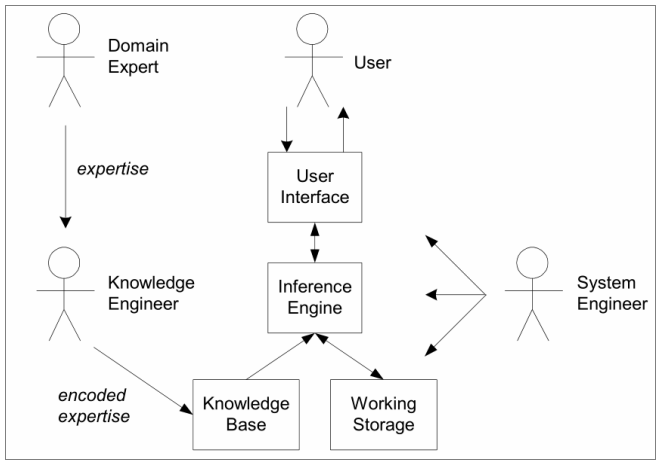


Figura 2.1: Componentes e interfaces en un sistema experto [Merritt, 2012]

Joseph C. Giarratano [Giarratano, 2014] establece que el primer paso para resolver cualquier problema es definir el área o dominio del problema que va ha ser resuelto. El dominio del problema es el área de conocimiento acotada y específica en la que un especialista en la materia puede resolver problemas con facilidad. Los sistemas expertos, al igual que los especialistas humanos, se orientan a un dominio de conocimiento concreto para solventar problemas específicos.

La visión de un Sistema Experto (ES, *Expert System*) es funcional y orientada a dar respuestas tras la recepción de los estímulos de entrada cuando son recibidos. Los sistemas expertos quedan incluidos dentro de los sistemas basados en conocimiento (KBS, *Knowledge Base Systems*) que conforman un marco más extenso y aportan una visión estructurada, orientada a conocer y a comprender cuáles son los orígenes del problema de forma global. Estos grandes sistemas involucran tanto a usuarios, ingenieros de conocimiento, expertos del dominio, como al propio conjunto de componentes de un sistema experto; las interfaces de usuario, generadores de explicaciones, el subsistema de adquisición de conocimiento, etc.

Dennis Merritt define los sistemas expertos en la publicación *on-line* del libro *'Building expert system with Prolog'* [Merritt, 2012], donde establece que son: «...aplicaciones de ordenador que encierran experiencias no algorítmicas para resolver ciertos tipos de problemas». Y concluye al afirmar que los servicios ofrecidos por esas aplicaciones de ordenador, como por ejemplo jugar al ajedrez, planificar decisiones financieras o auditar sistemas en tiempo real, requirieron originalmente para su resolución del empleo de la experiencia humana. Los sistemas expertos para *Merritt*, contienen un conjunto de subsistemas principales y de interfaces de conexión con los individuos que juegan los distintos roles frente a esos sistemas

(Figura 2.1, pág. 12). Para *Giarratano,* [Giarratano, 1997], *Ishida* [Ishida, 1994] y *Merritt*, los componentes esenciales de un sistema experto son:

- ***Base de conocimientos*** (KB***,*** *Knowledge Base*) donde se recoge la experiencia en un formato declarativo. A menudo utilizando *reglas* del tipo IF THEN. Originalmente a las sentencias declarativas del conocimiento en este formato se les llamaba producciones. De ahí deriva que cuando la base de conocimientos solamente está compuesta de condiciones de producción se denomine (PM, *Production Memory*) concepto asociados en ocasiones al término *Knowledge Base*. Ambos términos son extensamente utilizados por *Ishida* en [Ishida, 1995], en [Kersten & Kak, 1993], [Luger & Chakrabarti, 2011], [Peers, 1998], [Perraju & Prasad, 2000], [Rahimi et al., 2007] y [Rubin et al., 1988] entre otros.
- ***Memoria de trabajo*** (WM, *Working Memory*), son los datos específicos para que un problema pueda ser resuelto. También conocida como memoria de *hechos*.
- ***Motor de inferencia*** (IE, *Inference Engine*) es el núcleo funcional del sistema experto, capaz de derivar recomendaciones a partir de las *reglas* de la base de conocimientos y de los *hechos* en la memoria de trabajo.
- ***Interfaz de usuario*** (GUI, *Graphics User Interface*) subsistema que permite y gestiona el diálogo entre usuario y sistema.

Prosiguen los autores arriba citados con la descripción de los roles principales desempeñados por las personas en su interacción con estos sistemas:

- ***Experto del dominio:*** el individuo o individuos, quienes son expertos en la resolución de problemas cuyo campo de conocimiento coincide con el dominio del sistema experto que desea desarrollarse.
- ***Ingeniero de conocimiento:*** la persona con los conocimientos y la habilidad suficientes para transcribir en código el conocimiento del experto con un formato declarativo propio de los sistemas expertos.
- ***Usuario final:*** la persona que utiliza el sistema experto y que es informada de manera similar a como lo habría hecho un *experto del dominio* a tenor de los datos introducidos tras entregar un razonamiento en base a los *hechos*.

En la actualidad, sigue siendo materia de debate decidir cuál es la mejor forma de construir y desarrollar un sistema experto. Por un lado están los defensores de utilizar una *shell*, porque ese subsistema permite expresar en Lenguaje de alto-nivel las reglas y funciones sobre el dominio de conocimiento específico. Por otro, aquellos que argumentan que un sistema experto debe ser implementado en Lenguaje Prolog, LISP, o en lenguajes convencionales como C++ o *Java*, dejando inmerso dentro del código el conocimiento del *experto del dominio*.

2.2.1 Clasificación de los sistemas expertos

Los sistemas expertos se organizan en función de las propiedades y de los componentes más notables que poseen, como la *shell*, que es un subsistema habitualmente empleado como

elemento diferenciador en los sistemas expertos. Si se emplea como criterio de organización al *modo de razonamiento* que utiliza el sistema experto, resulta la siguiente clasificación:

- ***Sistemas expertos de razonamiento basado en objetivos*** (*Goal-Driven Reasoning*). También conocidos como sistemas expertos de encadenamiento inverso. En estos sistemas el motor de inferencia es capaz de fragmentar un objetivo en pequeños subobjetivos que son ensayados de manera más sencilla. [Merritt, 2012]. Con encadenamiento inverso es posible solventar problemas de manera eficiente cuando las soluciones pueden ser estructuradas como una colección de posibilidades. Esto es, cuando puede seleccionarse una solución como 'la mejor' entre un conjunto de opciones. Pertenecen a este tipo de problemas los de identificación y los de diagnóstico. En el segundo grupo el sistema debe aportar además, un módulo de justificación de los resultados o, ser capaz de mostrar cuál ha sido el camino seguido para llegar a dicha conclusión con indicadores porcentuales de las soluciones que quedaron en segundo y tercer lugar.

- ***Sistemas expertos de razonamiento borroso*** (*Uncertainty Reasoning*). Son sistemas expertos que han sido preparados para tratar con *reglas* y *hechos* que no aportan datos precisos o que son incompletos. Existen variantes en función del grado de imprecisión o de carencia que son capaces de soportar este tipo de sistemas expertos.

- ***Sistemas expertos de razonamiento dirigido por datos*** (*Data-Driven Reasoning*). Conocidos como sistemas expertos de encadenamiento hacia delante o *sistemas de producción*. En estos sistemas el *motor de inferencia* encuentra las soluciones concatenando y acumulando en la *memoria de trabajo* las conclusiones progresivas a partir de un *hecho* inicial que utiliza para deducir y alcanzar la solución final.

  En numerosas situaciones, no es posible enumerar todas las posibles soluciones que el sistema tendría que analizar si tratase de solventar el problema con el enfoque dirigido por objetivos. El caso típico de aplicación se produce en problemas de posibles ubicaciones de elementos en un espacio finito donde, las posibilidades son realmente elevadas y el encadenamiento por objetivos no sería aplicable. En estos casos, el sistema experto mantiene una 'fotografía' del estado del problema y utiliza esa imagen como el *estado actual* de la solución, el cual es modificado en cuanto existen *reglas* sobre la *memoria de trabajo* que satisfacen dicho estado particular del problema y es entonces cuando el motor de inferencia interviene y hace avanzar el problema hacia un nuevo estado.

2.2.2 Representación de datos y resolución de problemas

Se sintetizan a continuación dos conceptos trascendentes: el primero es el concepto de *representación de datos*, al que le sigue el concepto de *resolución de problemas*.

Representación de datos: En un problema determinado, los *datos* pueden adoptar uno, dos o varios simultáneamente, de los cinco modelos de representación de datos que son mostrados en la Tabla 2.1. La representación más simple para expresar un dato es el (**1**) binomio *atributo-valor*. Por ejemplo [color-blanco] o [tamaño-grande]. Sin embargo, cuando un sistema requiere ejecutar un proceso de razonamiento que involucre a múltiples objetos requiere de un

modelo que adhiera el objeto a la pareja [*atributo-valor*] formando (**2**) *tripletes*. Ese modelo evoluciona con rapidez cuando existen múltiples objetos en el sistema con similares propiedades. Para ese tercer caso, la forma de almacenamiento y de representación más adecuada de los datos es la (**3**) *tabla de registros*, donde el existe un campo de clave única que define a cada registro y que almacena al objeto donde a su vez se localizan las propiedades.

Niveles y modelos de representación de datos

Nivel:	***Modelo:***	***Ejemplos:***
Parejas:	***Atributo-Valor***	• color — blanco
Tripletes:	***Objeto Atributo-Valor***	• silla-con-brazos — anchura — 30 • silla-simple — anchura — 20
Registros:	***Tablas en bases de datos***	***Objeto*** | ***anchura*** | ***color*** | ***tipo*** silla@1 | 3 | naranja | informal armario@1 | 2 | marrón | colonial
Frames:	***Jerarquía de objetos***	Normalmente expresados en notación UML.
Ontologías:	***Jerarquía de conceptos***	Recogidas en archivos OWL de una ontología.

Tabla 2.1: Niveles y modelos de la Representación de datos

El siguiente paso en la representación de datos es ocupado por los (**4**) *frames*, que incorporan cierta inteligencia a la información al permitir que los objetos puedan heredar las propiedades de otros objetos. Mediante *frames* es posible asociar a cada una de las propiedades de los objetos, métodos que se ejecutan cuando se interroga o se establece el valor de dichas propiedades (e.g. los métodos *getting* y *setting*). Un paso a mayores en la representación de datos son las (**5**) *ontologías*, que permiten especificar conocimiento [Gruber, 1995] y que están compuestas por entidades responsables de establecer la parte estática de la representación del dato denominadas *conceptos del dominio* donde se establecen además, cuáles son las *relaciones* existentes entre los *conceptos* y las *tareas*, y cuáles *tareas* pueden llevarse a cabo empleando a dichos *conceptos*. Esos dos últimos elementos de las ontologías representan las *acciones*, también conocidas como la parte dinámica del dominio.

Resolución de problemas: Destacan *Joseph* y *Jenifer Bigus* que, para los investigadores en AI el campo de la *resolución de problemas* ha ocupado una parte importante de sus estudios y publicaciones. Esas investigaciones se concentraban en *cómo* resolver problemas; no sólo en mostrar las soluciones de cómputo finales pues, como indican en [Bigus & Bigus, 2001] cualquier ejercicio de *resolución de problemas* requiere en primer lugar, la definición breve y precisa de «qué se trata de hacer». Y una vez conseguido ese primer paso fundamental en la resolución de problemas, lo más crítico es entonces «...la representación del problema en el formato y en los términos del ordenador».

El patrón más útil para la *resolución de problemas* que ha llegado hasta nuestros días, es la aproximación mediante el *espacio de estados finitos*. Se trata de descomponer el problema en fotogramas como si se tratase de instantáneas en las que participan los elementos del problema que existen en dicha etapa. La primera de esas instantáneas es el *estado inicial*. Una vez lograda dicha representación inicial, se le incorporan el conjunto de *operadores* que pueden ser utilizados para modificar dicho estado y provocar el avance hacia un nuevo escenario

formado por elementos y operadores distintos. La combinación de los operadores y el estado inicial forman lo que se conoce como el '*espacio de estados*' del problema.

Esa secuencia de *estados* se denomina *path*. Cada secuencia se acumula en la memoria y, cuando se localiza una de dichas secuencias con el modelo que cumple con todas las condiciones del problema en ese instante, entonces queda seleccionada y provoca la consecución y el acercamiento hacia el objetivo final. La última secuencia recibe el nombre de *estado final*. Existen problemas en los que resulta imposible definir un *estado* plenamente satisfactorio por lo que, este modelo tiene que emplear el enfoque del *coste* o *máximo beneficio*, para decidir cuál es la opción más adecuada en cada transición de un estado a otro.

Si un problema puede ser representado mediante un conjunto de estados finitos y de operadores que lo hagan avanzar hacia posibles estados más cercanos a la solución, entonces dicho problema puede ser solventado mediante este modelo de resolución de problemas donde es posible emplear algoritmos de cuantificación de mayor o menor *coste* para cada uno de los caminos que pueden ser seleccionados para transitar de un estado al siguiente. Establecen *Russel* y *Norvig* en [Russell & Norvig, 2010], tres aspectos importantes a considerar en relación al rendimiento de dichos algoritmos:

- Un *algoritmo* se considera óptimo si es capaz de encontrar la mejor opción entre un conjunto de soluciones todas válidas simultáneamente.
- Una *estrategia* es completa cuando garantiza que encontrará la solución, siempre que esa solución exista.
- La *complejidad* puede ser medida en términos de tiempo y de espacio, dando lugar a 'complejidad por consumo de tiempo' y 'complejidad por número de estados' que es necesario considerar y abarcar de forma simultánea por parte del algoritmo para aportar una opción adecuada a la solución del problema.

2.2.3 Sistemas de producción

Los *sistemas expertos* de encadenamiento hacia delante se denominan *sistemas de producción*. Hoy continúan representando uno de los logros más notables de la AI al ofrecer un modelo que permite traspasar el conocimiento al ordenador e incorporar la información del problema en si misma. Están construidos sobre sistemas expertos genéricos basados en *reglas* y utilizan esa *representación del conocimiento* como el elemento clave para construir los razonamientos [Mas, 2005].

Toru Ishida [Ishida, 1994] recuerda que cuando los *sistemas de producción* fueron estudiados como modelos de conocimiento para humanos o para agentes inteligentes y *Forgy*, logró la implementación del algoritmo de reconocimiento de patrones de alta velocidad [Forgy, 1982], «...los *sistemas de producción* se convirtieron en una de las herramientas más populares de construcción de *sistemas expertos*». Como indica *Giarratano* en [Giarratano, 2014], se ha corroborado la certeza de aquella afirmación y la propagación de aquel efecto en una de las herramientas de construcción se sistemas expertos más extendida en el mundo: CLIPS (C++ Language Integrated Production System), un compilador escrito en Lenguaje C++

creado en el Centro Aeroespacial Lyndon B. Johnson de la NASA en los ochenta, que aporta el Lenguaje de programación del mismo nombre y que permite la construcción, depuración y ejecución de *sistemas expertos basados en reglas con encadenamiento hacia delante*, o para abreviar, *sistemas de producción.* Por su parte el Lenguaje *Prolog*[6], desde principios de la década de los setenta hasta el presente, es la referencia de los *sistemas expertos* dirigidos por objetivos o de encadenamiento inverso. Por supuesto, con *Prolog* también es posible programar sistemas expertos de encadenamiento hacia delante.

Para *Ishida* [Ishida, 1994] un *sistema de producción* está definido por un conjunto de *reglas* o producciones que denomina (PM *Production Memory*) y por los datos contenidos en forma de *hechos* en la memoria de trabajo (WM *Working Memory*). A la *representación del dato* en un sistema de producción se le conoce como *hechos* y cada *hecho* concreto es un elemento de la memoria de trabajo (WME, *Working Memory Element*). Cada *regla* o *producción* consiste en una agrupación de elementos condicionales denominada (LHS, *Left Hand Side*) de la regla y de un conjunto de acciones llamadas (RHS, *Right Hand Side*). Se muestra a continuación un ejemplo de *regla* escrita en Lenguaje *OPS5*, escrita por *Forgy* en 1981:

```
(  p make-possible-trip
       (city ^name <x> ^state New-York)
       - (weather-forecast
       ^place <x> ^date tomorrow ^weather rainy)
 -->
       (make possible-trip ^place <x> ^date tomorrow)
)
```

Figura 2.2: Regla de encadenamiento hacia delante en OPS5 [Forgy, 1981]

La *agenda* es un componente clave en los *sistemas de producción*. Su misión es acumular las *reglas* que han sido satisfechas por los *hechos* presentes durante un *estado* concreto del sistema. La *agenda* incorpora sólo la parte derecha de la *regla* que se ha visto satisfecha, a la que aporta un espacio de espera antes de ser ejecutada.

Una vez entra en fase de disparo; todo el sistema experto al completo se prepara para realizar un cambio desde el estado previo a uno posterior al disparo de la *regla*. Por ese motivo el núcleo del *sistema* necesita bloquear todas las alteraciones e incorporaciones de nuevos *hechos* a la memoria de trabajo WM o, la incorporación de nuevas *reglas* en la memoria de producción PM hasta que concluya el ciclo (Figura 2.3).

Un segundo elemento, que no es distintivo de los *sistemas de producción* en exclusiva sino que fue una consecuencia de la evolución de este tipo de sistemas, fue la aparición del módulo para adquirir nuevos conocimientos. Se encuentra ubicado entre la *interfaz del usuario* y el *núcleo* que compone el motor de inferencia. Ese módulo adicional permite al usuario final introducir nuevas reglas en el sistema, sin requerir la presencia del *ingeniero de conocimiento*

---

[6] ***Prolog***: Acrónimo de *Programmation en Logique (Programming in Logic).* Inventado por *Alain Colmerauer* en Francia y *Robert Kowalski* en Inglaterra. *Prolog* es un lenguaje de programación declarativa. En programación declarativa el programador especifica el objetivo que debe ser alcanzado como solución al problema planteado.

para su codificación. El mecanismo empleado para adquirir conocimiento es una función opcional en muchos sistemas expertos actuales. Ciertas herramientas utilizadas para la construcción de sistemas expertos incorporan técnicas de inducción como mecanismo para obtener conocimientos. Sin embargo, las reglas que establece el *ingeniero de conocimiento* son siempre más complejas que las obtenidas por estos procesos automatizados.

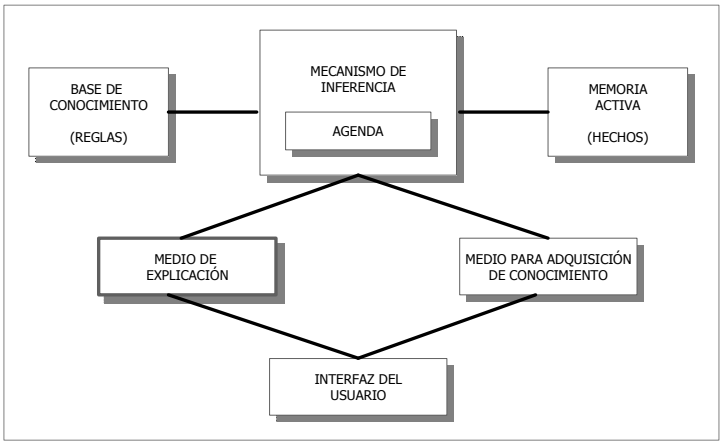


Figura 2.3: Esquema simplificado de un sistema de producción

2.2.4 Encadenamiento hacia delante

Ernest Friedman-Hill [Friedman-Hill, 2003], [Friedman-Hill, 2016] lidera en Sandía National Laboratories del gobierno de los EE.UU., el proyecto de desarrollo del compilador *Jess*[7], herramienta de construcción de *sistemas de producción* desarrollada íntegramente en Lenguaje *Java* similar a CLIPS. Para *Friedman-Hill*, el *motor* de un *sistema de producción* bascula continuamente entre dos fases:

(1) ***Reconocimiento de patrones***. Fase asumida por el *motor de inferencia*, donde se comprueba si existen *reglas* presentes en la memoria de producción (PM) que son satisfechas por los nuevos hechos en la memoria de trabajo (WM) y si es así, trata de incorporar a esas reglas como 'reglas-activadas' dentro de la *agenda*.

(2) ***Ejecución de reglas***. Fase asumida por el *motor de ejecución* destinada a ejecutar o disparar las *reglas* acumuladas previamente en la *agenda*. Ese acto modifica a su vez los *hechos* contenidos de la memoria de trabajo (WM) y en ocasiones, modifica incluso a *reglas* presentes en la propia memoria de producción (PM). Esto cierra el *ciclo de ejecución* del sistema (*véase* Figura 2.4).

Un elevado número de problemas pueden ser resueltos mediante este tipo de sistemas de encadenamiento hacia delante. Sin embargo, la mala utilización de este modelo de encadenamiento tiene su más afamado y triste exponente en el sistema XCON de la firma (DEC, Digital Equipment Corporation), que poseía capacidades para entregar la configuración física

[7] ***Jess***: The Rule Engine for the Java Platform (c) 2003-2016 Sandia Corporation.

óptima de microprocesadores en los ordenadores en función de los datos de entrada que recibía de las peticiones de los clientes. El sistema XCON gestionaba dichos datos y trabajaba en un proceso de tipo *encadenamiento hacia delante*. Los autores en [Barker et al., 1989] destacaban cómo el sistema —que poseía originalmente sólo 210 *reglas* para validar los pedidos de configuraciones hardware de la firma DEC— acumulaba nuevas *reglas*, alcanzando en 1989 la cifra de 17.500 *reglas* de comprobación de configuraciones posibles, conteniendo además los datos sobre unos 31.000 componentes hardware distintos. *Barker*, *O'Connor*, *Bachant* y *Soloway* reportan en el citado artículo que el 50% de las *reglas* de XCON tenían que ser anualmente modificadas. Años más tarde *Ishida* [Ishida, 1994] recuerda que una de las mayores dificultades para el mantenimiento de XCON, como sistema experto a gran escala, fue el *control de la ejecución* del disparo de las *reglas* pues «...tuvo que ser implementado en el interior de las mismas».

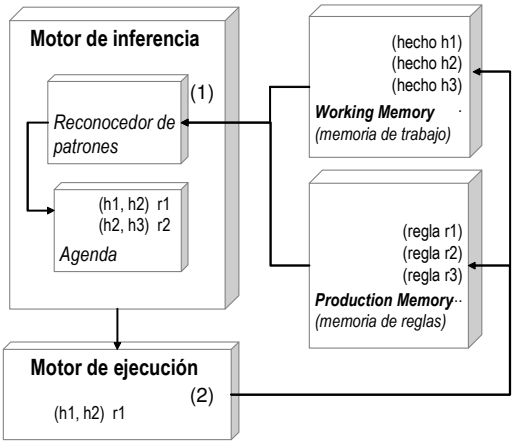


Figura 2.4: Motor de un sistema de producción [Friedman-Hill, 2003]

Volviendo sobre las palabras de *Ishida* en su artículo [Ishida, 1995] publicado en la primera conferencia internacional de sistemas multiagente, indicaba que tal vez fue desafortunado alcanzar el primer gran éxito de los *sistemas de producción* con el sistema XCON porque tal éxito generó unas expectativas irracionales que asumían que «...los sistemas expertos podrían resolver problemas de alta complejidad sin la descripción de procedimientos». *Ishida* recuerda que tales palabras fueron malinterpretadas y fueron la causa de erróneas visiones y esperanzas sobre estos sistemas, centrando el origen y la causa del problema en la utilización del Lenguaje OPS5[8]. Ya que, por ese motivo, sólo era posible el encadenamiento hacia delante y, al contrario de lo que sucede en Lenguaje *Prolog*, en OPS5 no existía un mecanismo de encadenamiento inverso. Por ello, resultó necesario idear un algoritmo de inferencia inversa que retrocediese en la dirección de las *reglas* precedentes para encontrar un camino posible que le llevase hasta dicho objetivo. Ese mecanismo fue implementado como un algoritmo en el sistema XCON que 'enterraba' dentro de las propias *reglas*, el proceso de ejecución del disparo de reglas. *Ishida* sentenció entonces que: «...se estaba solicitando a XCON crear una

[8] ***OPS5***: Intérprete creado por Charles L. Forgy y traducido a LISP por *Wood and Jim Kowalski*.

secuencia de acciones o lo que es lo mismo, se le estaba pidiendo *planificar*[9], que es exactamente aquello para lo que no estaba estructuralmente diseñado el Lenguaje OPS5».

2.2.5 El conjunto-conflicto

Joseph Giarratano destaca en [Giarratano, 2014] como, entre ciclo y ciclo en un *sistema de producción* construido mediante CLIPS, las *reglas* que han quedado activadas dentro de la *agenda* son ordenadas en función de sus índices de prioridad. A ese proceso de ordenación se le llama *Resolución de conflictos*, pues elimina los problemas sobre cuál debería ser la *regla* a disparar en primer lugar frente a las demás. Durante el estado de almacenamiento y espera dentro de la *agenda*, las *reglas* reciben la denominación de *conjunto-conflicto*. El disparo de una *regla* es un proceso que inicia el motor de inferencia y que provoca el cierre del sistema. Prosigue con la ejecución de la parte derecha de la *regla* RHS con prioridad (*salience*) más elevada y, una vez ejecutada, se elimina del conjunto de *reglas activadas* en la *agenda*.

El proceso de ejecución de *reglas* es llamado 'disparo' (*firing)* por su similitud con la actividad sufrida por las neuronas del cerebro humano tras dispararse y caer en un período de refracción que provoca su inactividad durante un tiempo prolongado. El motor de ejecución de un *sistema de producción* emula ese efecto de refracción pues, de no ser así, caería en un bucle infinito de activaciones debido a que esa misma *regla* que acaba de ejecutar, se activaría una y otra vez por la presencia en la *memoria de trabajo* de los mismos *hechos* que provocaron su activación.

2.2.6 Sistemas de producción distribuidos

Ishida y *Gasser* en [Gasser & Ishida, 1991], [Ishida, 1994] e [Ishida, 1995] clasifican los sistemas de producción concurrentes en:

- ***Sistemas de producción paralelos síncronos***, conocidos como *sistemas de disparo paralelo*, donde todas las reglas de la memoria de trabajo WM son globalmente sincronizadas en cada ciclo.

- ***Sistemas de producción paralelos asíncronos***, conocidos como sistemas de *producción distribuidos*, donde las reglas se disparan en paralelo sin una sincronización global.

Establecen así, las bases para la definición de los sistemas de producción multiagente:

- ***Sistemas de producción multiagente***, ambos autores los definen como «...aquellos donde múltiples programas gobernados por *sistemas de producción*, compiten o cooperan para resolver uno, o múltiples problemas.»

---

[9] ***Problemas de planificación***: Un ejemplo de la dificultad que supone implementar la planificación contando exclusivamente con la capacidad de encadenamiento de reglas hacia delante, puede encontrarse en directorio `/examples/` de la distribución estándar de CLIPS, en el ejemplo *mab.clp* conocido como el problema *The Monkeys and Bananas*, que expone una posible implementación del problema común de *planificación* en AI sobre un *sistema de producción*.

Las arquitecturas que permiten compartir información son la clave para que múltiples *agentes* (nombrados originalmente como *actores*) interactúen compartiendo la *memoria de hechos*. Esas arquitecturas permiten compartir la información y facilitan la implementación de los siguientes paradigmas de resolución de problemas:

- ***Paradigma de resolución de problemas orientado a datos.*** Tiene su máxima expresión en los sistemas de resolución de problemas *basados-en-casos* y ha sido ampliamente aceptado como un complemento adecuado para los *sistemas de resolución de problemas* que emplean tecnologías de sistemas expertos basados en reglas. Este paradigma propone una solución basada en la utilización de un motor de bases de datos de manera que la *memoria de trabajo* (donde se sitúan los *hechos*), no es sólo una memoria a corto plazo sino que mantiene los hechos de forma persistente para que puedan ser alterados y modificados por múltiples sistemas de resolución de problemas de manera concurrente.

- ***Paradigma de resolución de problemas cooperativo y distribuido.*** Es el paradigma donde múltiples actores autónomos pueden cooperar para resolver un problema único. La manera de mantener la consistencia entre los distintos agentes sigue siendo motivo de debates porque, pequeñas variaciones en los algoritmos de sincronización provocan grandes diferencias de rendimiento.

2.3 Sistemas DPS

Daniel Corkill [Corkill, 1991] establece una visión retrospectiva de los *Sistemas DPS* a los que denomina *Sistemas de Pizarra* (BBS, *Blackboard Systems*), y destaca que fueron el primer intento de integración orientada a permitir la *cooperación* entre módulos independientes de software con el objetivo de alcanzar el paradigma de la flexibilidad en el campo de la resolución de problemas emulando «...la *metáfora* de los sabios en distintas áreas del conocimiento que intercambian sus soluciones sobre una pizarra compartida por todos y donde el problema avanzaba hacia la solución a medida que se producían las intervenciones de cada uno de ellos». *Durfee* y *Rosenschein* exponen en [Durfee & Rosenschein, 1994], una visión comparativa de los *sistemas DPS* frente a los *sistemas multiagente*, de gran utilidad para la comprensión de los *sistemas DPS* y con un tratamiento preciso del contexto histórico donde se acuñó finalmente el término *sistemas multiagente*.

Durfee y *Rosenschein*, alertan a los investigadores noveles que se incorporan a esta rama del conocimiento sobre las sutiles, aunque importantes, diferencias que existen entre las investigaciones en *sistemas DPS* y las investigaciones en *sistemas multiagente*. Voz a la que se unen *Lesser* y *Corkill* en el ensayo revisado en 1995 sobre *tendencias en software cooperativo distribuido de resolución de problemas* [Durfee et al., 1995]. Denotan que, a mitad de los años setenta, las investigaciones en AI ya habían alcanzado significativos progresos en métodos para el tratamiento intensivo del conocimiento. Con éxitos como los *sistemas de producción*, capaces de acumular el conocimiento en sencillas líneas de código (*reglas*) como piezas independientes, se establecieron las bases para las propuestas de construcción de sistemas modulares tratando de emular la *metáfora de la cooperación de expertos*. De esa forma, los *sis-*

temas de pizarra y los *marcos de Actores*[10] aglutinaron muchas de las propiedades de las ideas emergentes en aquella década y de los trabajos sobre *redes-neuronales* que se encontraban inspirados en modelos biológicos. Todos en conjunto, formaron el escenario de fundamentos óptimo que, junto al florecimiento de las tecnologías de las comunicaciones, llegó a convertir a la *inteligencia artificial distribuida* «en la rama más prometedora de la AI».

Continúan los autores en [Durfee & Rosenschein, 1994] situando a los *sistemas DPS* en la raíz de la *inteligencia artificial distribuida* donde formalmente orientaban sus investigaciones hacia: el campo de la resolución de problemas, a resolver problemas mediante sistemas distribuidos, a gestionar agentes (o nodos de resolución) con múltiples capacidades conectadas a fuentes de conocimiento remotas, etc. Investigaciones que se materializaron en proyectos reales como por ejemplo el sistema para la resolución de problemas de tráfico aéreo y problemas de monitorización de tráfico rodado (DVMT)[11]. Otro ejemplo destacado de aquellos trabajos iniciales fue la descomposición y localización de las tareas sobre una red de comunicaciones mediante la aproximación Contract-Net (CN).

La evolución hacia la *nueva metáfora* [Durfee & Rosenschein, 1994], como llaman los autores a la incorporación de los *sistemas multiagente* a la *inteligencia artificial distribuida*, fue consecuencia natural de la evolución de los *Sistemas DPS* pues, esos sistemas se encontraban afincados en la resolución de problemas mediante el uso de ordenadores y «...una vez que los ordenadores pudieron ser programados, fue lógico asumir que los agentes desplegados en esos nuevos *Sistemas DPS*, podrían adoptar las acciones en los momentos adecuados y realizar el trabajo para el que fueron construidos».

Para *Durfee*, es en aquellos momentos cuando ven la luz las primeras investigaciones de *sistemas multiagente* (MAS), denominadas originalmente sistemas *cooperativos-distribuidos de resolución de problemas*, como refleja el ensayo de 1989, revisado por *Lesser y Corkill* en 1995 [Durfee et al., 1995]. *Durfee*, denota que numerosas investigaciones pioneras en sistemas *cooperativos-DPS* fueron construidas sobre la suposición de que existía un paralelismo entre el *modelo social* —presente en la literatura sobre *economía* y la *teoría de juegos*— y el comportamiento esperado en estos nuevos *sistemas DPS* formados por *agentes* con autonomía y capacidad de actuación y decisión propias. Pero, como recuerda *Durfee*, el intento de equiparar ambos modelos no fue adecuado porque la base sobre las que está construido el *modelo social* es el individuo y profundiza en las condiciones necesarias que hay que generar para que esos individuos (como personas) deseen realizar aquellas acciones esperadas o programadas. Ese modelo no opera adecuadamente con el comportamiento lineal del *agente*, capaz sólo de comprometerse, compartir tareas, comunicar la verdad a otros agentes, etc. Y esas propiedades son imposibles de alcanzar en grupos de individuos de un *entorno social*

---

[10] ***ACTORS Framework***: nota de traducción. Los marcos de actores representan el primer sistema de módulos con propiedades de autonomía frente a la pizarra, y son considerados el modelo precursor original de los sistemas multiagente.

[11] ***DVMT***: *Distributed Vehicle Monitoring Testbed*, desarrollado en la Universidad de Massachusetts bajo la dirección de *Victor R. Lesser*, en 1981 y un período de vida útil hasta 1991.

por lo que, el *modelo social*, no resultó el más correcto como base para las investigaciones en sistemas multiagente.

Ante esa realidad se indica en [Durfee & Rosenschein, 1994] que el foco de las investigaciones en los sistemas multiagente apuntó hacia el *agente*, tratando de aportar a esta *entidad-software* nuevas capacidades para interactuar de forma significativa con otros *agentes*. En investigaciones posteriores, indican *Durfee* y *Rosenschein*, que se inspiraron en el concepto oriundo de las Ciencias Sociales sobre el comportamiento racional de manera que, al trasladarlo al agente, sería equiparable al comportamiento que alienta maximizar el propio beneficio. Gracias a ese enfoque de racionalidad basada en el estímulo del beneficio propio, surgieron líneas de investigación en MAS dirigidas a identificar bajo qué condiciones de este tipo, los *agentes* podrían 'racionalmente' seleccionar sus actuaciones y obtener así las propiedades esperadas a partir de su comportamiento en *sociedades de agentes*.

El enfoque entonces adecuado para contextualizar los *sistemas DPS* y los *sistemas multiagente* es considerar que no se trata de dos sistemas con nombres diferentes que poseen ciertos puntos en común sino: «...de *líneas de investigación* separadas con un inicio común y con campos de investigación distintos». Concluyen los autores en [Durfee & Rosenschein, 1994] de forma sintética declarando que: «...en inteligencia artificial distribuida, existen agentes que actúan en un entorno para componer un sistema».

A través de las propiedades que exponen cada uno de esos tres elementos: *agente*, *entorno* y *sistema*, puede describirse con precisión si se trata de un *sistema DPS* o de un *sistema multiagente* (Tabla 2.2) en función de las respuestas aportadas a las siguientes cuestiones:

Matriz de Líneas de investigación por propiedades de agente, entorno y sistema

Línea de investigación	*Propiedades de:* ***Agente***	*Propiedades de:* ***Entorno***	*Propiedades de:* ***Sistema***
DPS	prefijadas	variables	prefijadas (externamente)
MAS	variables	prefijadas	prefijadas (internamente)

Tabla 2.2: Matriz para distinguir Líneas de Investigación [Durfee & Rosenschein]

- ***Propiedades de Agente:*** ¿Es único?, ¿Presenta un comportamiento racional?, ¿Posee preferencias?, ¿Son compartidas por todos los demás agentes?, ¿Cuáles son sus capacidades?, ¿Posee conocimiento de la existencia de otros agentes?
- ***Propiedades de Entorno:*** ¿Qué tipo de entorno?, ¿Es un entorno estático?, ¿Es cerrado?, ¿Es un entorno benigno?, ¿Hay resultados previsibles de las acciones tomadas por los agentes?, ¿Está temporalmente acotado?
- ***Propiedades de sistema:*** ¿Qué propiedades globales presenta?, ¿Asegura y gestiona el sistema, el acceso a los recursos?

Y establecen finalmente que: (**1º**) las *líneas de investigación* en *sistemas DPS* tienen como objetivo obtener sistemas con capacidad de respuesta, resiliencia y desempeño eficiente, bajo condiciones variables del *entorno* y con *agentes* con propiedades preestablecidas. Y (**2º**) que

las *líneas de investigación* en *sistemas MAS* tienen como objetivo central obtener un sistema con propiedades internas específicas que se ejecuta en un entorno acotado pero, donde los *agentes* poseen propiedades individuales variables y no necesariamente homogéneas entre los mismos.

Años más tarde *Durfee* en [Durfee, 2001] es aún más explícito al establecer que los *sistemas DPS*, «...demandan *coherencia* y *competencia* en cada uno de sus *agentes*». Esas propiedades significan que desde la etapa de diseño se establecen las propiedades necesarias en los agentes para que posean el incentivo necesario para trabajar juntos con precisión y además, cuenten con el conocimiento necesario para que puedan hacer dicho trabajo de forma correcta.

2.3.1 Software-colaborativo[12]

Daniel Corkill [Corkill, 2003] recuerda que, los investigadores en *inteligencia artificial* han utilizado el paradigma de los *sistemas de software-colaborativo* para abordar la resolución de problemas de elevada complejidad. Para *Corkill*, el paradigma del *software-colaborativo* es una aproximación eficiente de la máxima «*divide y vencerás*», aplicada al campo del desarrollo y del mantenimiento de grandes y complejas aplicaciones de software. En esas aplicaciones existe un conjunto numeroso de pequeños módulos de software que han sido desarrollados de marean independiente y que son acoplados posteriormente para conformar el sistema completo. De esa forma establece *Corkill* que: «... los detalles particulares y la complejidad asociada a cada módulo, es encapsulada en el interior de cada uno de esos elementos por lo que, para llevar a cabo los procesos de computación, resulta necesario que cada módulo posea capacidades de colaboración en un sistema formado por múltiples módulos».

La reflexión de *Corkill* establece que el paradigma del *software-colaborativo* es íntegramente aplicable a los *sistemas de pizarra* y a algunos, pero no a todos, los *sistemas multiagente* (Figura 2.5, pág. 25). Sitúa entonces un área ovalada extensa donde incluye por completo a los *sistemas de pizarra* y, parcialmente, a los *sistemas multiagente*.

Para *Corkill*, los estudios sobre *sistemas de pizarra* y *sistemas multiagente* basculan entre dos posiciones opuestas. En ambos paradigmas se encuentran otras subáreas de investigación relacionadas que incluyen a la i*ngeniería de software orientada a componentes*, la *programación a gran escala* (PIL, *Programming in Large*), los *sistemas de objetos distribuidos*, los *sistemas de soporte al trabajo en grupo* (CSCW, *Computer-Supported Cooperative Work*), los entornos de colaboración formados por *entornos humanos de colaboración* y el grupo software dedicado a la investigación de *comportamientos en sociedades biológicas*.

Y establece así *Corkill* finalmente, que existen 6 claves que han de ser tenidas en cuenta para implementar *software-colaborativo* de una manera efectiva:

---

[12] **Collaborating-software**: No existe una término en castellano para definir la calidad de colaboración. *Corkill* define *collaborating-software* como software de colaboración construido específicamente con cualidades para el reparto de tareas entre módulos interconectados y programado de esa manera, para facilitar al *ingeniero de software* emplear dichas capacidades.

1) Representación: Incorporar en los *módulos de software* capacidades suficientes para que puedan comprender los mensajes emitidos por otros *módulos*.

2) Conciencia: Construir los *módulos de software* con capacidades para procesar los eventos y sucesos del sistema, tan pronto se produzcan.

3) Investigación: Dotar a los *módulos* de capacidades para localizar con celeridad información acerca de las funcionalidades que poseen los demás módulos.

4) Interacción: Crear *módulos de software* con capacidades multitarea para que puedan trabajar y acometer con otros módulos tareas compartidas.

5) Integración: Construir los *módulos de software* con capacidades para combinar y aprovechar los resultados producidos por otros módulos.

6) Coordinación: Incorporar en los *módulos* capacidades para concentrarse en alcanzar sus objetivos y además, ejecutar dichos procesos en instantes precisos.

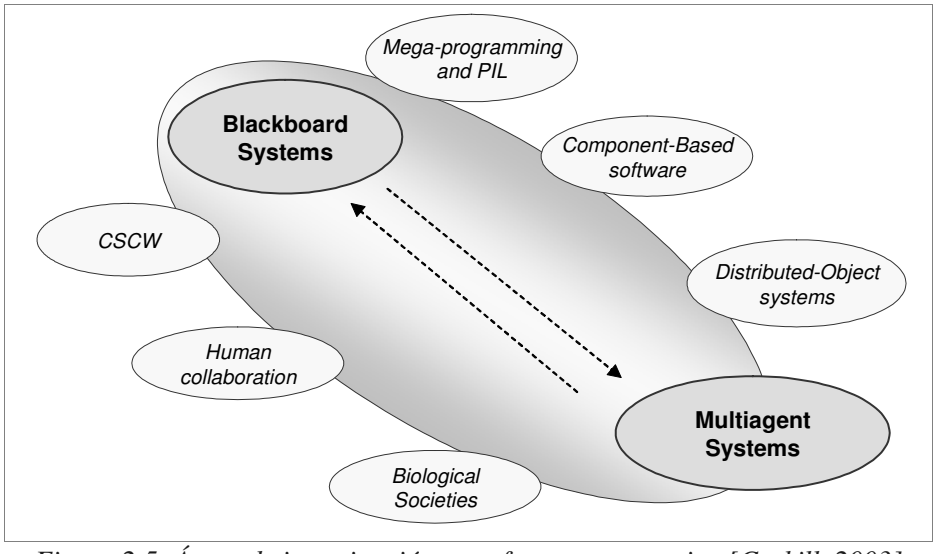


Figura 2.5: Áreas de investigación en software-cooperativo [Corkill, 2003]

2.3.2 Sistemas de pizarra

Establece *Francisco Garijo* [Garijo, 2005] que los éxitos de los resultados obtenidos en los *sistemas de producción* basados en *reglas*, donde el *conocimiento* había sido representado y codificado en pequeños segmentos, marcaron el comienzo de la modularización. Los *sistemas de pizarra* según [Erman et al., 1980] y [Hayes-Roth, 1985], fueron el primer ejemplo de organización modular en componentes especializados. La primera implementación de un sistema de pizarra tuvo lugar en 1970 para solucionar un conjunto de problemas de interpretación de señales acústicas (Hearsay-II). En aquellos momentos, indicaba *Corkill* [Corkill, 2003] que: «...las aplicaciones basadas en arquitecturas de pizarra eran las candidatas ideales para abordar ese tipo de problemas complejos que no permitían una sencilla estructuración».

La implementación para conectar los módulos que componían un sistema de información tradicional, consistía en ensamblar el conjunto de módulos utilizando los requerimientos que imponía el flujo de datos (*véase* Figura 2.6-izquierda, pág. 26). De esa manera, y cuantas veces resultaba necesario, los módulos aparecen tantas veces como ocasiones requería el flujo

de datos del sistema. Ese modelo de implementación funciona correctamente siempre que tanto módulos, como elementos de comunicación entre módulos, permanezcan estáticos.

En los sistemas donde resultaba necesario modificar los módulos o la ordenación del flujo de datos y por tanto de comunicaciones entre los módulos, ese modelo ya no era viable. Una vez que surgió la solución de interconexión entre módulos conocida como *arquitectura de pizarra* formada por un elemento con carácter neutro y anónimo que establece las comunicaciones entre los módulos, más un elemento intermedio que hace las veces de repositorio de datos, fue posible crear caminos de comunicación entre módulos gracias al 'mecanismo de control' que realizaba la moderación de los datos y mantenía el flujo hacia los módulos de manera dinámica. La información situada en la pizarra es pública y disponible para todos los módulos, incluido el propio *mecanismo de control*, motivo que permite incorporar nuevos módulos en el sistema y reordenar con sencillez el flujo de datos. Para *Corkill*, [Corkill, 2003] los *sistemas de pizarra* constan de:

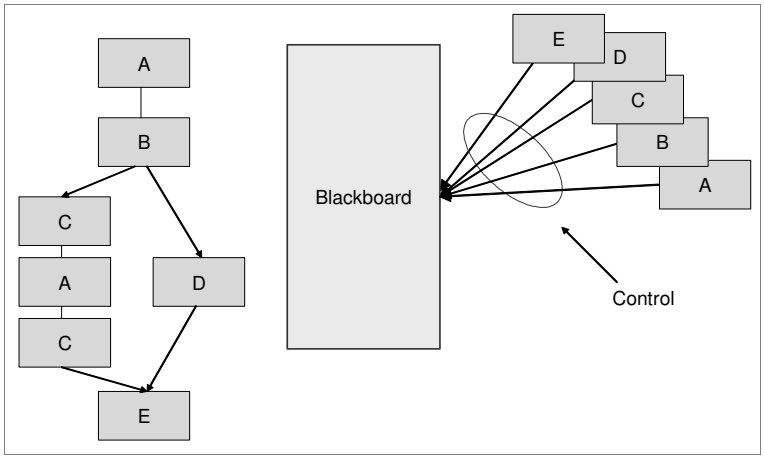


Figura 2.6: Conexión de módulos en arquitectura de pizarra [Corkill, 2003]

- ***Fuentes de conocimiento*** (*KS, Knowledge sources*). Módulos independientes del sistema que contienen la experiencia para resolver problemas concretos.
- ***Pizarra*** (*BB*, *Blackboard*). Repositorio global que contiene datos de entrada, soluciones parciales de la solución y otros datos que pertenecen a distintos estados de la solución del problema.
- ***Componente de control*** (Control Component). Módulo destinado a la toma decisiones, en tiempo de ejecución, en función del estado de resolución del problema, del coste de utilización de los KS y de la disponibilidad de los mismos.

La *pizarra* (Figura 2.7, pág. 27) es una estructura disponible para todas *Fuentes de conocimiento* (*KS, Knowledge sources*) y es utilizada como; (**1**) memoria común que alberga los datos recibidos, espacio de almacenamiento de soluciones parciales, soluciones alternativas y soluciones finales, e información de control para la comunicación entre módulos, (**2**) instrumento para la comunicación que aporta capacidades de *buffer* para asumir esperas o retrasos y (**3**) mecanismo de activación de las *Fuentes de conocimiento*.

Corkill define al *componente control* como la *shell de control* (Figura 2.7) y como el mecanismo responsable de dirigir el proceso de resolución de problemas, permitiendo y facilitando a los *módulos de conocimiento* intervenir en los instantes precisos en función de los cambios que se han observado en la *pizarra*. En los sistemas de *pizarra* las *fuentes de conocimiento* (KS) adoptan la forma de módulos funcionales e independientes y cada KS se encuentra especializado en la resolución de ciertos aspectos concretos del problema. Destacar que cada KS no requiere de otros KS subordinados para realizar su contribución en el proceso de resolución.

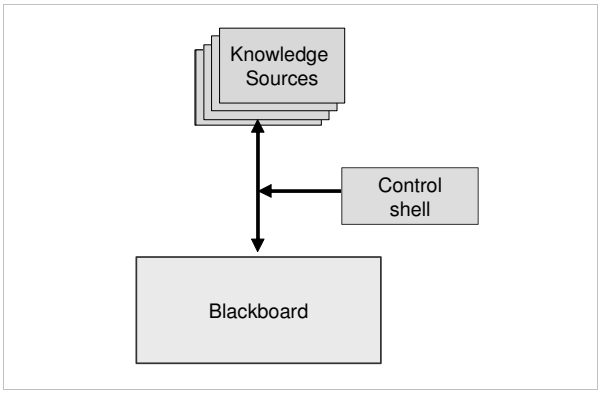


Figura 2.7: Componentes de la pizarra [Corkill, 2003]

2.4 Sistemas multiagente

Daniel Corkill [Corkill, 2003] coincide con *Edmund Durfee* [Durfee et al., 1995] cuando declara que los *sistemas multiagente* son «...otra tecnología distinta que puede ser utilizada para construir *software-colaborativo*» y demuestra, siguiendo un razonamiento por *reductio ad absurdum*[13], que resulta imposible utilizar la tecnología de los *sistemas multiagente* para construir nuevas aplicaciones basadas en el paradigma de la *arquitectura de pizarra* como el principio rector para dirigir el diseño e implementación de nuevas aplicaciones colaborativas. *Corkill* comienza proponiendo para la demostración que, la forma de utilizar la arquitectura de *pizarra* empleando la tecnología de los sistemas multiagente, sería asimilar al diseño seguido para construir *Fuentes de Conocimientos* (KS) empleando *agentes-software* tal y como se muestra en la distribución (**a**) de la Figura 2.8, pág. 29.

En algunos aspectos, integrar una *Fuente de conocimiento* (KS) en el interior de un *agente-software* sería una implementación considerablemente cercana a la resolución de problemas mediante la *metáfora de los expertos*. Una vez se alcanza ese punto de integración de las *Fuentes de conocimiento* en los *agentes*, advierte *Corkill* que los *sistemas multiagente* presentan grandes contrastes con los *sistemas de pizarra* ya que no poseen un repositorio central de datos y, cada *agente*, posee autonomía para comunicarse con los otros agentes de forma

[13] ***Demostración por contradicción***: *Reductio ad absurdum*, empleada por *Corkill* para demostrar la imposibilidad de conseguir que un sistema multiagente se comporte como un sistema DPS, empleando progresivas proposiciones de integración de los componentes.

privada sin estar sujeto a un control externo que los coordine, por lo que efectúan esas comunicaciones a partir de decisiones internas. Ese enfoque se opone a la mayor parte de las investigaciones en *sistemas multiagente* donde, lo que se busca, es potenciar propiedades en los *agentes* para conseguir diseños de *agentes* muy poco acoplados.

Corkill denota que el escenario marcado como **(a)** (Figura 2.8, pág. 29) exige la presencia de un *componente pizarra* o de algún *agente* con funciones de *pizarra* pues sería «...el único modo de avanzar en la resolución del problema ya que cada agente, debe decidir los datos que tiene que compartir y con qué agente tiene que hacerlo». La solución alternativa podría consistir en la emisión continua de los resultados parciales obtenidos por cada *agente* hacia todos los demás *agentes*. Pero, de nuevo, aparece el problema de conocer *a-priori*, qué datos recepcionar en esas emisiones masivas.

El escenario marcado como **(b)** en la Figura 2.8, propone dedicar un *agente* para que asuma las *funciones de pizarra* pero, tal como indica *Corkill*, esa solución recibió fuertes críticas de los investigadores en *sistemas multiagente* pues se encuentra en la dirección opuesta a los principios de autonomía y desacoplamiento que deben presentar los *agentes*. Sin embargo, el tiempo hizo que esos *agentes especializados* fueran tomando un lugar propio dentro de la tecnología de los *sistemas multiagente* y fueran finalmente aceptados porque eliminaban la necesidad de replicar, en todos los agentes, las capacidades específicas de un agente complejo, como podría ser e.g. un 'agente-de-bolsa'. Con esa aproximación, se demostraba que los agentes necesitan establecer comunicaciones con la *pizarra* para avanzar en la resolución de los problemas planteados y por tanto, pierden su capacidad de colaboración con otros agentes. Y aún más grave, no pueden decidir en qué fase de resolución del problema tienen que actuar. En suma, en el escenario marcado como **(b)** los agentes-software pierden su autonomía de toma de decisiones.

En el tercer escenario marcado como **(c)**, propone *Corkill* incluir, dentro del agente destinado a proveer de las *funciones de pizarra*, un *módulo de control* para arbitrar e informar a cada *agente* del momento en el que deben actuar y la forma en que han de hacerlo para resolver el problema.

En el escenario **(c)** el *sistema multiagente* se encuentra en una situación realmente alejada de los principios originales con los que fueron diseñados los agentes y las plataformas que los contienen. La autonomía se ha perdido y la capacidad de interacción se ha restringido. Como indica *Corkill*, cuando se realiza una observación directa del modelo de configuración **(c)** (Figura 2.8), es posible concluir que se ha implementado una arquitectura idéntica a la que forman las *Fuentes de conocimiento* (KS) de los *sistemas de pizarra* pero, sobre un sistema multiagente. Un paso hacia delante en la demostración, sería considerar viable la configuración del escenario marcado con la letra **(d)** (Figura 2.8), en el que se han incluido el componente *pizarra*, el *control* y las *Fuentes de conocimiento* dentro cada *agente*; alcanzando una nueva arquitectura de gran flexibilidad, que utiliza entidades de software que pueden alcanzar dimensiones pseudo-corporativas y que interactúan estrechamente en un entorno multiagente extenso que abarca varias corporaciones o, lo que es lo mismo, «...se ha llegado al modelo de una Intranet Corporativa moderna» [Corkill, 2003].

En el escenario **(d)** se pueden distinguir dos tipos de casos para alcanzar la resolución de problemas. En el primero cada *pizarra* en cada *agente* estaría conectada a una única *Fuente de conocimiento* (KS) del propio *agente* y es, por tanto ese agente, el que juega el papel de *pizarra compartida* para que —junto a los demás *agentes*— puedan alcanzar la resolución de problemas. En ese caso, **(d)** se convierte en el escenario **(c)**. Una segunda aproximación sería contemplar que los *agentes* poseen todas las *Fuentes de conocimiento* (KS) dentro de si mismos para alcanzar la resolución de problemas. En tal caso, indica *Corkill*, que se trataría de una aplicación de *software-colaborativo* homogéneamente distribuida. Por lo que, queda demostrado que ninguna de las cuatro configuraciones posibles representadas en Figura 2.8, alcanzaría la capacidad de resolución de problemas mediante el uso de *sistemas multiagente*.

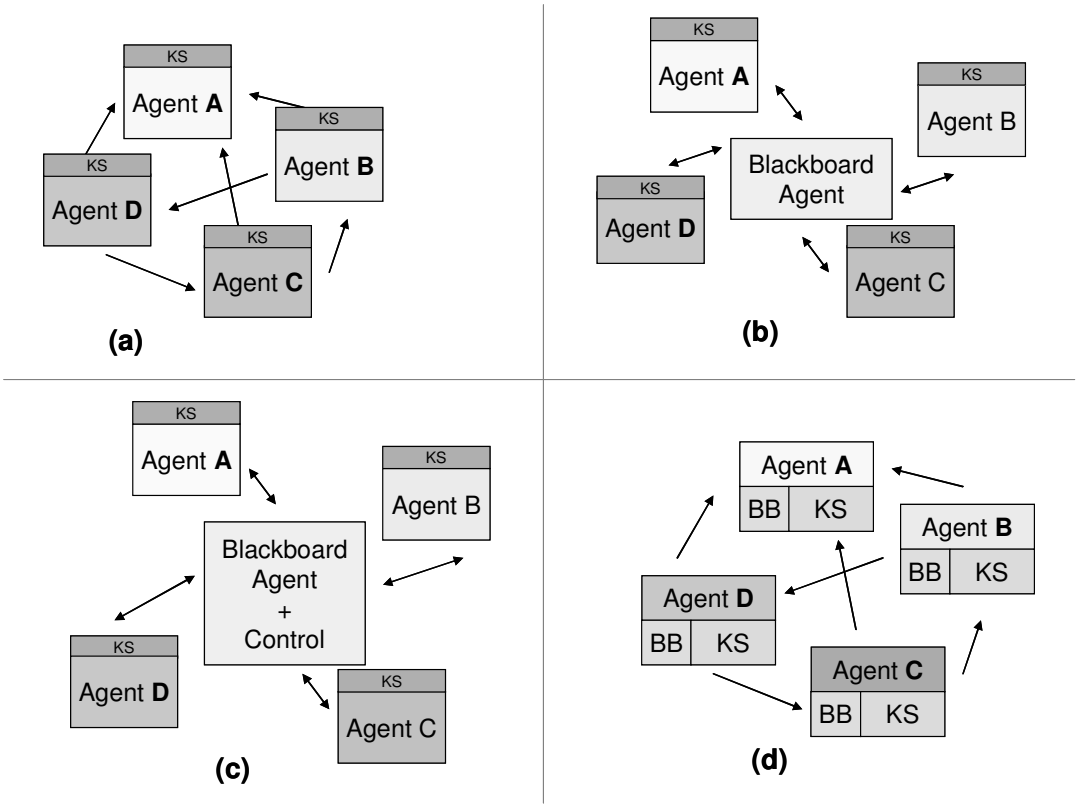


Figura 2.8: Sistemas colaborativos formados por agentes [Corkill, 2003]

Corkill [Corkill, 2003], recuerda que las líneas de investigación principales en tecnología de *sistemas multiagente* se concentraron en la construcción de *agentes* con períodos de vida largos y con capacidades para colaborar de manera concurrente en un escenario distribuido. Y aunque fueron elevados los rendimientos obtenidos mediante la utilización de *sistemas de pizarra*, no es posible olvidar la importancia que poseen aspectos como: (**1**) la gestión de la concurrencia de procesos y (**2**) la capacidad de distribución de los *módulos de software* que componen las aplicaciones construidas sobre *sistemas multiagente*. Y concluye destacando que «...lo más importante es utilizar la tecnología apropiada en cada contexto concreto».

Concluir con una referencia a *Costin Badica*, que junto a *Lars Braubach* y *Adrian Paschke* en [Badica et al., 2011] establecieron una clasificación de la arquitectura interna de los *siste-*

mas multiagente en función del tipo de agente y del tipo de marco (o arquitectura) que ha sido utilizado para la implementación de esos sistemas. De esa forma, consiguen ampliar el modelo de clasificación empleando conceptos como: sistemas multiagente con arquitectura basada en servicios, sistemas multiagente basados en computación GRID, sistemas multiagente que utilizan arquitectura *peer-to-peer* (P2P), sistemas multiagente en CLOUD, sistemas constituidos por redes inalámbricas gestionadas por *reglas* y, escenarios de procesamiento de eventos de alta complejidad.

2.5 Resumen

- Los *sistemas de producción* son sistemas expertos basados en reglas con mecanismos de inferencia dirigidos por datos o de encadenamiento hacia delante. Desde los 80 hasta la actualidad, siguen representando una de las herramientas más populares para la creación de sistemas expertos.
- Los *componentes esenciales* de un *sistema de producción* son la Memoria de Producciones (o Memoria de Reglas o, Base de datos de Conocimientos), la Memoria de trabajo, el Motor de inferencia (a su vez fragmentado en el motor de reconocimiento de patrones y en el motor de ejecución) y la interfaz de usuario final o *Shell*.
- Los *sistemas expertos* se clasifican según el *modo de razonamiento* en: sistemas de razonamiento basados en objetivos, sistemas de razonamiento borroso y sistemas de razonamiento dirigidos por datos, también conocidos como de sistemas de razonamiento por encadenamiento de reglas hacia delante.
- La clave de la resolución de problemas complejos se encuentra en la *representación del conocimiento* que, a su vez, es consecuencia directa del tipo de representación de datos.
- Los dos *paradigmas de resolución de problemas* son: orientado a datos (basados en casos) y paradigma de resolución de problemas cooperativo-distribuido donde múltiples actores-software pueden cooperar para resolver un problema único.
- *Daniel Corkill*, *Victor R. Lesser*, *Edmund H. Durfee* y *Jeffrey S. Rosenschein*, establecen: que las líneas principales de investigación en tecnología de *sistemas multiagente* se han concentrado en la construcción de *agentes* con períodos de vida largos y en la incorporación de capacidades para colaborar con otros agentes de manera concurrente en un escenario distribuido.

3 La plataforma de agentes

Cualquier tecnología suficientemente avanzada,
es indistinguible de la magia

Arthur C. Clarke
(Tercera Ley de Clarke)

Objetivos del capítulo:

- *Destacar las recomendaciones de FIPA aplicadas en esta investigación.*
- *Presentar propiedades específicas de la plataforma JADE.*
- *Exponer particularidades sobre los comportamientos de los agentes.*

3.1 Introducción

Este capítulo está dedicado a la descripción de la plataforma multiagente JADE, utilizada para la implementación de los modelos de integración agente-motor de inferencia obtenidos como resultado de la investigación y, como escenario donde comprobar su validez y adecuación. El apartado 3.2 está dedicado al concepto de agente software y a describir su modelo de gestión, su lenguaje de comunicación, la estructura de mensajes y los protocolos de interacción, todo ello de acuerdo a las especificaciones FIPA. El apartado 3.3 es una síntesis de la arquitectura y el subsistema de envío de mensajes en la plataforma multiagente JADE.

Fabio Bellifemine [Bellifemine et al., 2007] nos advertía de la problemática asociada cuando se adopta la decisión de utilizar la tecnología multiagente al destacar que: «existe un conjunto de dilemas que siempre han de ser resueltos previamente; uno de ellos es establecer, cómo se van a comunicar los agentes». Dicho problema es complejo y obliga a construir una capa intermedia de software o *middleware* en las aplicaciones que pretenden utilizar este tipo de tecnología. La plataforma multiagente JADE [CSELT & TILab, 2017] es uno de los marcos de desarrollo de sistemas multiagentes que aporta una solución a ese grave problema, que supondría tener que implementar la capa intermedia de comunicación entre agentes.

La plataforma JADE es simultáneamente, tanto el entorno donde se despliegan los agentes como la capa de la que se obtienen los servicios de comunicación necesarios para la interconexión de los agentes. Además JADE, es una plataforma que facilita a los agentes contar con su propiedad intrínseca relacionada con su habilidad para migrar hacia servidores remotos

gestionados por la plataforma multiagente (según se define en la normativa de declaración de propiedades que posee un Agente que es conforme con FIPA).

Para obtener una visión completa de la plataforma JADE se han empleado las publicaciones de los autores: *Giovanni Rimassa* [Rimassa, 2003], guías de programación de JADE [Bellifemine et al., 2010], libro de desarrollo de agentes JADE [Bellifemine et al., 2007], especificación [FIPA000025] dedicada a la librería de *protocolos de interacción* entre agentes —de la que cabe destacar la utilización del estándar de modelado UML por *Giovanni Caire* [Giovanni et al., 2003][14]— y, el propio software que implementa la plataforma de agentes JADE [CSELT & TILab, 2017] incluidas también, las versiones anteriores de la misma plataforma, que han sido utilizadas para corroborar la compatibilidad inversa y el funcionamiento de agentes con versiones antiguas del motor de inferencia *Jess* (inferiores a 8.0a1).

La plataforma JADE destaca, entre las plataformas multiagente disponibles, por su fidelidad en la adopción de las especificaciones de FIPA y por encontrarse desarrollada íntegramente en Lenguaje de programación *Java*.

3.1.1 Fundación para la estandarización de *agentes inteligentes*

FIPA es una organización sin ánimo de lucro de carácter internacional ubicada en Génova, Suiza. Está dedicada a la promoción de la industria de desarrollo de agentes-software inteligentes. Su actuación, desde 1997, ha sido llevada a cabo mediante la publicación de recomendaciones para el soporte de la interoperabilidad entre agentes software y entre aplicaciones de software basadas en agentes. La fundación FIPA nace de la colaboración abierta entre sus miembros[15] que forman parte de compañías y universidades con dilatada experiencia y actividad investigadora en el campo del desarrollo de los sistemas multiagente. FIPA ha publicado periódicamente los resultados de sus actividades, tanto hacia sus propios miembros como hacia la comunidad científica y a terceras partes interesadas, y con ello ha logrado contribuir a la consolidación del cuerpo de estándares que componen la definición de la *Agencia* en la actualidad.

Del vasto conjunto de especificaciones se enumeran las que han tenido especial relevancia para esta investigación: *Arquitectura abstracta* [FIPA00001], *Lenguaje de comunicación de agentes ACL* [FIPA00008], *Lenguaje del contenido* [FIPA00018], *Gestión de agentes en la plataforma* [FIPA00023], *Librería de protocolos de interacción* [FIPA00025], *Protocolo de interacción de suscripciones* [FIPA00035], *Librería de actos de comunicación* [FIPA00037], *Estructura de mensajes ACL* [FIPA00061], y la *Representación de mensajes mediante cadenas de texto* [FIPA00070].

---

[14] ***UML***: Unified Modeling Language. Véase: [Rumbaugh et al., 2007] y [Pilone & Pitman, 2005]

[15] ***Miembros de FIPA***: En enero de 2000, contaba con 56 miembros en representación de 17 países repartidos por todo el mundo. La información actualizada sobre FIPA se encuentra en http://www.fipa.org

3.2 Agentes software

Para *Bellifemine* [Bellifemine et al., 2007], el *agente software* o para abreviar el *agente*, «...es un tipo especial de componente que posee autonomía y que aporta unas interfaces para interactuar y comunicarse con otros sistemas y agentes, exponiendo comportamientos similares al actor humano». Desde la visión de la *Ingeniería de software*, un agente JADE es un objeto que extiende la clase `jade.core.Agent` y que se ejecuta de manera autónoma en un entorno cerrado. Estos objetos poseen capacidades de interacción con la plataforma multiagente y exponen métodos para incorporar sobre si mismos; comportamientos, recepción de mensajes, transmisión de mensajes, registro de su identidad en el directorio de agentes de la plataforma multiagente y registro de sus servicios ofrecidos a los demás agentes en el directorio de servicios, también conocido como *servicio de páginas amarillas*.

El modelo de ejecución de un *agente* en la plataforma JADE corresponde con el modelo de los *objetos activos* de *Java* (véase 4.2.3, pág. 53) que utiliza el modelo *multitarea* preemptivo o apropiativo característico de *Java* donde, la propia máquina virtual *Java* en coordinación con el sistema operativo subyacente, asigna tiempos de CPU y recursos a los procesos que se encuentran en ejecución [Beneventi et al., 2004]. Sin embargo, las tareas que lleva a cabo el propio objeto *agente* desplegado sobre la *plataforma de agentes*, son ejecutadas mediante una planificación progresiva gestionada por el agente con el modelo multitarea *cooperativa*.

Las tareas que lleva a cabo un *agente* son declaradas mediante objetos *behaviours* (o *comportamientos*) como especializaciones de la clase `jade.core.behaviours.Behaviour`. Esos objetos *comportamiento*, una vez creados, son incorporados al *agente* y reciben paulatinamente el control del hilo de ejecución cedido por el agente para iniciar su método `action()`. El hilo de ejecución es cedido por el objeto *comportamiento* hacia el *agente* una vez que su método `done()` devuelve el valor *true*.

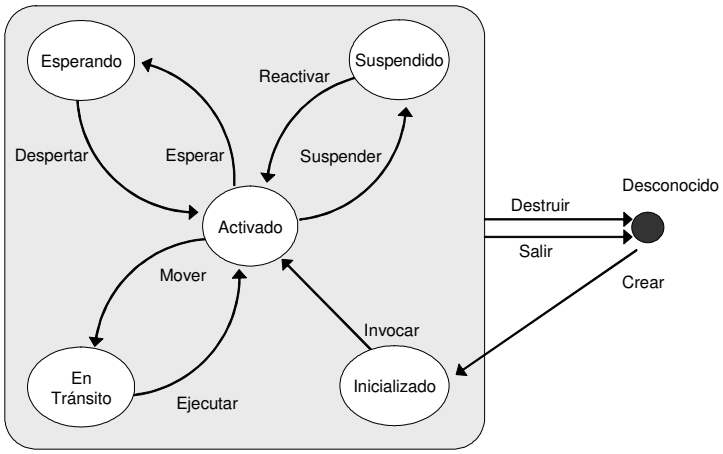


Figura 3.1: Ciclo de vida del agente conforme con FIPA. [FIPA00023]

En las plataformas de agentes conformes con FIPA un *agente* existe físicamente sobre la plataforma de agentes (**AP**, *Agent Platform*) y utiliza las funcionalidades que la AP le ofrece para desplegarse [FIPA00023]. En dicho contexto un *agente* es un proceso software físicamente visible que posee un *ciclo de vida* particular que tiene que ser gestionado por la plataforma.

Ese *ciclo de vida* (Figura 3.1) fue especificado por FIPA para los agentes-inteligentes y se define mediante *estados* que puede adoptar un objeto *agente* que se encuentran bajo la responsabilidad del *sistema de gestión de agentes* **AMS** (Figura 3.2, pág. 36). El *ciclo de vida* de un *agente* posee las propiedades siguientes:

- *Interno a la plataforma de agentes:* un agente está gestionado físicamente por la plataforma de agentes y el ciclo de vida de un agente está sujeto siempre a dicho espacio delimitado por dicha plataforma.

- *Independiente de la aplicación:* el modelo de *ciclo de vida* del *agente* es independiente de cualquier sistema y define exclusivamente estados y transiciones posibles del servicio ofrecido por el *agente*.

- *Orientado-a-la-Instancia:* el agente que asume el modelo del *ciclo de vida* es, en realidad, una instancia con nombre único en la plataforma que se encuentra en un estado de ejecución independiente.

- *Único:* cada agente sólo puede presentar un estado en la plataforma de agentes en un instante dado que es común para todas las plataformas donde puede actuar.

Los subsistemas recomendados por FIPA para conformar una plataforma multiagente poseen las responsabilidades siguientes en función del *estado* que presentan los *agentes*:

- *Agente en estado activado:* la entidad **AMS** (*Agent Management System*) (Figura 3.2, pág. 36), no interviene y el módulo de la plataforma de agentes destinado al transporte de mensajes, denominado **MTS** (*Messages Transfer System*), reencamina los mensajes del *agente* de forma habitual.

- *Agente* en alguno de los estados; *inicializado, en espera o suspendido:* el módulo de transporte de mensajes **MTS** almacena los mensajes hasta que el *agente* vuelve a alcanzar el *estado activado*, momento a partir del cual el **MTS** reenvía los mensajes al *agente*.

- *Agente en estado de tránsito:* el módulo de transporte de mensajes **MTS** almacena los mensajes que el *agente* haya recibido hasta concluir su traslado a la nueva localización. Si el traslado tuvo éxito, se reencaminan los mensajes hacia dicha localización y en caso contrario, el módulo **MTS** reenvía los mensajes hacia el agente en su posición original. Destacar que, sólo los agentes móviles pueden alcanzar el *estado de tránsito*.

- *Agente en estado desconocido:* el módulo **MTS** o bien almacena los mensajes, o bien los rechaza dependiendo de la política aplicada por el **MTS** y los requerimientos de transmisión de mensajes (*véase* pág. 15, en especificación [FIPA00023]).

FIPA describe las *transiciones* entre *estados* de un agente por los siguientes verbos:

- *Crear:* corresponde al cambio de estado desde la no-existencia hasta que el agente alcanza el estado *inicializado* en la plataforma. Denotar que, en estado *inicializado*, el agente ya se encuentra en el directorio de *páginas blancas* gestionado por el **AMS**.
- *Invocar:* es la transición desde el estado *inicializado* hasta el estado *activado*: coincide con instante de la llamada al método `setup()` del *agente*.
- *Destruir:* es la transición desde la existencia del agente en la plataforma hasta su *finalización forzosa*. Este cambio de estado sólo puede ser invocado por el agente **AMS** y no puede ser ignorado por el *agente*.
- *Salir:* es el cambio de estado que culmina con una *finalización* en términos de invitación dirigida al *agente* y, que además, puede ser ignorada por el agente.
- *Suspender:* cambiar el estado del agente desde *activado* hasta su estado *suspendido*. Puede ser ejecutada bien por el propio *agente* o por la autoridad **AMS**.
- *Reactivar:* es la transición que cambia el estado desde *suspendido* hasta *activado*. Esta transición sólo puede ser llevada a cabo por el **AMS**.
- *Esperar:* es el tránsito entre el estado *activado* y el estado de *espera*. Esta transición sólo puede ser llevada a cabo por el propio *agente*.
- *Despertar:* es la transición que lleva al *agente* desde su estado de *espera* al estado *activado*. Esta transición sólo puede ser llevada a cabo por el **AMS**.
- *Mover:* es el cambio de estado desde *activado* hasta el estado de *Tránsito*. Sólo puede ser iniciada por el propio *agente* y además, debe ser un *agente móvil*.
- *Ejecutar:* se trata de un tipo de transición que sólo puede ser realizada por el **AMS** pues, modifica el estado del agente desde su estado en *Tránsito* hasta *activado*, normalmente desplegado en otro contenedor de la plataforma multiagente. Destacar de nuevo que, sólo es aplicable en *agentes móviles*.

3.2.1 Modelo de referencia para la gestión del agente

El modelo de referencia para la gestión del agente «es un marco normativo establecido por FIPA en el cual los agentes existen y operan» [FIPA000023]. Con ese modelo lógico se establece la creación, registro, localización, comunicación, migración y retirada de un agente de la plataforma de agentes. Las entidades que están contenidas en este modelo de referencia (*véase* Figura 3.2, pág. 36), son agrupaciones lógicas de capacidades por lo que no implican una configuración física concreta. Además, los detalles de implementación de una plataforma de agentes quedan al criterio de los desarrolladores de sistemas multiagente.

Para FIPA, el modelo de referencia consta de los siguientes componentes lógicos:

- *Agente*: es un proceso computacional que implementa la autonomía y las funcionalidades de comunicación de una aplicación. Un Agente se comunica mediante el lenguaje de comunicación de agentes o, para abreviar (**ACL**, *Agent Communication Language*). Es la pieza central de una AP y combina una o más capacidades, como por ejemplo, estar públicamente accesible desde un *directorio de agentes*. Es necesario que el *agente* posea

un propietario (*owner*), una identidad única conocida como **AID** (*Agent Identification*) y, de manera opcional, puede tener un conjunto de direcciones adicionales o *alias* registrados a través de las cuales el agente puede ser contactado.

- *Servicio de Directorio* **DF** (*Directory Facilitator*): es un componente opcional de la AP que si está presente, debe adoptar el modelo de un *servicio* a libre disposición de los agentes. **DF** aporta funcionalidades similares a un directorio de *páginas amarillas* (*yellow-pages*) para localizar servicios prestados por otros agentes en la plataforma que han registrado sus capacidades sobre el servicio de directorio **DF**.
- *Sistema de Gestión de Agentes* (**AMS**): es un componente obligatorio en las plataformas de agentes conformes con el modero de referencia de FIPA. Ejerce funciones de supervisión y de control de acceso y utilización de la AP. Sólo puede existir un componente **AMS** en una AP. Ese componente mantiene una lista de las direcciones de identificación de los agentes (**AID**, *Agent IDentifier*), de los metadatos adicionales que completan la descripción del agente, direcciones, propiedades, etc. Por ello, el **AMS** debe incorporar un *Servicio de páginas blancas* de los agentes permitiendo la identificación unívoca de los agentes como entidades. Ese servicio obliga a los agentes a registrarse para obtener un identificador único y válido antes de alcanzar el estado: *activado*.

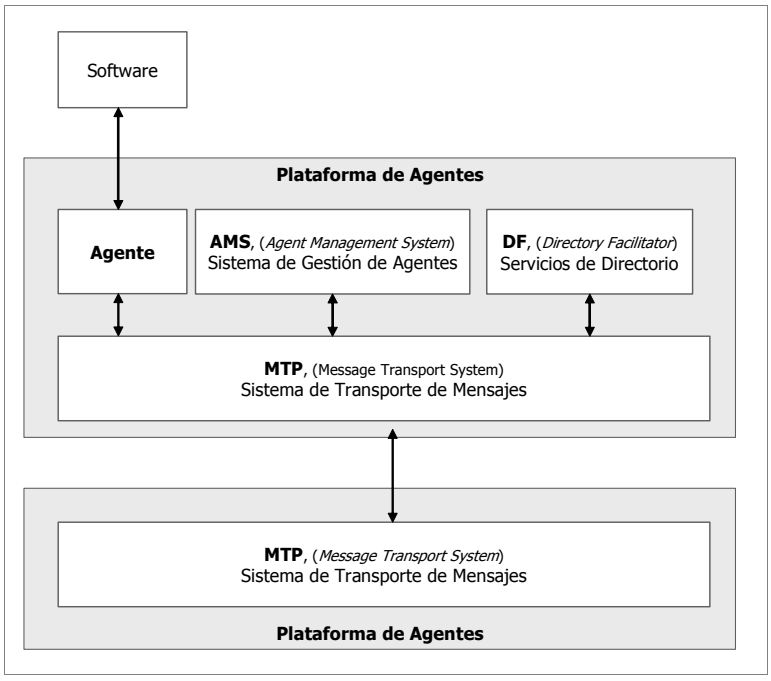


Figura 3.2: Modelo de referencia de la gestión del agente. [FIPA00023]

- *Servicio de Transporte de Mensajes* (**MTS**): es el mecanismo por defecto para comunicar a agentes localizados en diferentes plataformas, según se especifica en [FIPA00067].
- *Software*: resaltado en la Figura 3.2, representa a todas aquellas colecciones de instrucciones ejecutables que son accesibles para un *agente* pero que no son parte del agente. Un *agente* puede acceder y utilizar software, por ejemplo un *sistema experto basado en reglas*, con lo que añade nuevas capacidades y servicios a su perfil original de *agente*.

- Y finalmente, una *Plataforma de Agentes* (**AP**): que aporta la infraestructura física en la cual los agentes se despliegan. La AP estará constituida por las máquinas físicas, el sistema operativo, el software de soporte al agente, más los componentes obligatorios de FIPA, (AMS, MTS y DF) a los que se les suman los *agentes*.

3.2.2 Lenguaje de comunicación de Agentes

Justificaba *Dominic Greenwood* [Greenwood, 2004] que, una de las claves del éxito de la acogida de la normalización propuesta por FIPA fue la especificación ACL [FIPA00008] (**ACL**, *Agent Communication Language*). En copublicaciones realizadas por *Greenwood* con *Bellifemine* [Bellifemine et al., 2007] exponían que la buena acogida de esas propuestas de normalización fueron: «...gracias al método de trabajo impuesto por la propia FIPA, a la visión global que se trataba de mantener en todo momento al realizar de forma simultánea las normas sobre modelado, a la metodología, la semántica y los servicios de los *agentes*, a la vez que se avanzaba en la propia definición de las especificaciones y propiedades de codificación en ACL, y gracias, a no especificar los comportamientos o intenciones del *agente*».

En palabras de *Greenwood*, «...fue un esfuerzo no sólo tecnológico sino de coordinación humana para la promoción de tecnologías y especificaciones de interoperabilidad que facilitarían la interconexión extremo-a-extremo de sistemas de *agentes inteligentes* en un marco tanto académico como industrial». Para la fundación FIPA, el alcance del proyecto estuvo formado por: la gestión del *ciclo de vida* del *agente*, el *transporte* de mensajes, la *estructura* del *mensaje*, los *protocolos* de interacción entre agentes, la explotación de *ontologías* y la *seguridad*. Y como *Greenwood* recuerda «...de forma intencionada, el *agente* siempre permaneció fuera del alcance de las especificaciones FIPA».

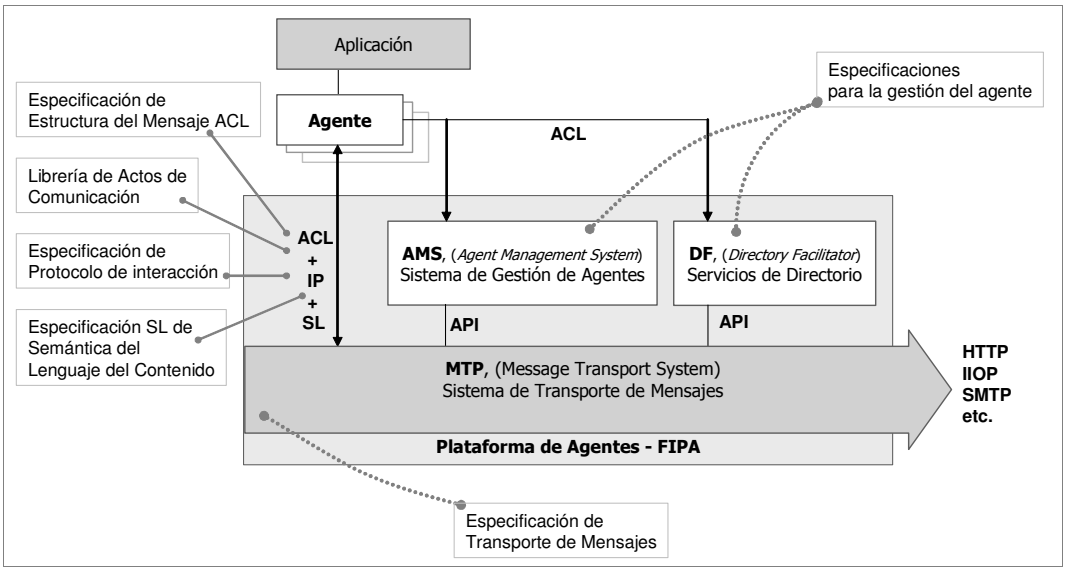


Figura 3.3: Visión de la Plataforma de Agentes FIPA [Greenwood, 2004]

Para *Greenwood*, las especificaciones de FIPA se encuentran fuertemente relacionadas y esa circunstancia no supuso una falta de flexibilidad. El *agente* como entidad de software no ha pertenecido al conjunto de especificaciones FIPA y por ese motivo se expone en una posición

externa a la plataforma de agentes, como se observa en la Figura 3.3. El *agente* se comunica con las entidades de la plataforma utilizando mensajes ACL. En ocasiones accediendo de forma directa a las entidades AMS o DF y en otras ocasiones, utilizando el *middleware* de comunicaciones MTP para establecer contacto y remitir mensajes ACL a otros agentes.

Ya sea en comunicaciones con entidades de la plataforma o con otros agentes de la misma, puede emplearse uno de los Lenguajes de Contenido (SL[16], LEAP[17], XML[18], cuando sea necesario captar el significado semántico y siempre que ese significado, sea analizable por el *agente* destinatario. Dichos lenguajes en FIPA se denominan de manera genérica (**CL**, *Content Languages*).

Estas especificaciones funcionan en bloque por su estrecha relación:

- Especificación de la Estructura de Mensajes ACL [FIPA00061].
- Especificación de Librería de Actos Comunicativos [FIPA00037].
- Especificaciones de la Librería de Protocolos de Interacción [FIPA00025].
- Especificaciones del Lenguaje del Contenido SL [FIPA00018].

Y esas normas y especificaciones incorporan propiedades a la especificación central que define el *Lenguaje de Comunicación de Agentes* **ACL**, la norma [FIPA00008].

3.2.3 Estructura del mensaje según FIPA

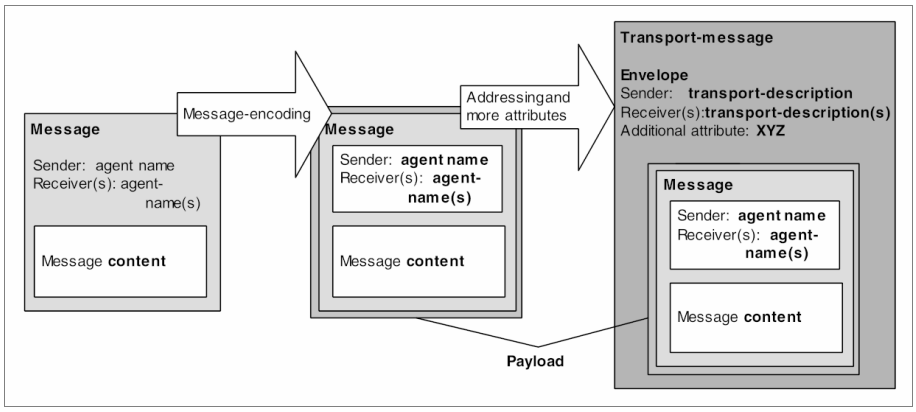


Figura 3.4: Proceso de transformación de un Mensaje [FIPA00001]

Para construir un *Mensaje ACL* (Figura 3.4) es necesario aplicar un proceso de transformación progresivo. Primero se toma el *contenido* y una vez decididos el remitente y el destinatario de dicho *contenido* se modela hasta alcanzar el concepto de *mensaje*. Ese *mensaje* se

---

[16] ***SL Content Language***: es un Lenguaje de contenidos expresado en cadenas legibles para humanos. Por ello, en SL las expresiones del contenido del lenguaje son cadenas y están basadas en la sintaxis S-Expression.

[17] ***LEAP Content Language***: es un Lenguage de contenidos byte-encoded. No es legible para humanos y las expresiones del contenido quedan codificadas como secuencias de bytes.

[18] ***XML Content Language***: utilización del Lenguaje XML para codificar el contenido de los mensajes remitidos y enviados entre agentes o entre el agente y las entidades de la AP.

transforma utilizando un esquema o estructura de datos. Las normas que definen dicha estructura son establecidas en los parámetros obligatorios de FIPA para los mensajes ACL y las representaciones disponibles para codificar dichos datos estructurados que pueden ser o bien en formato XML, notación de Cadenas FIPA, Binario, etc. El resultado de dicha codificación se denomina el *payload*.

Y se modela así el objeto destinado al *transporte del mensaje*, formando por la agrupación del bloque *payload* más un conjunto de atributos adicionales sobre transmisión por el medio físico conocidos como envoltorio o *Envelope*. Esos atributos del *Envelope* son entre otros: el nombre del esquema de representación utilizado en la codificación de datos del mensaje, los datos sobre seguridad aplicados al contenido, la encriptación aplicada al *payload* y otros datos específicos que deben ser visibles por la capa *middleware* de la plataforma, para que sea viable transportar adecuadamente —y de manera interoperable entre plataformas multiagentes distintas— el contenido del mensaje ACL.

En resumen, las entidades abstractas[19]: [*content* y *message*] forma el objeto *payload* que es transportado por el *envelope*, el cual es un objeto bien conocido por los mecanismos de transmisión de mensajes de todas las plataformas multiagente. En la Figura 3.5 (pág. 40) ha quedado plasmada la especificación [FIPA00061] donde se define cómo se recubren los datos de un mensaje en un formato de código interoperable. El objeto *mensaje ACL*, puede estar expresado mediante cadenas de texto plano, para o cual los agentes emisor y receptor, acuerdan utilizar sus capacidades (*véase* `GetContentManager` en las implementaciones de los comportamientos destinados a la transmisión de mensajes) y pueden utilizar una Ontología para obtener el significado semántico del contenido de dichos mensajes. Para FIPA, los conceptos definidos en la ontología *fipa-acl*, establecen la sintaxis y la estructura en la que se basan los *mensajes ACL*. Esa estructura queda a su vez definida, en los siguientes términos:

Frame	*fipa-acl-message*
Ontology	*fipa-acl*

Tabla 3.1: Frame y Ontology empleados para definir Mensajes ACL [FIPA00061]

- *Frame:* para la ontología *fipa-acl* es el nombre obligatorio de la entidad que representa a cada instancia de un *Mensaje ACL.*
- *Ontology:* es el nombre de la ontología *fipa-acl*, cuyo dominio define los parámetros de un mensaje ACL según FIPA, (véase Tabla 3.1).
- *Parámetro:* identifica a cada componente dentro de ese *Frame*. Cada *parámetro* es expresado en el código fuente (*encoding*[20]) de una forma única. Dicha codificación se utiliza en cada una de las respectivas especificaciones relacionadas.

[19] ***Entidades abstractas***: No hay una traducción al castellano comúnmente aceptada y se ha preferido mantener los términos en inglés para facilitar la comprensión del texto.
[20] ***Encodig*****:** corresponde con el concepto 'expresado en código-fuente.

- *Descripción:* es una definición en lenguaje natural del significado de cada *parámetro*. En ocasiones se pueden incluir notas aclarativas sobre su significado y uso.
- *Valores reservados:* Es una lista de constantes definidas por FIPA asociadas a ese *parámetro* en particular.

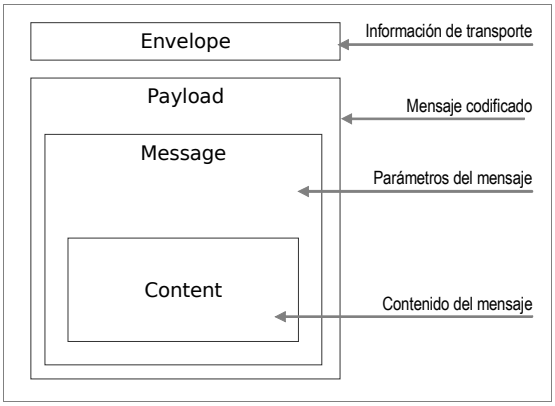


Figura 3.5: FIPA estructura del mensaje [Bellifemine et al., 2001]

Todos los parámetros en los *mensajes ACL* de FIPA, comparten el mismo *Frame* y la misma *Ontology* de la Tabla 3.1 por especificación [FIPA00061]. Los nombres de *parámetros* se relacionan en la Tabla 3.2, (pág. 42). Estos parámetros, se clasifican por categorías según:

A. Tipo de acto comunicativo
B. Participantes en la comunicación
C. Contenido del mensaje
D. Descripción del contenido
E. Control de la conversación.

El significado asociado a cada *parámetro* de un *mensaje ACL* es el siguiente:

- `performative`: valor entero que establece el tipo de *acto comunicativo* del mensaje.
- `sender`: nombre del emisor. En este caso, el identificador del *agente* emisor.
- `receiver`: una lista con los nombres de agentes destinatarios del mensaje.
- `reply-to`: es el nombre del agente al que deben dirigirse las respuestas en sucesivos mensajes de esta conversación, sustituyendo así al actual emisor del mensaje que habría sido el destinatario por defecto.
- `content`: es el contendido del mensaje (en formato texto, binario, XML, etc.). El significado del contenido de un *mensaje ACL* siempre es interpretado por el receptor del mensaje. Esto es particularmente relevante cuando el contenido incluye expresiones que presentan significados coherentes pare el emisor y el receptor.
- `language`: indica la versión del lenguaje de contenidos que debe emplearse para analizar expresiones en el contenido del mensaje.
- `encoding`: es el mecanismo utilizado para codificar los datos del mensaje. Pueden ser uno de los formatos: XML, Cadenas FIPA, LEAP, Binary, etc.

- `ontology`: nombre de la ontología que asocia significados a cada uno de los símbolos presentes en el contenido del mensaje.
- `protocol`: nombre del *protocolo de interacción* que deben seguir el emisor y el receptor del mensaje y que establece cuál será el modelo de conversación mantenida entre ambas entidades.
- `conversation-id`: es un identificador de la conversación para facilitar el mantenimiento del hilo de una conversación al identificar los mensajes asociados a la misma.
- `reply-with`: es una expresión que deberá utilizar cada uno de los agentes destinatarios cuando respondan con mensajes que pertenecen a una conversación mantenida por los receptores de forma conjunta con un mismo emisor.
- `in-reply-to`: es utilizado por el agente destinatario cuando responde a un mensaje dentro de una misma conversación pero, que se encuentra compuesta de múltiples diálogos. Coincidirá con el valor de la cadena que le haya venido incluida en el parámetro: `reply-with`.
- `reply-by`: es un valor de tiempo, expresado como fecha, que indica hasta cuando el emisor esperará recibir alguna respuesta a su consulta.

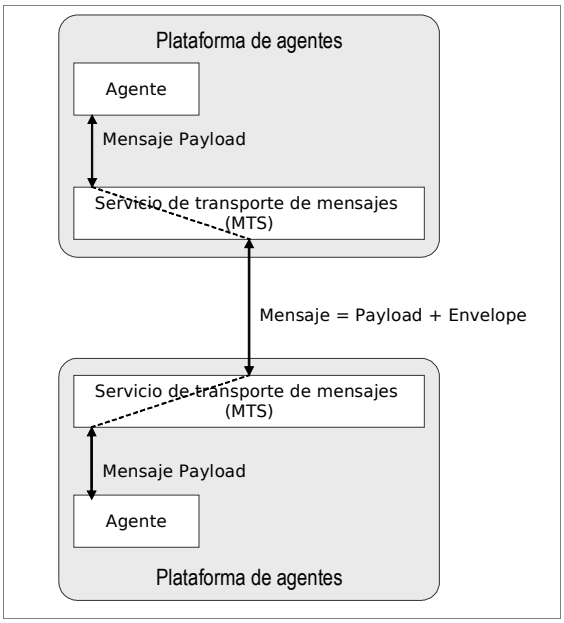


Figura 3.6: Modelo de referencia de transporte de mensajes [FIPA00067]

Un mensaje codificado en ACL contiene un conjunto de *parámetros*. El único obligatorio establecido por FIPA es *performative* aunque normalmente, se espera que también estén presentes los valores asociados a los parámetros: *sender*, *receiver* y *content*. Si un *agente* no es capaz de procesar alguno de estos parámetros a la llegada de un mensaje replicará con un mensaje con *performative* igual a `NOT-UNDERSTOOD`. FIPA deja abierta la posibilidad de utilizar otros parámetros que son modificados con prefijo '`X-`' para mejorar su identificación.

La especificación [FIPA00067], recoge de manera homóloga los parámetros relativos al objeto *Envelope* y completa así, el objeto abstracto de *transporte del mensaje*. Esos parámetros

descriptivos del *Envelope* son parte de las normas del servicio transporte de mensajes de agentes. El modelo de referencia para el transporte de mensaje entre agentes comprende tres niveles: **(1)** el protocolo de transporte del mensaje (**MTP**, *Message Transport Protocol*), **(2)** el servicio de transporte de mensaje (**MTS**, *Message Transport Service*), y **(3)** el *Mensaje ACL* codificado en el *payload* y recubierto por el *Envelope* que permite su transporte.

Desde un punto de vista abstracto un *mensaje de transporte* se compone de dos partes: una que expone sólo información de transporte denominada *Envelope* y otra que contiene la información del mensaje ACL emitida por el agente y denominado *payload* (véase Figura 3.5, pág. 40). A efectos de interpretación del contenido del mensaje por parte del *agente* receptor, sólo la sección del mensaje que mantiene la información en notación ACL y que se encuentra en el *payload*, tendría un posible valor semántico. Pero, todo lo demás, es información de transporte que se suma al mensaje [FIPA00023]. Esa información es de uso exclusivo del agente y no está definida por FIPA. Como siempre, bajo ciertas circunstancias y en concreto cuando el *payload* se encuentra encriptado, es necesario procesar la información presente en el *Envelope* del mensaje para que sea posible conocer cuál ha sido la codificación utilizada por el emisor para la representación de datos en ACL [FIPA00067].

Parámetros de la Ontología fipa-acl por categorías

Parámetro	*Categoría del parámetro*
`:performative`	A. Actos comunicativos
`:sender`	B. Participantes
`:receiver`	
`:reply-to`	
`:content`	C. Contenido del mensaje
`:language`	D. Descripción del contenido
`:encoding`	
`:ontology`	
`:protocol`	E. Control de la conversación
`:conversation-id`	
`:reply-with`	
`:in-reply-to`	
`:reply-by`	

Tabla 3.2: Parámetros en mensaje ACL categorizados [FIPA00061]

En la (Figura 3.6, pág. 41), el MTS es el servicio de transporte de mensajes. Aporta el mecanismo de transferencia de mensajes ACL entre agentes, ya se encuentre dichos agentes en una misma AP o en diferentes APs. En JADE este servicio lo expone el agente ACC (**ACC**, *Agent Communication Channel*).

El *Envelope*, descrito en el protocolo de transporte de mensajes MTP, se describe y se expresa en los siguientes términos:

(1) El *Envelope* del mensaje, se compone por un conjunto de parámetros.

(2) Los Parámetros del *envelope* adoptan una estructura de datos: `atributo-valor`.

(3) Parámetros obligatorios son: `to, from, data, acl-representation`.

(4) Existe la posibilidad de incorporar Parámetros adicionales.

Una vez que el mensaje llega al *agente ACC*, ese agente puede remitirlo directamente hacia los agentes destinatarios empleando el nombre del agente o de la lista de agentes que se encuentran en `intended-receiver`. El mensaje es transmitido por el *agente ACC* como un único bloque de información, incluyendo al *Envelope*, hasta ser recibido por el destinatario o destinatarios del mensaje.

3.2.4 Librería de protocolos de interacción de FIPA

La especificación [FIPA00025] aporta la estructura recomendada de las secuencias habituales seguidas durante los diálogos mantenidos por los entre agentes y da forma, a la recomendación denominada Librería de *Protocolos de Interacción* (IPL[21]) que incluye: los estados asociados a un protocolo de interacción (IPs), el mantenimiento de la librería y los criterios para incluir nuevos IP[22]. El alcance de esta especificación contempla además, una descripción de cómo expresar gráficamente y cómo han de ser interpretados los protocolos de interacción mediante AUML[23] más, la lista de los protocolos de interacción (IPs) definidos por FIPA.

Parámetros del objeto Envelope de un mensaje FIPA-ACL

Parámetro	*Descripción*
`:to`	Si no existe valor asociado a *intended-receiver* se utiliza para generar dicho valor.
`:from`	Puede utilizarse como parámetro obligatorio. En tal caso el ACC devuelve un error si no se aporta un nombre de agente.
`:comments`	Campo de comentarios del *Envelope*.
`:acl-representation`	Esta información es para el recipiente final utilizado en el transporte del mensaje.
`:payload-length`	Utilizado por el ACC para eficiencia en el transporte.
`:payload-encoding`	Esta información es para el recipiente final utilizado en el transporte del mensaje.
`:date`	Esta información es para el recipiente final utilizado en el transporte del mensaje.
`:intended-receiver`	Este parámetro permite conocer el destino del mensaje.
`:received`	Se incorpora un nuevo parámetro *received* cada vez que el mensaje pasa a través de un ACC.
`:transport-behaviour`	Reservado.

Tabla 3.3: Campos en el Envelope de un mensaje ACL [FIPA00067]

Los requisitos de la especificación [FIPA00025] para que un *protocolo de interacción* cumpla la normativa son: aportar mecanismos y patrones para comprobar la interacción con agentes, facilitar la reutilización de otros IP ya propuestos y exponer mediante notación AUML el nuevo *protocolo de interacción*.

[21] IPL: acrónimo de Interaction Protocol Libray.
[22] IP: acrónimo de Interaction Protocol.
[23] AUML: Agent Unified Modeling Language.

3.2.5 Protocolos de Interacción

Durante una conversación entre agentes es habitual encontrar pautas que se repiten asiduamente. Son patrones que repiten la misma secuencia de intercambio de mensajes y con el mismo tipo de actos comunicativos (*performative*) en cada uno de los mensajes intercambiados en cada estadio del diálogo. Esos patrones de intercambio de mensajes son conocidos como *protocolos de interacción* o IPs. En el proceso de diseño de agentes es posible elegir el grado de atención mostrado por el agente para analizar; el significado del contenido del mensaje recibido, los objetivos perseguidos por el agente emisor, o el estado interno de creencias de agente emisor. Al situar todos estos procesos, dentro una secuencia planificada de emisiones y recepciones de mensajes, se logra que el agente exponga la conducta esperada.

Desde la visión de FIPA, un agente debe tener la capacidad de verse envuelto y mantener múltiples diálogos con otros agentes simultáneamente. A cada uno de esos diálogos pautados o protocolos de interacción, FIPA les asigna el término *conversación* y agrupa de esa forma las secuencias de diálogos idénticos bajo el modelo conversación y establece que; «todos los mensajes emitidos y recibidos que se encuentran bajo el control de un *protocolo de interacción* concreto, hacen referencia exclusivamente a una conversación».

3.3 Sistemas multiagente con JADE

Indica *Fabio Bellifemine* en [Bellifemine et al., 2001] que, la plataforma JADE aportó en 2001 un conjunto de herramientas y metodologías en un momento de carencia grave de utilidades disponibles para construir aplicaciones orientadas a agentes. Además, JADE incorporaba la capacidad de integrar esas nuevas aplicaciones con sistemas heterogéneos a través del bus de comunicaciones MTP. Hoy, la plataforma JADE (**JADE**, *Java Agent Development Environment*), es un reconocido marco de software para construir sistemas orientados a agentes y para gestionar la transmisión de mensajes hacia recursos disponibles a través de la red y donde se ejecutan, de manera conforme a las especificaciones de FIPA, facilidades adicionales para la conexión con otros sistemas de agentes inteligentes interoperables. Prosigue indicando *Bellifemine* que JADE, puede ser considerado también un *agente middleware* que ha implementado de una manera eficiente a una plataforma de agentes (AP) motivo por el que, a su vez, también puede ser empleado como herramienta para el desarrollo de sistemas multiagente. *Bellifemine* concluye con la enumeración de las 8 características principales:

1. *Posee tres agentes fundamentales*: el agente para la gestión del sistema (**AMS**, *Agent for Mananagement System*), el agente que presta el servicio de directorio (**DF**, *Directory Facilitator*) y el agente de comunicaciones (**ACC**, *Agent Communication Channel*). JADE activa de forma automática estos tres agentes (Figura 3.7, pág. 45).

2. *Es un sistema distribuido*: la plataforma puede ser fragmentada entre distintos servidores. Normalmente, sólo se ejecuta una instancia de la máquina virtual de *Java* dedicada a JADE en cada servidor y cada agente ordinario emplea sólo un *hilo de ejecución* de dicho proceso principal para su puesta en Servicio. Del mismo modo, sólo se utilizan Eventos de *Java* como método eficiente y de bajo consumo de recursos, para mantener

las comunicaciones entre agentes que se encuentran en un mismo servidor. Cada agente puede ejecutar tareas en paralelo y la plataforma JADE es responsable de organizar estas tareas de manera cooperativa.

3. *Existen múltiples Agentes-DF*: en JADE pueden ser ejecutados de manera redundante, múltiples instancias de Agente-DF, para obtener escenarios coherentes en entornos multi-dominio donde, un dominio, corresponde con un conjunto lógico de agentes. Todos los servicios asociados a dichas agrupaciones lógicas de agentes [FIPA00023] son expuestos a través de los Agentes DF.

4. *Declara una interface de programación avanzada* (*API*): en Lenguaje *Java* para *enviar y recibir mensajes desde y hacia otros agentes*. Los mensajes en Lenguaje ACL [FIPA00008] están representados como objetos estándar de *Java*.

5. Es *conforme con el estándar FIPA asociado al protocolo IIOP*, para interconectar diferentes plataformas multi-agente [FIPA00075].

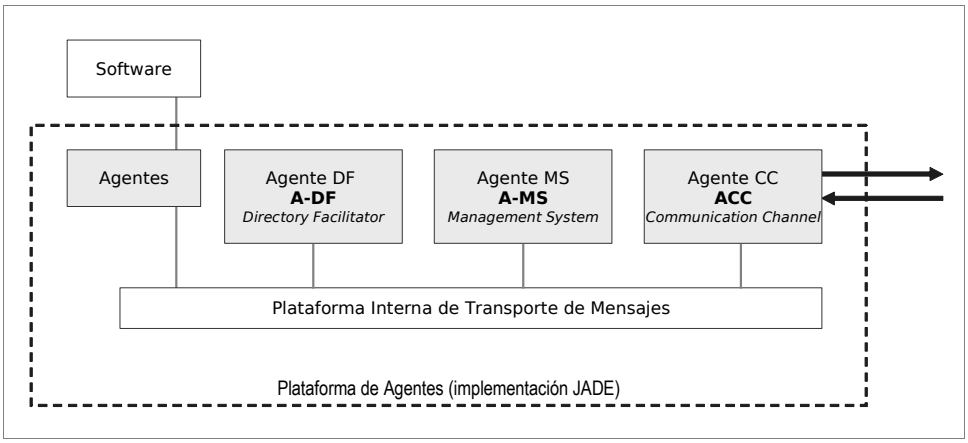


Figura 3.7: Implementación JADE del modelo de referencia FIPA

6. Aporta un *Modelo interno de transmisión de mensajes*. JADE aporta un subsistema de envío y recepción de mensajes internos de bajo consumo para transmitir los mensajes en ACL para agentes que se encuentran dentro de la misma plataforma pues, esos mensajes son transmitidos y codificados como objetos *Java* y, no sólo como cadenas, evitando de esta forma procedimientos de encriptación y desencriptación de contenidos en origen y destino de mensajes.

7. Aporta *Protocolos de interacción* implementados en una librería *Java* lista para ser utilizada. Estos protocolos se recopilan en [FIPA00025].

8. Presenta una *Interfaz Grafica de Usuario*. JADE aporta una interfaz gráfica para gestionar visualmente los agentes desde un único punto de acceso. Aporta para ello el agente **RMA**, (*Remote Monitoring Agent*). La actividad de la plataforma es monitorizada y auditada mediante emisión de LOGs generados continuamente hacia la salida estándar del sistema operativo (o hacia archivos). El *ciclo de vida* de un agente y las operaciones que

se efectúen sobre dicho agente (creación, cambio de estado, transporte, finalización del agente, etc.), pueden ser gestionadas a través de la interfaz gráfica del agente **RMA**, que fue creada para facilitar la administración de los agentes de la plataforma JADE.

Como indicaban los autores en [Bellifemine et al., 2001], JADE puede ser visto desde dos ángulos. El primero ofrece una visión de un sistema de ejecución de agentes que son conformes con las especificaciones propuestas por FIPA (tipo de transmisión de los mensajes, lenguaje en el que están expresados, ciclo de vida del agente, etc.). El segundo es una visión de la plataforma JADE como herramienta de construcción de nuevas aplicaciones orientadas a agentes; generando y permitiendo comprobar el desarrollo de dichos agentes que se utilizarán sobre la misma plataforma o sobre plataformas de agentes externas conformes con FIPA.

3.3.1 Arquitectura del sistema de ejecución

Una plataforma de agentes en ejecución aporta un conjunto de servicios a los agentes que se despliegan sobre ella. Estos servicios caen dentro de dos grandes categorías: *gestión del mensaje* y *gestión del agente*. En la primera categoría se modelan las recomendaciones de FIPA correspondientes con la estructura y codificación en ACL, el Lenguaje del contenido, la librería de actos comunicativos y los Protocolos de interacción. En la segunda categoría caen las especificaciones sobre Transporte del mensaje, Servicios de suscripción al directorio de páginas amarillas y la propia Gestión del agente desde donde se gobierna el ciclo de vida de todos los agentes activos de la plataforma.

En JADE, los agentes se encuentran confinados dentro de *contenedores*. Cada uno de esos *contenedores* se ejecuta dentro de una máquina virtual *Java* independiente. La comunicación entre contenedores utiliza *Java* **RMI** (Java *Remote Method Invocation*). Cada uno de los *contenedores* puede también actuar como un Cliente IIOP con capacidad de reencaminar los mensajes en hacia contenedores remotos en plataformas multiagente externas.

La librería de *Java* **RMI** es utilizada como la capa *middleware* en las comunicaciones entre *contenedores* y es la responsable de producir ese efecto de *plataforma-única* que expone JADE, aunque ejecute en múltiples servidores simultáneamente. Los *contenedores* son a su vez agentes-contenedores y exportan una interface remota RMI que aporta la infraestructura necesaria para realizar todas las operaciones de la plataforma JADE, como por ejemplo la gestión del *ciclo de vida* del *agente*. Dado el caso de requerir un cambio de estado en un *agente* desde un estado *suspendido* hasta el estado *activo* (Figura 3.1, pág. 33), es obligatorio que sea el agente AMS de JADE con los privilegios máximos, la entidad de software que realiza dicha reactivación en los agentes.

Cuando el agente AMS recibe esas solicitudes reconoce el *contendor* donde está ubicado el *agente* en estado *suspendido* que debe *reactivar*. Ese modelo es operativo porque el agente AMS contiene en su interior el *Servicio de páginas blancas* donde se encuentra el listado completo de nombres y ubicaciones de todos los agentes de la plataforma con sus estados en tiempo real. Si el *agente* se encuentra en el *contenedor principal* entonces, con una llamada simple de la función de reactivación, cambiará su estado. Pero, si el agente se encuentra en un *contenedor* remoto, el AMS utiliza funcionalidades de la librería *Java* RMI para realizar

una llamada remota al *contenedor* de dicho agente, ejecutando la función de reactivación el *contenedor remoto* en nombre del agente AMS.

[Elaboración propia]

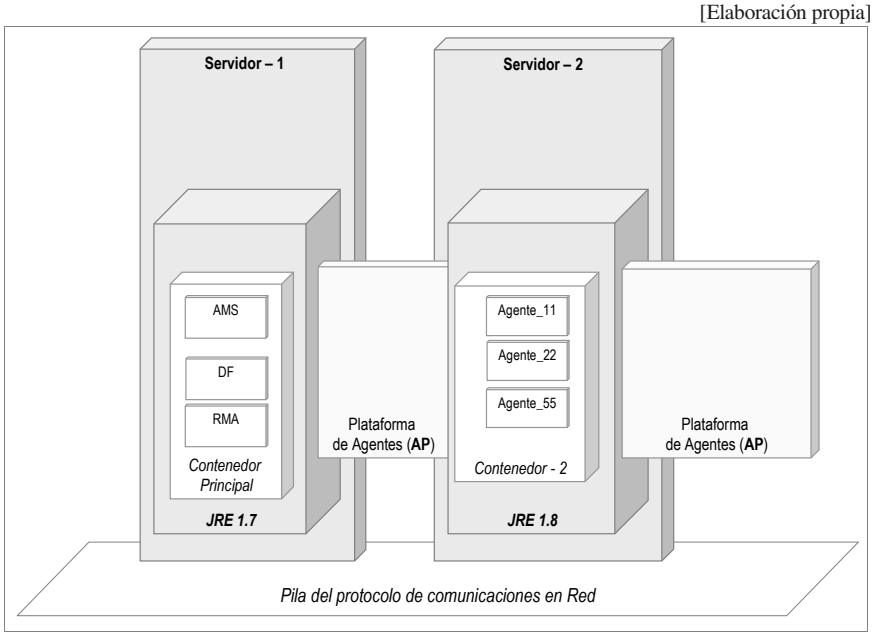


Figura 3.8: Arquitectura modular de la plataforma JADE

3.3.2 Una plataforma distribuida

Desde el punto de vista del usuario la plataforma JADE aparece como una aplicación simple, sin embargo, desde el punto de vista del desarrollador de aplicaciones se trata de un sistema distribuido que puede ser desplegado sobre distintos servidores interconectados. El primero de esos servidores (Figura 3.8, etiqueta: 'Servidor-1') juega un papel especial. En dicho servidor se encuentra el *Main-container* que acapara las responsabilidades del mantenimiento de las comunicaciones RMI-IIOP[24] (Java *Remote Method Invocation Internet Inter-ORB Protocol*) con otras plataformas externas. La plataforma JADE puede desplegar uno o varios *contenedores* pero, en todos los casos, los contenedores se ejecutan en una instancia de máquina virtual de *Java* diferente. Cada contenedor ofrece un espacio donde se despliegan y ejecutan los agentes haciendo uso de los recursos físicos del sistema.

La tecnología *Java* RMI es utilizada por JADE para establecer la comunicación entre *contenedores* y convierte a cada *contenedor remoto* en un Cliente RMI-IIOP al que es posible re-

[24] **RMI-IIOP**: Con la Librería de Java RMI (Java *Remote Method Invocation*) las aplicaciones pueden acceder a objetos remotos mediante el protocolo de Internet denominado IIOP (*Internet Inter-ORB Protocol*). Si además, se tienen en cuenta un pequeño número de restricciones en la implementación de las aplicaciones que utilizan la Librería Java RMI, entonces los clientes CORBA (*Common Object Request Broker Architecture*) construidos en cualquier lenguaje de programación, también pueden acceder a servicios tipo RMI-IIOP.

mitir mensajes y objetos CORBA[25] (*Common Object Request Broker Architecture*) completos. Cuando un contenedor remoto transmite mensajes hacia el *Main-container*, ambos se interpretan como plataformas externas que tratan de comunicar con JADE. A su vez, el contenedor principal *Main-container* actúa como servidor RMI-IIOP que permanece a la escucha permanentemente. Cuando el *Main-container* recibe las solicitudes desde otros *contenedores remotos* de la plataforma JADE, actúa como un centro de recepción y emisión hacia otras plataformas externas, empleando en ese proceso al *agente ACC* para establecer dichos enlaces con las plataformas externas.

3.3.3 Subsistemas de envío de mensajes

La arquitectura recomendada por FIPA para el desarrollo de plataformas multiagente es P2P o de igual a igual. Propone además FIPA, un contexto de comunicaciones adicional destinado al envío y la recepción de mensajes múltiples estableciendo, mediante el uso de *protocolos de interacción*, el modelo de conversación entre los agentes.

Para implementar estos requisitos, la plataforma JADE emplea tecnologías *Java* RMI-IIOP, objetos CORBA y el *Modelo de Gestión de Eventos*. Pero, como denota *Giovanni Rimassa* [Rimassa, 2003] era evidente el vacío existente entre los mecanismos empleados para el paso de mensajes empleados de manera exclusiva por los agentes que hacen uso del Lenguaje ACL; y las soluciones necesarias para atender las *solicitudes y respuestas* de comunicaciones entre *objetos distribuidos* que requieren hacer uso de RMI-IIOP y de objetos CORBA. Y fue «...por ese motivo, por el que los *agentes ordinarios* de JADE no son objetos distribuidos mientras que, los *agentes-contenedores* sí lo son».

Concluye *Fabio Bellifemine* [Bellifemine et al., 2010] en relación a estos problemas citados y a las soluciones que fueron implementadas, que JADE posee tres mecanismos de transporte para realizar la transmisión de mensajes y que utiliza la localización, del emisor y del destinatario, como factores para decidir qué mecanismo ejecutará finalmente el proceso:

(1º) Si el agente destinatario y el emisor se encuentran en el mismo *contenedor*, se utiliza el *Modelo de Gestión de Eventos* que supondrá el mínimo consumo de recursos.

(2º) Si el agente destinatario se encuentra sobre un contenedor diferente al agente emisor del mensaje pero ambos, pertenecen a la misma plataforma JADE, se utiliza RMI-IIOP y el coste de la transmisión es igual a la suma de la serialización del mensaje en la CPU del emisor, el transporte, la des-serialización en la CPU en el destinatario y el procedimiento RMI-IIOP de llamada al contenedor del destinatario.

(3º) Si el destinatario y el emisor se encuentran en plataformas multiagente diferentes, se utiliza CORBA y RMI-IIOP. El coste entonces se calcula como la suma de los proce-

---

[25] **CORBA**: Common Object Request Broker Architecture (CORBA) Estándar definido por el Object Management Group (OMG). Permite el desarrollo de aplicaciones distribuidas e interoperables. Un objeto CORBA es aquel que cumple este estándar.

sos de serialización del mensaje ACL para su transmisión como objeto CORBA hacia la localización remota, más el coste de recepción del mismo mediante RMI-IIOP cuando el receptor es el *contenedor* de la plataforma JADE y, finalmente, la reorganización y reestructuración del mensaje al formato ACL.

Esos tres mecanismos se ven siempre precedidos del proceso de localización del *agente destinatario*, proceso que es llevado a cabo mediante consultas a la *tabla local de localizaciones de agentes* del *contenedor*. Esa *tabla* se llama **LADT** (*Local-Agent Descriptor Table*) (*véase* Figura 3.9, pág.49). Cuando se inicia el proceso de transmisión de un mensaje ACL, se comprueba la localización del *agente destinatario* en la **LADT** perteneciente al *contenedor* del agente emisor del mensaje y, solamente si no se localiza al agente destinatario, se lanza una consulta dirigida a la tabla global de localizaciones **GADT** (*Global-Agent Descriptor Table*) ubicada y gestionada en el *Main-Container*; tabla que se apoya además, en la tabla de nombres de los contenedores que conforman la plataforma JADE llamada **CT** (*Container Table*).

Como ese proceso continuo de localización de agentes podría convertirse en un cuello de botella para el rendimiento de la plataforma JADE, las respuestas obtenidas tras las consultas a la tabla global **GADT** del *Main-Container*, se almacenan por el propio *contenedor* del agente emisor del mensaje ACL, en una tabla local que le sirve de *caché* llamada ***GADT-caché***. De esa forma, se evitan continuas consultas de localización de *agentes destinatarios* al contenedor principal.

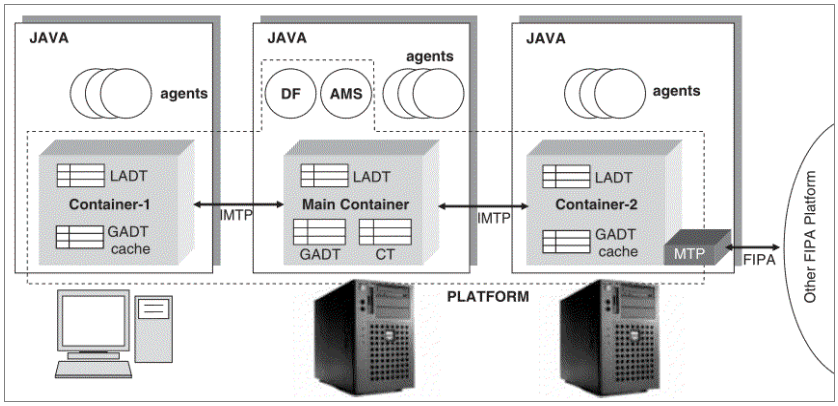


Figura 3.9: Relación entre elementos arquitectónicos [Bellifemine et al., 2010]

3.4 Resumen

- FIPA no emite especificaciones sobre el modo de crear agentes. Mantiene así, uno de los principios fundamentales establecidos sobre interoperabilidad que establece que: «...la interoperabilidad debe ser garantizada sólo mediante las capacidades de comunicación del agente inteligente al utilizar: mensajes ACL, protocolos de actos comunicativos, el lenguaje de contenidos SL y ontologías».

- Una importante decisión de diseño adoptada por la plataforma JADE consiste en desplegar y ejecutar a los agentes dentro de su *contenedor* y emplear, un único objeto *contenedor* de agentes, por cada instancia de *máquina virtual Java* en ejecución.

- En JADE los *contenedores* son clientes RMI-IIOP; poseen caché de direcciones en la tabla GADT-caché, poseen una tabla LADT de los agentes del propio contenedor y permiten gestionar a la plataforma JADE la capacidad intrínseca de movilidad (de los *agentes móviles*) entre contenedores. Para JADE, los *contenedores* conforman plataformas externas de agentes sobre las que pueden intervenir con llamadas *Java* RMI.

- Las características de implementación de la plataforma JADE son tratadas con mayor profundidad en [Poggi et al., 2004], [Poggi et al., 2007], [Beneventi et al., 2004], [Poggi & Tomaiuolo, 2011] y, la construcción de ontologías, en [Caire & Cabanillas, 2010].

4 Gestión de la concurrencia y de los comportamientos

La vida es lo que sucede,
mientras estás ocupado haciendo otros planes.

John Lennon

Objetivos del capítulo:

- *Presentar el mecanismo de reparto de tiempo utilizado por JADE.*
- *Resumir los comportamientos empleados en la implementación de las técnicas.*

4.1 Introducción

Cierra este cuarto capítulo el *estado del arte* de las tecnologías involucradas en la presente tesis de investigación. En el apartado 4.2 se recopilan en esta ocasión las soluciones que aporta la plataforma JADE para definir *hilos de ejecución* y *grupos* de hilos de ejecución utilizados para confinar los procesos y establecer prioridades en los agentes. Se incluye además, una síntesis de los criterios de diseño empleados por JADE para resolver el problema de concurrencia de tareas ejecutadas por los agentes y finaliza el capítulo, en el apartado 4.3, con una enumeración de los comportamientos empleados posteriormente para modelar la conducta de los *agentes* implementados.

4.2 Sistema JADE en tiempo de ejecución

El agente que realiza la gestión integral de la plataforma JADE se denomina *agente* **AMS**[26] y, el agente destinado a exponer la colección de *Servicios* en la plataforma se denomina *agente* **DF**[27]. Ambos son generados en el instante de arranque de la plataforma JADE cuando se invoca con `jade.Boot` desde línea de comandos o bien, mediante un objeto `jade.Boot3` desde el

[26] ***AMS***: Agent Management System. Agente principal que ejerce la gestión de todos los agentes desplegados sobre la plataforma JADE.

[27] ***DF***: Agent Directory Facilitator. Agente de JADE que actúa como simulador del *servicio de directorio de localización de capacidades* de los agentes desplegados y en tiempo de ejecución en la plataforma JADE. Conocido como Servicio de Páginas Amarillas.

interior de una aplicación desarrollada en *Java*. Sólo cuando se emplea el parámetro `-gui` se incorpora un tercer agente con el nombre RMA (*Remote Monitoring Agent*) que facilita al *actor-humano* la gestión remota de agentes desde su interfaz gráfica. El *agente AMS* es único en la plataforma y asume la responsabilidad de gestionar los agentes, por lo que su nivel de privilegios es el más elevado en la plataforma. Por contra, los *agentes DF* no poseen tales privilegios pero sí, una base de conocimientos útil que mantiene activo el *Servicio* de *páginas amarillas* en la plataforma. Esos agentes DF permiten además, iniciar instancias de si mismos para realizar federaciones y adoptar la especificación [FIPA00023] como se refleja en su Apdo. 4º, denominado: *servicios de gestión de agentes*.

Giovanni Rimassa [Rimassa, 2003] destaca que una de las características obligatorias que fueron implementadas para alcanzar el estándar FIPA, y que se convirtió en piedra angular durante la construcción de JADE, fue su capacidad para construir y transmitir Mensajes ACL, en concreto la especificación [FIPA00067] dedicada a describir el *servicio de transporte de mensajes* MTS (*Message Transport Service*) y la especificación [FIPA00075] dedicada al protocolo de transporte de mensajes mediante RMI-IIOP. Ambas exigían la representación de datos en formato XML y capacidades de transmisión sobre protocolo HTTP o, utilizar *cadenas de texto* para serializar mensajes como objetos CORBA y remitir mediante RMI-IIOP.

4.2.1 Plataforma distribuida de agentes

Continúa *Rimassa* [Rimassa, 2003] estableciendo que, una característica diferenciadora de JADE de otras implementaciones de plataformas de agentes es su capacidad para coordinar máquinas virtuales de *Java* en ejecución sobre distintos servidores mediante *contenedores*. Dentro de cada *contenedor* se ejecutan cero o más agentes. La plataforma JADE se compone de uno o más *contenedores* y los agentes AMS y DF se ejecutan obligatoriamente dentro del contenedor denominado *Main-Container*. Desde ese *contenedor* se gestionan todas las instancias de máquinas virtuales *Java* unidas través de sus *contenedores* remotos mediante Java RMI, mostrando la clásica visión de plataforma continua[28] que ofrece JADE al usuario desde la interfaz gráfica del agente RMA.

Las funciones de gestión del AMS son conformes a la especificación [FIPA00023]. Esas funciones son: *creación* de nuevos *agentes*, *eliminación* de *agentes* y *seguimiento* del proceso de *migración* de *agentes*. Se suman: mantenimiento del ciclo de vida de cada *agente* (*ver* Figura 3.1, pág. 33), capacidad de *registrar* a cada nuevo agente con las propiedades de la plataforma, eliminación de los privilegios para extraer al agente de la plataforma, modificación de las descripciones del agente y la capacidad de *búsqueda* de agentes mediante su identificador. A esos mecanismos de gestión y control del *agente* se incorporan; las funciones del *Agente DF* para soportar la *suscripción* de agentes en servicios descritos por el *directorio de localización de servicios*. Ese mecanismo corresponde a la especificación [FIPA00035]. Con ese

[28] ***JADE In-Process***: Cada instancia de *contenedor* ocupa por completo la máquina *Java*. Como concluye *Rimassa*, esta última afirmación no siempre es cierta pues se cuenta con la interface de JADE *In-Process* que facilita a aplicaciones externas iniciar contenedores de JADE embebidos en procesos internos para conectar aplicaciones que se remiten mensajes.

modelo de nomenclatura de *cuentas únicas* se cumplen las recomendaciones de FIPA sobre autonomía exigida a los agentes. La solución que aporta JADE es una implementación mediante *objetos Activos* de *Java* para cada agente en la plataforma.

4.2.2 Transiciones de estado en el ciclo de vida del *agente*

Para *Giovanni Rimassa* [Rimassa, 2003] las transiciones de *estado* de un *agente* en JADE pueden ser ejecutadas o bien por la autoridad de la plataforma (el agente AMS) o bien, y sólo en ciertos casos concretos, por el propio *agente* (véase, Tabla 4.1). Con ese reparto de competencias se consigue el balance entre autonomía y capacidad de control de la plataforma sobre dicho *agente*.

Resumen de transiciones y autorizaciones

Estados origen	Transición	Estado destino	Método	Posee privilegios:	
				AMS	El agente
Desconocido	Crear	*Inicializado*	`new jade.core.Agent`	•	
Inicializado	Invocar	*Activado*	`setup()`	•	
Activado o Esperando	Suspender	*Suspendido*	`doSuspend()`	•	•
Suspendido o Esperando	Reactivar	*Activado*	`doActivate()`	•	
Activado	Esperar	*Esperando*	`doWait()`	•	•
Esperando	Despertar	*Activado*	`doWake()`	•	
Activado	Mover	*En tránsito*	`doMove( Location )`	•	•
Activado o Esperando o Suspendido	Destruir	*Desconocido*	`doDelete()`	•	•*
Activado	Clonar	*Desconocido*	`doClone()`	•	

Tabla 4.1: Privilegios necesarios para modificar el estado de agentes en JADE

4.2.3 Hilo de ejecución de un agente sobre JADE

Para *Doug Lea* [Lea, 1999] una solución que permite categorizar el extenso campo de posibilidades que ofrecen los *Threads*[29] consiste en considerar cuál es la *intención del programador* durante la declaración del *Thread*.

```
01 |      new Thread( algun_objeto_Runnable ).start();
```

Código 4.1: Instancia de Thread según [Lea, 1999]

En la sentencia del Código 4.1, denota *Doug Lea* que sería válido interpretar: (**1**) que el programador desea invocar el método `run()` para que se lance sobre un nuevo espacio de ejecución, que finalizará tan pronto devuelva el control `run()` al *hilo de ejecución* temporal que lo ha invocado o, (**2**) que el programador trata de crear un nuevo *hilo de ejecución* y sobre di-

[29] ***Thread***: En programación, el *Thread* es la secuencia de pasos que se ejecutan de uno en uno y que constituyen el hilo de ejecución del programa. En programación orientada a objetos, un *Thread* es un objeto que permite crear un nuevo hilo de ejecución aislado [Arnold et al., 1996].

cho hilo un objeto que iniciará su método `run()`. Por tal motivo, la clasificación de los tipos de *Hilos de Ejecución* puede estar basada en utilizar la *intención del programador*, resultando los dos casos siguientes:

1. Hilo de ejecución orientado a la Tarea. En ese caso la intención es crear un *Thread* para invocar de manera asíncrona algún método y realizar así, en paralelo, una tarea específica que puede abarcar desde un proceso sencillo hasta una sesión completa de usuario. Las técnicas de invocación de hilos de ejecución *basadas en tareas* pueden soportar escenarios de paso de mensajes que escapan a las limitaciones de los procedimientos de llamada a función clásicos. Estos diseños se observan en computación-paralela, en sistemas de uso intensivo y marcos de programación orientados a eventos.

2. Hilo de ejecución orientado al Actor. En ese caso la intención para utilizar un *Thread* es crear y establecer, de manera dinámica, un *Objeto Activo* autónomo y con capacidades de procesamiento independiente, con el objetivo de permitir a ese nuevo objeto interactuar con otros *objetos activos* y reaccionar a estímulos externos. Se trata entonces de diseños *basados en actores* que se encuentran en los sistemas distribuidos, sistemas de control y en los sistemas reactivos.

Los *agentes* en la plataforma JADE son *Objetos Activos* que se ejecutan en su propio *Thread*[30], que además y por diseño particular de JADE, pertenecen al grupo de hilos de ejecución nombrado como: *"JADE User Agents"* tal y como puede apreciarse en el segmento de código extraído de la plataforma JADE, perteneciente a la clase *FullResourceManager* del paquete *jade.core* (Código 4.2, pág.55). Este grupo de *Threads* es dependiente a su vez del grupo denominado *''JADE"* (*véase línea* 59, Código 4.2), que incluye a todos los *objetos* principales de la plataforma JADE.

Cada *agente* en el instante de su creación recibe un *Thread* por parte de la autoridad de la plataforma JADE (en concreto por parte del agente AMS) y ese *Thread* es además inicializado como miembro del grupo de *Threads "JADE User Agents".* De esa forma JADE establece una jerarquía de prioridades en la ejecución de los hilos que transmite progresivamente desde el grupo principal hacia los grupos inferiores, llegando hasta el grupo de hilos en los que ejecuta a los *agentes* creados por los usuarios. Destacar que los *agentes*, al ser *objetos activos*, pueden recibir:

- *Eventos*: mediante clic de ratón, etc.
- *Notificaciones*: producidas por cambios de estado que son capturados mediante implementaciones de la interface *Listener*.
- *Posts*: al recibir mensajes.
- *Activaciones*: que les solicitan un cambio de estado por ejemplo.
- *Órdenes*: que les obligan a imprimir o, hacer otras tareas.

---

[30] ***Thread** o **Hilo de ejecución*** (nota de traducción): En lo sucesivo se utilizarán Thread e Hilo de ejecución de manera indistinta.

- *Relays*: como indica *Doug Lea* [Lea, 1999], «...mensajes remitidos al *agente* para su reencaminamiento o su despacho hacia un tercero».

```
35 | private static final String USER_AGENTS_GROUP_NAME = "JADE User Agents";
36 | private static final String SYSTEM_AGENTS_GROUP_NAME = "JADE System Agents";
37 | private static final String CRITICAL_THREADS_GROUP_NAME = "JADE Time-critical Threads";
   .../...
50 | public FullResourceManager() {
51 |    parent = new ThreadGroup("JADE") {
52 |       public void uncaughtException(Thread t, Throwable e) {
53 |          if (!terminating) {
54 |             super.uncaughtException(t, e);
55 |          }
56 |       }
57 |   };
58 |
59 | agentThreads = new ThreadGroup(parent, USER_AGENTS_GROUP_NAME);
60 | agentThreads.setMaxPriority(Thread.NORM_PRIORITY);
```

Código 4.2: Inicialización de Threads en JADE [CSELT & TILab, 2014]

4.2.4 Cola privada de mensajes del agente

Recuerda *Bellifemine* [Bellifemine et al., 2001] que para dar soporte a las capacidades sociales de los agentes JADE, se implementó en cada agente su propia *cola de mensajes* (Figura 4.1, pág.56) donde poder almacenar los mensajes recibidos desde el Servicio de Transferencia de Mensajes (MTS) de la plataforma. *Giovanni Rimassa* [Rimassa, 2003] enumera los distintos tipos de *políticas de recepción* de mensajes que pueden establecerse y que afectan de manara directa al *planificador de tareas* (*véase* Apdo. 4.3) del *agente*:

1. Política *Agent-wise Blocking*: produce un bloqueo completo del agente cuando espera a recibir nuevos mensajes, incapacita al agente para cualquier actividad y se prioriza la recepción de mensajes hasta que un mensaje sea recibido y gestionado en la cola.
2. Política *Agent-wise Blocking with timeout*: actúa de forma similar a la política anterior pero establece un límite de tiempo máximo de espera de recepción del mensaje.
3. Política *Conversation-wise Blocking*: produce un bloqueo completo del agente empleando el identificador de la conversación activa con un interlocutor como el filtro del mensaje que permitirá finalizar el estado de bloqueo del agente.
4. Política *Conversation-wise Blocking with timeout*: similar a la anterior con un valor límite de tiempo de espera a recibir el mensaje de liberación del agente.
5. Política *Polling*: o también conocida como política de *almacenamiento intermedio*. Cuando se emplea esta política no se bloquean las actividades del agente pero la cola de mensajes utiliza un espacio de memoria para el almacenamiento de mensajes que permanecen a la espera de ser tratados. Denotar que, al emplear esta política, se obtiene un valor `null` cuando se consulta la *cola de mensajes* y ésta se encuentra vacía.

Las políticas de bloqueo del agente, que son aplicables durante la recepción de mensajes, se complementan con un modelo adicional para *seleccionar* los mensajes de la cola:

1. Política de selección FIFO: que permite seleccionar los mensajes en orden natural de una cola, esto es, se gestiona en primer lugar el mensaje más antiguo en la cola.

2. *Política de selección por Pattern Matching*: que permite seleccionar de la cola sólo aquellos mensajes que cumplen con el conjunto de restricciones y condiciones de filtrado impuestas para el tratamiento de mensajes entrantes en cola. Ese conjunto de restricciones se expresa mediante un patrón de filtrado (o, *Message Template*).

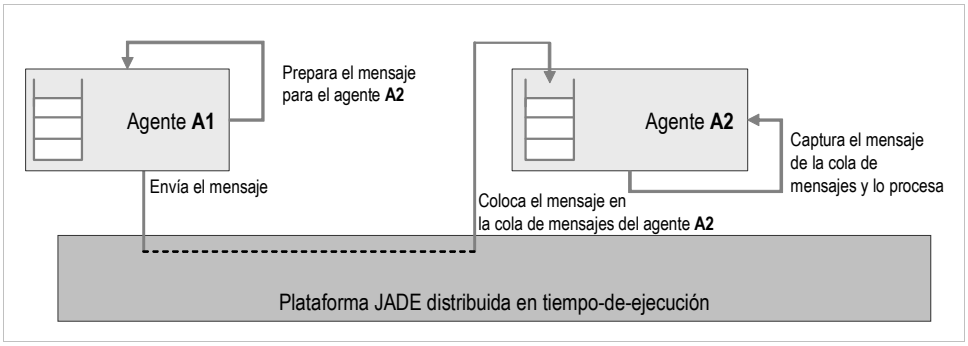


Figura 4.1: Paradigma de paso asíncrono de mensajes [Bellifemine et al., 2001]

4.2.5 Ciclos de vida de los Threads en *Java*

Para Muñoz Escoí [Escoí et al., 2013], los hilos de ejecución de los procesos en una aplicación *Java* seguirán el *ciclo de vida* similar al mostrado en la Figura 4.2:

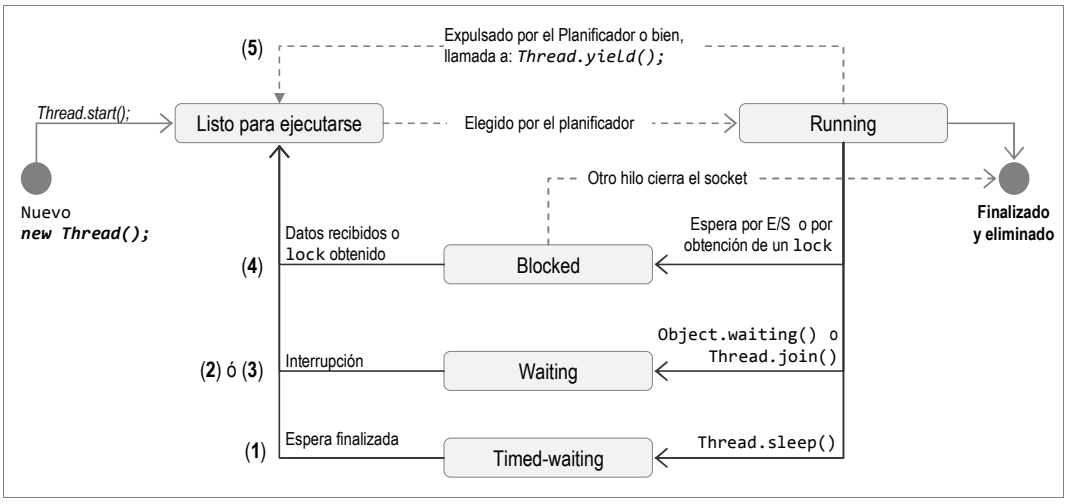


Figura 4.2: Ciclo de vida de los hilos de ejecución en Java [Escoí et al., 2013]

El hilo es creado mediante `new Thread()` y el cambio hacia el estado denominado *'Listo para ejecutarse'* se alcanza con su método `start()`. Sin embargo, como se aprecia en la Figura 4.2, es posible retornar al estado *'Listo para ejecutarse'* desde distintas situaciones: (**1**) cuando ha finalizado un tiempo de inactividad fijado por el método `sleep()`, (**2**) por interrupciones desde llamadas al método `join()` o (**3**) por recibir la llamada de el método `waiting()`, (**4**) por la incapacidad de obtener un bloqueo de un recurso físico `lock`, o porque (**5**) se ha invocado el método `yield()` y es expulsado del planificador de Hilos de *Java.* En el estado de ejecución *Running* el hilo procesa sus instrucciones por alguno de los procesadores y abandona este estado cuando haya concluido todas sus instrucciones.

4.3 Gestión de la concurrencia en el *agente*

Establece *Giovanni Rimassa* [Rimassa, 2003] que en JADE se adoptó, como decisión de diseño, asociar un único *Thread* a cada *agente* y ejecutar sobre ese *Thread*, todas las tareas necesarias para modelar su conducta y llevar a cabo las conversaciones mantenidas de manera simultánea con otros interlocutores. Esa decisión era opuesta al modelo de asignación de un *Thread* por cada *conversación* pero, en JADE, la economía en el consumo de recursos fue primordial y, como es necesario que un agente mantenga varias líneas de comunicación de manera simultánea «...la solución aportada para gestionar la concurrencia tareas en el agente fue *el modelo jerárquico de gestión de comportamientos*».

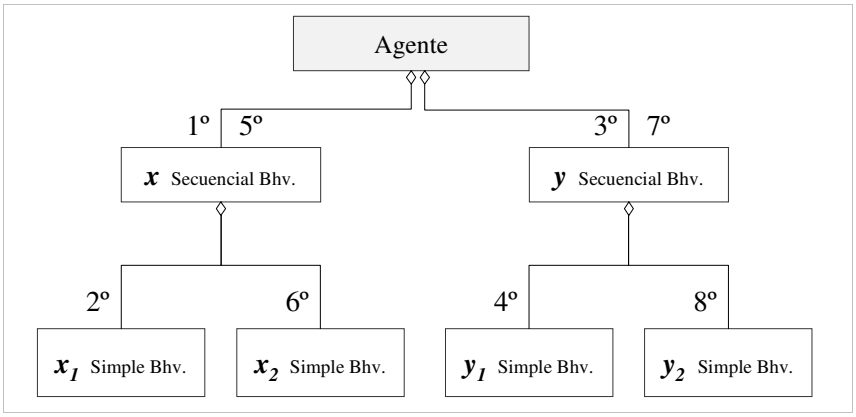


Figura 4.3: Modelo jerárquico de reparto de tareas en JADE [Rimassa, 2003]

Los objetos *comportamiento* permiten al *ingeniero de software*, encapsular los procesos y tareas que ha de realizar un *agente*. Esos objetos puede ser incorporados, activados o eliminados de la definición del *agente software* pero, no es posible forzar la ejecución de un *comportamiento* en un instante concreto porque, esa decisión es adoptada por un *planificador de tareas* interno que analiza el conjunto de objetos comportamiento *activos* en el agente y los gestiona y ejecuta en función de un modelo jerárquico de reparto equitativo de tiempos, también conocido como gestión de concurrencia de tareas tipo *Round-Robin* o multitarea *cooperativa*. El *planificador de tareas* (*Scheduler*) no asigna prioridad alguna a las tareas, sólo gestiona la entrega y recuperación del *hilo de ejecución* a cada comportamiento, permitiendo la ejecución de varias *conversaciones* simultáneas mediante un único *Thread*. Además, permite al ingeniero «...construir *árboles jerárquicos* de *comportamientos* donde cada nodo actúa como un *planificador de tareas* de los objetos inferiores» [Rimassa, 2003].

La Figura 4.3, representa la secuencia de ejecución generada por el *planificador de tareas* sobre dos comportamientos complejos que contienen, a su vez, comportamientos subordinados. Los comportamientos se han denominado x e y, y los subcomportamientos se han nombrado como: x_1, x_2, y_1 e y_2 respectivamente. Con la aproximación implementada por JADE, el *planificador de tareas* entrega el control a x y luego a y alternativamente. Cuando x toma el control ejecuta x_1 liberando a continuación el control que se entrega a y, que a su vez lo entrega de manera inmediata a y_1. Una vez finalizado, y_1 devuelve en control para comenzar de nuevo x, el cual delega el control a x_2. Al final, la secuencia que resulta de la explotación del modelo de *Round-Robin* es la siguiente $\{ x.x_1, y.y_1, x.x_2, y.y_2 \}$ [Rimassa, 2003]. Destacar

que, la multitarea *cooperativa* implica que los comportamientos no se ejecutan hasta que el *comportamiento* previo finaliza y devuelve el control, del *hilo de ejecución*, al *planificador*.

4.3.1 Comportamientos básicos

Un comportamiento se considera básico [Bellifemine et al., 2007], cuando hereda de la clase denominada *SimpleBehaviour* y se ejecuta de forma monolítica sin posibilidad de detener su ejecución (*véase* Figura 4.4, pág. 60). La tarea que debe ser acometida, se modela en el interior del método `action()` del objeto comportamiento. Una vez finalizado `action()`, se evalúa el método `done()` y, si devuelve (*true*), es entonces cuando el comportamiento habrá finalizado definitivamente. Si `done()` devuelve (*falso*) el comportamiento sigue formando parte del *pool* de comportamientos a la espera de una nueva llamada del *planificador de tareas* para ser ejecutado. Existen 8 comportamientos básicos: *AMSSubscriber*, *MsgReceiver*, *SimpleAchieveREInitiator*, *SimpleAchieveREResponder*, a los que se suman los comportamientos más habituales: *OneShotBehaviour*, *CyclicBehaviour*, *TickerBehaviour* y *WakerBehaviour*.

- *OneShotBehaviour*, está diseñado para completarse íntegramente y salir del *planificador de tareas* del agente una vez que ha finalizado. Esos objetos tienen implementado el método `done()` para que siempre devuelva (*true*). Como indica *Bellifemine*, «...ese método puede ser reescrito en la misma declaración de la instancia del comportamiento para obtener objetos *oneShotBehaviour* especiales».

- *CyclicBehaviour*, está diseñado para retornar al *pool* de comportamientos pendientes de ejecución tan pronto finalizan los procesos que se encuentran en su método `action()`. Implementa su método `done()` de manera que devuelve siempre (*falso*) y por ello, las tareas recogidas dentro de este comportamiento son ejecutadas ininterrumpidamente.

- *SimpleBehaviour*, es una clase abstracta de comportamiento que aporta gran flexibilidad y permite, por ejemplo, acumular en número de ejecuciones realizadas. Se recomienda implementar en el método `done()` comprobaciones sobre la ejecución de los procesos.

Todos los objetos comportamiento heredan de la clase `Behaviour` dos métodos abstractos: `onStart()` y `onEnd()` que son ejecutados antes de `action()` y tras la finalización de `done()` al devolver (*true*) instantes antes de la eliminación del comportamiento del pool del agente. Recuerda *Bellifemine* [Bellifemine et al., 2007] «...si se utiliza el método `removeBehaviour()` se eliminará el comportamiento sin ejecutar el método `onEnd()`, al tratarse de un proceso de eliminación invocado por el *agente* y no, por el *planificador de tareas* interno». Los objetos comportamiento poseen el campo *myAgent*, que mantiene una referencia al objeto *agent* al que pertenecen los comportamientos. Cuando un comportamiento ha finalizado por completo, esto es, cuando el método `done()` ha devuelto (*true)* y el método `onEnd()` ha tenido lugar, dicho objeto no vuelve a ser ejecutado nunca más y sólo es posible reincorporarlo al conjunto de comportamientos mediante su método `reset()` y con la subsiguiente incorporación mediante el método *myAgent* `.addBehaviour()` del *agente*.

4.3.2 Operaciones programadas

Establece *Bellifemine* [Bellifemine et al., 2007], cómo ejecutar tareas en instantes de tiempo programados, mediante objetos *SimpleBehaviour* de tipo disparo ininterrumpido y de tipo temporizador (Figura 4.4, pág. 60). Esos comportamientos son:

- *TickerBehaviour*, que está diseñado para ejecutar el método `onTick()` a intervalos finitos e iguales de tiempo fijados dentro del constructor del comportamiento.
- *WakerBehaviour*, diseñado para ejecutar el método `onWake()` trascurrido un tiempo finito que es fijado dentro del constructor de este objeto comportamiento.

4.3.3 Comportamientos compuestos

JADE propone la utilización del objeto *Composite* [Bellifemine et al., 2007] como mecanismo recomendado para crear comportamientos compuestos (Figura 4.4, pág. 60). *Composite*, permite anidar conjuntos de comportamientos y gestionar la ejecución de cada uno de ellos mediante un control por fases. Establece *Rimassa* que, la organización de las tareas internas realizada por los objetos *Composite*, utilizan `scheduleFirst()` y `scheduleNext()` como instrumentos para asignar tareas en una ordenación que es relativa respecto a la primera vuelta de ejecuciones, y a continuación, respecto a la segunda vuelta de las ejecuciones, etc. Estos métodos son abstractos y pueden ser definidos en las subclases de *Composite*.

Los dos comportamientos compuestos en JADE permiten ejecución en *serie* y en *paralelo* de tareas son: *SerialBehaviour* y *ParallelBehaviour*. A ambos se suma el tipo *máquina de estados finitos* (FSM, *Finite State Machine*) que extiende a *SerialBehaviour*. Las propiedades de estos comportamientos son las siguientes:

- *SequentialBehaviour*, es un comportamiento diseñado para modelar ejecuciones secuenciales de los procesos que forman una tarea. Se exige la ejecución paso a paso de cada uno de esos procesos, denominados subcomportamientos o *child*. Cuando uno de los *child* del comportamiento tipo *Sequential* finaliza, su método `done()` devuelve (*true*), y el programador de tareas del agente procede con el siguiente *child*, (cuando reciba el testigo de ejecución) y así, sucesivamente.

- *FSMBehaviour*, está diseñado para facilitar la emulación de una máquina de estados (**FSM**, *Finite State Machine*) que ejecuta a los subcomportamientos en función del valor entregado en el instante de la finalización de cada subcomportamiento. Sólo cuando se obtiene un valor concreto se produce un *cambio de estado*. Para trabajar con los subcomportamientos, se registran los estados mediante el método `registerState()` que admite una etiqueta y el nombre del comportamiento a ejecutar. Para definir las transiciones entre estados se emplea el método `registerTransition()`, que admite tres parámetros; el nombre del estado origen, el nombre del estado destino y un valor entero que establece cuándo adoptar dicha transición. Mediante los métodos `registerFirstState()` y `registerLastState()` se marcan los nombres de estado de inicio y de final de la máquina de estados finitos.
- *ParallelBehaviour*, está diseñado para programar la ejecución en paralelo de subcomportamientos. En *ParallelBehaviour* se lanza el método `action()` de uno cualquiera de los

subcomportamientos y, sin analizar su estado de finalización se lanzan, de manera inmediata, todos los demás métodos `action()` de los restantes subcomportamientos. Este tipo de comportamiento compuesto permite analizar si han finalizados todos o si ha finalizado alguno de los subcomportamientos para abortar la tarea en conjunto o bien, cuando existe un límite de tiempo que no debe ser rebasado por ninguna de las subtareas.

[Elaboración propia]

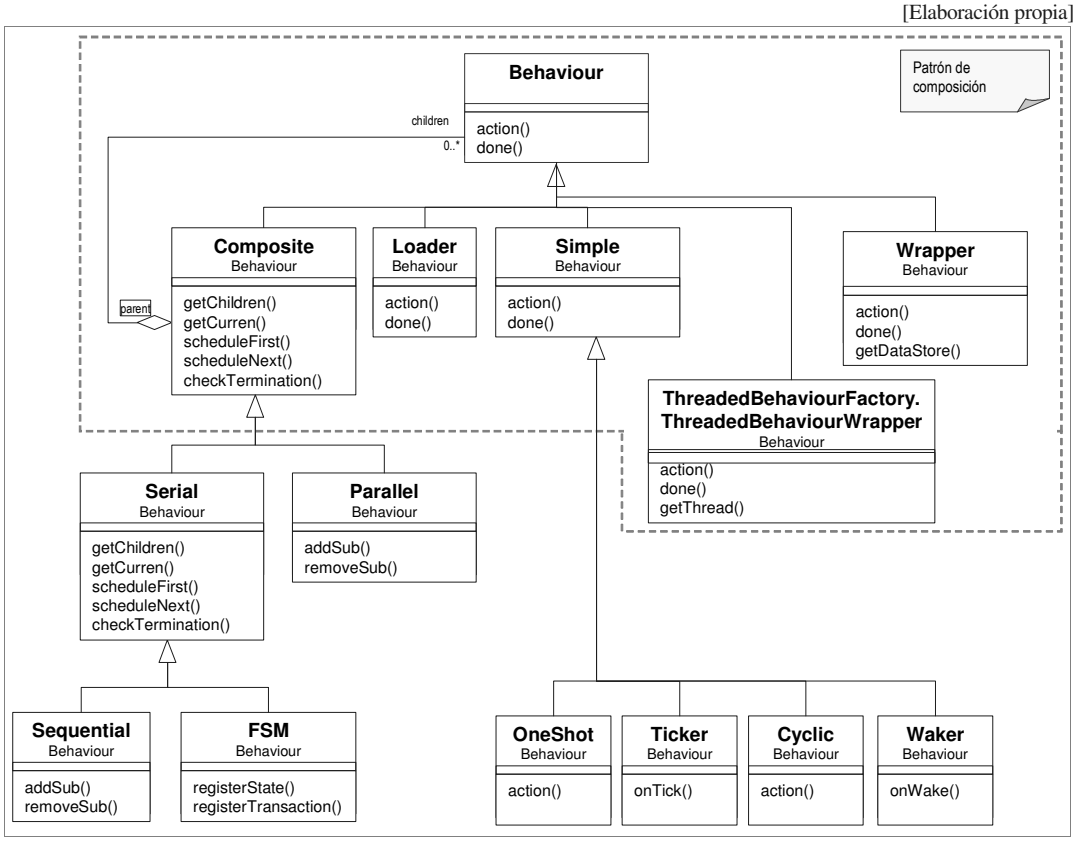


Figura 4.4: Jerarquía de comportamientos en JADE [Bellifemine et al., 2007]

4.3.4 Comportamientos especiales

Los más destacados son los siguientes: *LoaderBehaviour*, *ReciverBehaviour*, *ThreadedBehaviour* y *WrapperBehaviour*. Extienden la clase abstracta *Behaviour* y se localizan en el paquete *jade.core.behaviours.**. [Bellifemine et al., 2010].

- *ThreadedBehaviour*, está diseñado para envolver procesos o tareas prolongadas y para eliminar el posible bloqueo que generaría ese proceso en la actividad desarrollada por el *agente*. Aunque una solución sería emplear *Threads* independientes para ejecutar dichos procesos, JADE aporta esta solución elegante para modelar tareas de manera conforme con el diseño de los comportamientos de un agente.

- *LoaderBehaviour*, es un comportamiento aún más especializado que el anterior. Está destinado a realizar la carga de objetos tipo comportamiento que no se encuentran en el entorno de la máquina de *Java* donde se ejecuta el agente, esto es, dentro del alcance de las librerías accesibles que fueron definidas en la variable CLASSPATH. Este compor-

tamiento requiere utilizar el valor `ACLMessage.REQUEST`, la ontología denominada *BehaviourLoading* y, como valor del Lenguaje de codificación, LEAP[31].

- *ReceiverBehaviour*. Diseñado específicamente para recibir mensajes ACL porque contiene el método `receive()` integrado y superpuesto al método del mismo nombre que poseen los agentes de JADE. Es un comportamiento que finaliza con la recepción del mensaje y aporta el métodos `getMessage()` para el análisis del contenido del mensaje.

Indican *Bellifemine, Caire, Trucco y Rimassa* en [Bellifemine et al., 2010] que, los comportamientos envueltos dentro de instancias de la clase *ThreadedBehaviour*, deben atender a las siguientes tres recomendaciones:

1. Al utilizar el método `removeBehaviour()`, para tratar de eliminar el comportamiento del *planificador de tareas* del *agente*, no se obtendrá ningún efecto pues, es necesario tener en cuenta que los comportamiento envueltos sobre un *ThreadedBehaviour* requieren utilizar `getThread()` previamente sobre la instancia del comportamiento y, una vez capturada la referencia al *Thread*, llamar al método `interrupt()` para detener la ejecución de dicho comportamiento.

2. Cuando un *agente* se *elimina*, se *mueve* o entra en estado de *suspensión* es necesario detener explícitamente el hilo de ejecución del comportamiento envuelto, tal y como se propone en el párrafo anterior.

3. Es necesario prestar especial atención a las tareas del agente que emplean un comportamiento envuelto por un *ThreadedBehaviour* cuando acceden a recursos pues, esos recursos, pueden también estar siendo accedidos y utilizados por otros comportamientos.

4.4 Resumen

- La plataforma JADE resuelve el problema de concurrencia de tareas en los agentes con la implementación de las siguientes decisiones de diseño:

  - JADE declara *hilos de ejecución* únicos para cada *agente* y asocia, esos hilos, a un grupo de *hilos de ejecución* para subordinar la ejecución de los *agentes* al *hilo principal* que está asignado a la plataforma.

  - JADE impone una política de reparto equitativo de tiempos que no asigna prioridad a las tareas (multitarea cooperativa o programación de tareas tipo *Round-Robin*). Esto permite mantener varias conversaciones simultáneas en el agente mediante un único *hilo de ejecución*.

---

[31] ***LEAP***: En la plataforma JADE el paquete *jade.content* incorpora el Lenguaje de contenido LEAP. El Lenguaje LEAP compila las expresiones en formato binario.

- JADE aporta a los *agentes* la posibilidad de recubrir, mediante un envoltorio software (*WrapperBehaviour*), aquellos comportamientos que requieren ser ejecutados en hilos independientes del *hilo de ejecución* del agente pero, que se mantienen subordinados a *hilo de ejecución* principal asignado a la plataforma JADE.

5 Resultados de la investigación

'The promotion of technologies and interoperability specifications that facilitate the end-to-end interworking of intelligent agent systems in modern commercial and industrial settings'

FIPA Mission (1997)

Objetivos del capítulo:

- *Enumerar los instrumentos creados y describir las técnicas alcanzadas.*
- *Exponer los convenios propuestos para transformar y desplegar aplicaciones.*
- *Resumir las características de los Agentes implementados.*

5.1 Introducción

Este capítulo describe los resultados de la investigación. Está organizado de acuerdo a los siguientes apartados:

- El apartado 5.2 está dedicado a describir la solución de diseño para integrar un motor de inferencia en el agente.

- El apartado 5.3 describe la solución arquitectónica (modelo del dominio) para desplegar agentes con motor de inferencia integrado en plataformas multiagente.

- El apartado 5.4 describe una serie de utilidades software aportadas para facilitar la creación del escenario, la construcción de los agentes con motor de inferencia (nodos), el lanzamiento de estos agentes o de la propia plataforma multiagente, y la integración de *shells* de comandos del intérprete del lenguaje Java, lenguaje CLIPS, lenguaje Jess y cliente SQL.

- El apartado 5.5 describe la ontología creada para que los agentes con motor de inferencia integrado puedan interpretar el significado de los mensajes para: ejecutar sentencias en motores de inferencia remotos o locales, migrar entre contenedores de la plataforma multiagente y remitir o capturar problemas en curso de resolución hacia/desde un agente destinado a ofrecer un servicio de almacenamiento de soluciones intermedias. En este apartado se describe también un tipo de agente diseñado

para funcionar exclusivamente como entorno de desarrollo. El apartado concluye con la descripción, de manera normalizada, del protocolo de interacción del agente con su motor de inferencia y del agente que presta el servicio de almacenamiento de soluciones intermedias.

- En el apartado 5.6 se hace una descripción exhaustiva de las técnicas de software implementadas para integrar un intérprete del lenguaje Java en el agente, externalizar los comportamientos del agente, controlar la inicialización de estos comportamientos, cómo elaborar pruebas unitarias para el agente, integrar un motor de inferencia genérico en el agente, y migrar agentes entre contenedores.

- En el apartado 5.7 se encuentran las propuestas de agentes implementados y los convenios para implementar el modelo arquitectónico diseñado así como los diferentes tipos de agente involucrados.

- El apartado 5.8 recoge una síntesis de los factores críticos determinantes del éxito de la investigación.

- Finalmente, el apartado 5.9 es una comparativa de la solución propuesta con soluciones de integración existentes de agentes JADE con motores de inferencia.

La Tabla 5.1, (*véase*, pág.65) expone la clasificación de los productos generados por cada una de las *líneas de trabajo* que han compuesto la investigación. Ese modelo de gestión de proyectos, basado en *líneas de trabajo*, ha permitido una flexibilidad mayor respecto a las posibilidades que ofrece el modelo de planificación convencional por fases secuenciadas ya que, no se sabía *a priori*, cuáles serían las consecuencias de las soluciones encontradas en los artefactos de software ni tampoco, cómo se desviarían las estimaciones de tiempo/esfuerzo necesarias para culminar cada implementación. En resumen, la planificación quedó organizada en cuatro líneas de trabajo que avanzaban de forma asíncrona y comprometidas con numerosos *test-unitarios* para evitar la aparición de situaciones que pudieran bloquear el desarrollo final de los agentes.

Esas *líneas de trabajo* fueron identificadas con los siguientes nombres: Línea de generación de utilidades de software, Línea de productos instrumentales, Línea de elaboración de técnicas y Línea de creación de convenios y desarrollo de *agentes*. Nombres que el uso cotidiano simplificó hasta; *Línea* de *utilidades*, de *instrumentos*, de *técnicas* y de *propuestas*.

Al igual que se destacaba en apartados anteriores la necesidad de adoptar una decisión sobre la plataforma multiagente en donde desplegar las aplicaciones, también fue necesario poner una atención especial a la decisión adoptada sobre las tecnologías de motor de inferencia a integrar en el agente. La primera de las decisiones en ese sentido estuvo condicionada por la evolución de CLIPS, al incorporar la pasarela nativa CLIPSJNI y permitir llevar a cabo desarrollos con su motor de reglas en aplicaciones *Java* desde 2012 (última versión en, [Riley, 2016b]). Por otra parte, fue seleccionada la última versión de la *plataforma* multiagente JADE [CSELT & TILab, 2017] y se actualizaron los ejemplos que aporta JADE relacionados

con la integración de *Jess* Alfa 8.0a1, versión que expone una gran estabilidad y una evolución decidida a integrar dispositivos *Android* [Friedman-Hill, 2016].

[Elaboración propia]

Productos aportados por Línea de trabajo

	Líneas de trabajo			
	Utilidades	***Instrumental***	***Elaboración de Técnicas***	***Propuestas***
Productos	framework	Entorno de Desarrollo	Integración de un Intérprete de *Java*	Directorio de Aplicación
	psnode	Ontología Problem-Solvers	Gestión de comportamientos	
	launcher		Inicialización mediante *runlevels*	Agente *PsNodeAgent* + Nodos
		Protocolos de Interacción	Test unitarios y de componentes	
	shell		Comunicación con motor de inferencia	Agente *PsBoardAgent* + Base de datos *ticketDB*
		Documentación en JavaDoc	Migración de Nodos	

Tabla 5.1: Distribución de los Productos aportados por Llínea de trabajo

- La *Línea de trabajo de* ***utilidades*** fue destinada a diseñar e implementar software destinado al *ingeniero de software* y el *administrador de sistemas*. Esas utilidades permiten incluso generar la *infraestructura* empleada por los *agentes*, mediante procesos automatizados redactados como scripts del *sistema operativo*.

- La *Línea de trabajo* ***instrumental*** permitió organizar, entre otros, los trabajos de preparación de la documentación y, los trabajos de implementación del *Entorno de desarrollo* o de la ontología *Problem-Solvers* empleada por los agentes.

- La *Línea de trabajo de **técnicas*** facilitó organizar los trabajos de investigación destinados a: integrar el *motor de inferencia*, incorporar un intérprete de *Java*, descubrir el proceso de migración o crear la fragmentación en *niveles de ejecución*.

- La *Línea de trabajo de **propuestas*** ha aportado el espacio de organización donde imputar los esfuerzos que se han consumido para construir y depurar los agentes y la redacción desarrollo y comprobación de los convenios adoptados para construir las réplicas físicas de la *aplicación* que pueden estar presentes en los servidores gestionados por JADE.

5.2 Modelo de integración de *motor de inferencia* en *agente*

La primera de las cuestiones a resolver quedaba enunciada en los términos siguientes:

(**1ª**) Cómo integrar el *motor de inferencia* en el espacio de ejecución del *agente*.
(**2ª**) Cómo comunicar al *agente* con su *motor de inferencia*.

La *integración* de un motor de inferencia en un *agente-software* cuenta en la actualidad con soluciones destacadas como: JADE–Jess–Protégé por *Hoffman*, *Bellifemine* y *Friedman-Hill* [Hoffman et al., 2001], JADE–Jess por *Cardoso* [Cardoso, 2007], y JADE-DROOLS por *Poggi* y *Tomaiuolo* [Poggi & Tomaiuolo, 2011]. Sin embargo, ninguna acomete el problema de la migración ni la movilidad del agente por lo que, resultaba crítico alcanzar una solución nueva que mantuviese la movilidad del agente en la plataforma y la capacidad para afrontar procesos de inferencia prolongados (minutos, horas o incluso días), sin detener las funciones intrínsecas del agente. Esta solución se denominó "Integración *Agente-Motor* mediante *Interface Java* y conexión a *Nodo*" y se encuentra desarrollada en el apartado 5.6.5, pág. 98.

Para solventar las *comunicaciones* entre *agente* y *motor de inferencia* se revisaron soluciones publicadas por [Hoffman et al., 2001] y [Cardoso, 2007]. Sus propuestas emplean la función `(store)` de *Jess*, para traspasar información entre *agente* y *motor* a través de una estructura de datos de tipo *HashMap*. Es una solución viable pero, no permite afrontar un proceso de migración de una aplicación heredada completa para lograr su despliegue como aplicación distribuida compuesta por agentes en interacción. Se logró entones la solución basada en el uso de *Mensajes ACL* de forma que, *agente* y *motor*, se identifiquen y se comuniquen intercambiando datos exclusivamente mediante mecanismos alineados con las recomendaciones de la fundación FIPA para la construcción de agentes inteligentes. Esa solución de comunicaciones entre el agente y su motor de reglas se denominó "Comunicación *agente-motor* mediante *Protocolo de Interacción*" y se encuentra desarrollada en el apartado 5.5.3, pág. 88.

5.3 Diagrama: 'Modelo del dominio'

Otro descubrimiento trascendente fue el mecanismo necesario para desplegar un *sistema de producción* en el escenario multiagente: imprescindible para dotar de capacidades a los agentes para que pudiesen mantener y recargar el estado de la *base de conocimientos* y *la memoria de trabajo* que les compete e intercambiar, con otros agentes del mismo *sistema de producción*, tanto *Mensajes ACL* como *reglas*, *hechos* o instrucciones.

Cód.	***Término** empleado en el Modelo*	***Concepto** asociado*
(1)	DPS[32]-Framework-Builder	Constructor de Directorios de aplicación
(2)	DPS-Framework	Directorio de aplicación
(3)	Rule-Based Engines	Motores del sistema experto a integrar
(4)	MAS[33] Platform	Plataforma multiagente
(5)	Agent::Problem-Solvers Node	Instancias de Agentes-nodo
(6)	Tools and Utilities	Utilidades y herramientas auxiliares

Tabla 5.2: Términos asignados a entidades en el Modelo del Dominio

Para obtener esa ampliación de capacidades en los agentes era necesario concluir la solución para que los agentes-software alojasen la instancia del motor de inferencia y además, el escenario para que esos agentes pudiesen ejecutarse en cualquier arquitectura hardware y sistema operativo; manteniendo así su capacidad de migración entre contenedores de la plataforma.

[Elaboración propia]

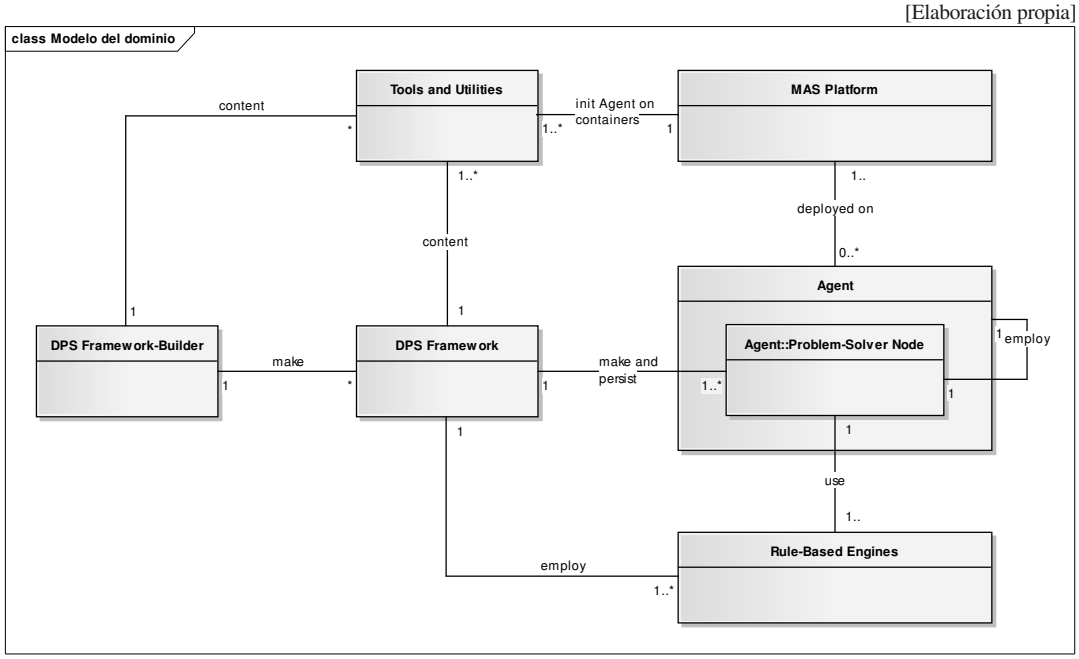


Figura 5.1: Modelo del dominio de las entidades participantes

El diagrama de la Figura 5.1 representa el *Modelo del dominio* de las entidades que han participado en la solución de integración del motor de inferencia y la construcción del escenario de despliegues, mostrando además, las interacciones entres dichas entidades. Recuerda *Craig Larman* [Larman, 2002] que «...algunas de esas entidades y conceptos del *Modelo del dominio* coincidirán finalmente con objetos y artefactos de software». En el capítulo 6º, dedicado a *pruebas de concepto*, se puede comprobar la validez de este diagrama de entidades a medida que se muestra cómo distribuir aplicaciones basadas en *sistemas de producción* sobre la plataforma multiagente JADE.

[32] DPS-Framework-Builder: Constructor del Entorno de Despliegue de Sistemas de Producción.
[33] ***MAS***: Multi-Agent Systems, o Sistemas multiagente.

A partir del diagrama de la Figura 5.1, se desprende que el modelo de integración entre *agente* y *motor de inferencia* necesario para facilitar el despliegue de aplicaciones completas basadas en *sistemas de producción*, exigía adoptar la decisión de diseño de incorporar en el agente; mecanismos para que el *agente* entrase en contacto con la realidad del sistema de ficheros utilizado por su motor. Por ese motivo, esta propuesta de integración se alejó definitivamente de las soluciones Jess–JADE–Protégé [Hoffman et al., 2001] y del modelo de integración JADE–Jess de [Cardoso, 2007] que ejecutan la plataforma multiagente desde el espacio de ejecución interna del *motor de inferencia*, como solución para mantener el acceso a los archivos de *reglas* y *hechos* desde el agente.

Aunque en las fases iniciales de la investigación se produjo un acercamiento y cierto paralelismo a los trabajos de *Poggi* y *Tomaiuolo* (ver [Poggi & Tomaiuolo, 2011], pág.5 sobre integración del motor *JBoss Rules* con una interface entre el motor de reglas y el modelo de comunicaciones definido por FIPA) se concluyó que; no sería de utilidad dicho paradigma en soluciones destinadas a facilitar el despliegue de *sistemas de producción* sobre plataformas de agentes ya que, presuponen que sería posible emplear el Lenguaje de programación de la tecnología del sistema experto para modelar e implementar el modelo de comunicaciones de FIPA en su interior mediante la extensión del objeto *motor de reglas*. Frente a tal circunstancia, las líneas de trabajo retornaron al *Modelo del dominio* de la Figura 5.1, donde la entidad denominada como *Agent::Problem-Solvers-Node*, es la pieza que aloja en su interior al *motor de inferencia* y el elemento responsable de explotar el modelo de comunicaciones FIPA con los mecanismos que le facilita la plataforma multiagente. Se exponen a continuación los significados de las distintas entidades mostradas en el *Modelo del dominio*.

5.3.1 Entidad: Constructor de Directorios de aplicación

La entidad *DPS-Framework-Builder* (Figura 5.1) representa el elemento responsable de fabricar el escenario de ejecución de la aplicación a transformar. Esa entidad analiza la arquitectura hardware, el sistema operativo y la máquina de *Java* del servidor donde será ejecutada. Con esos datos genera un directorio compacto formado por librerías que permiten el funcionamiento autónomo de los agentes que compondrán la aplicación. El *constructor de Frameworks* asegura la capacidad de migración de los *agentes* hacia contenedores remotos de la plataforma donde existen recursos de CPU o recursos de almacenamiento a nuestra disposición. Los agentes poseen la capacidad nómada de forma intrínseca [Helin, 2003] y esa propiedad se mantiene durante el proceso de transformación para asegurar que la distribución de carga computacional es viable y, aún más importante si cabe, es eficiente y es rentable en términos del esfuerzo que supone realizar la migración, depuración y puesta en servicio.

5.3.2 Entidad: Framework o el Directorio de aplicación

La entidad *Directorio de aplicación*, coincide con el directorio físico donde se almacenan todos los recursos y archivos de la aplicación original. El *DPS-Framework* es la representación materializada de cada réplica de la aplicación una vez distribuida y en ejecución sobre la plataforma JADE. Es un elemento visible para el observador y físicamente manipulable. Es una *estructura de directorios* sobre el sistema de ficheros y además, es un *convenio de nomenclaturas* para identificar los archivos (*véase* Apdo. 5.7.1, pág. 100).

El *DPS-Framework* ha sido creado con la intención de facilitar al *desarrollador* el versionado y la autonomía de la aplicación (incluidas localizaciones en dispositivos USB). Con esa propiedad, el *Directorio de aplicación* puede ser reubicado en puntos de montaje del sistema operativo cuando sea necesario escalar hacia a entornos de producción reales.

5.3.3 Entidad: Motores basados en reglas

La entidad *Rule-Based Engines* representa el *motor de inferencia* a integrar en cada agente. Las tecnologías de sistemas expertos utilizadas por las aplicaciones candidatas deben corresponderse con alguna de las herramientas de construcción de sistemas expertos citadas (véase herramientas en Tabla 5.3, pág. 70).

5.3.4 Entidad: Plataforma Multiagente

La entidad *MAS-Platform* en el diagrama del *Modelo del dominio* (Figura 5.1) representa la plataforma multiagente donde se despliegan los *agentes*.

5.3.5 Entidad: Agente—Nodo instancia del sistema de producción

Aunque no es propio de un *modelo del domino* [Griss et al., 2003], se ha asimilado esta entidad a objetos que representan las instancias de la clase `jade.core.Agent`. El significado de esas entidades es equiparable a *agentes* con un motor de inferencia integrado con el que se comunican mediante un mecanismo que no genera estados de bloqueo durante la obtención de inferencias realizadas por el motor, ni tampoco genera bloqueos por la ejecución de los comportamientos intrínsecos del *agente*.

5.3.6 Entidad: Herramientas y utilidades

Representan en el diagrama de *Modelo del dominio*, el grupo de piezas de software que permiten al *desarrollador*: iniciar, detener, desplegar sobre la plataforma, cambiar de contenedor, realizar ensayos para fragmentar la aplicación, crear nuevos agente, programar, verificar comportamientos, etc. Todas esas piezas de software se han construido de forma que están siempre accesibles desde el *Directorio de aplicación* cuando la variable CLASSPATH de *Java*, contiene [`lib/*`]. Las soluciones implementadas permiten la configuración de múltiples *Directorios de aplicación* conectados a las librerías centralizadas del servidor.

5.3.7 Entidad: Entorno Tecnológico de software

Aunque no forma parte del diagrama de *Modelo del dominio*, el entorno tecnológico subyace y es internamente reconocido por el *Constructor de Directorios de aplicación* para elaborar su estructura de forma coherente con el hardware, el sistema operativo y el tipo de motor de inferencia utilizado por la aplicación. En la Tabla 5.3 (ver pág. 70) se exponen las versiones de las herramientas de construcción de sistemas expertos utilizadas y de los componentes adicionales *Java* empleados.

En la Tabla 5.3 se observa como *Jess*, GNU-Prolog y Drools, se encuentran nativamente construidos en *Java* y como CLIPS requiere una *pasarela* para conectar *Java* a su motor nativo. Se seleccionó una tecnología de herramienta de construcción de sistemas expertos con

pasarela y otra sin ella para implementar el *Directorio de aplicación*. Las restantes entradas, excepto JADE y las librerías del sistema de gestión de bases de datos HSQLDB, se corresponden con librerías empleadas por el *agente* denominado *Entorno de desarrollo.*

Los *agentes* utilizan *Jess* o la *pasarela* CLIPSJNI para integrar un motor de inferencia y la librería *BeanShell*® (*Pat Niemeyer* [Niemeyer, 2008], *véase* Tabla 5.3, pág. 70) para integrar un intérprete de *Java* que permite al *agente*: externalizar comportamientos, compilar clases de comportamientos en tiempo de ejecución, ejecutar guiones de arranque por fases de inicialización y cargar funcionalidades redactadas sobre archivos en formato de texto.

[Elaboración propia]

Entorno Tecnológico: API de Sistemas expertos y Librerías utilizadas

Herramienta Librería	*Pasarela*	*Versión*	*OS y Máquina Java*	*Arquitectura*
CLIPS	CLIPSJNI - 0.3	6.30.15	Windows, OS-X Linux .dll/.so /.jnlib	32 y 64 bits
	CLIPSJNI - 0.5	**6.31.15**		
Jess	no necesita	6.1p8 (JRE 1.6)	JRE	
		7.1p2		
		8.0a1		
JADE	no necesita	**4.5.0**	JRE	
HSQLDB	no necesita	**5.2.2.1**	JRE	
SQLtool	no necesita	**5.3.3.7**	JRE + HSQLDB	
BeanShell	no necesita	**2.0b4**	JRE	
RSyntaxtextArea	no necesita	**2.5.7**	JRE	

Tabla 5.3: Descripción del Entorno Tecnológico: librerias y versiones empleadas

5.4 Relación de *utilidades-software* aportadas

Las *utilidades auxiliares* aportadas son las siguientes (*ver* Columna 2ª, Tabla 5.1, pág. 65):

(1) Utilidad de terminal: **framework**

(2) Utilidad de terminal: **psnode**

(3) Utilidad de terminal: **launcher**

(4) Utilidad de terminal: **shell**

5.4.1 Utilidad: framework o [dps-frameworkBuilder.jar]

Es una pieza de software encargada de construir el *Directorio de aplicación*. Se ejecuta desde línea de comandos de dos formas distintas. La primera es directa, al invocar al archivo denominado (*dps-frameworkBuilder-0.9.1.jar*) con la opción [-jar], la segunda requiere establecer la variable `CLASSPATH` para un funcionamiento correcto. Estos son los pasos necesarios a seguir para su ejecución desde un terminal:

1. Copiar el archivo `dps-frameworkBuilder-0.9.1.jar`

2. Invocar a la utilidad con la opción (***a***), o establecer la variable `CLASSPATH`, opción (***b***)

```
Opción (a) Sin establecer ninguna variable (en cualquier sistema operativo):
-------------------------------------------------------------------------------
[~/pruebas/]#  java -jar dps-frameworkBuilder-0.9.1.jar  new  app51

-------------------------------------------------------------------------------
Opción (b.1) Apple OS-X o GNU-Linux:         Opción (b.2) Microsoft Windows:
-------------------------------------------------------------------------------
[~/pruebas/]#  export CLASSPATH=./*:lib/*:    c:\pruebas\> set CLASSPATH=./*;lib/*;

-------------------------------------------------------------------------------
Opción (b) Crear el Directorio de aplicación mediante:
-------------------------------------------------------------------------------
[~/pruebas/]#  java framework  new app51
```

Código 5.1: Creación del Directorio de aplicación con dps-frameworkBuilder

Se aconseja emplear la opción (***a***) porque el archivo (*dps-frameworkBuilder-0.9.1.jar*) de instalación puede ser eliminado del sistema una vez creado el *Directorio de aplicación*.

[Elaboración propia]

```
[~/pruebas/]# java -jar dps-frameworkBuilder-0.9.1.jar
 ---------------------------------------------------------------------
 Department of Electrical Engineering, Systems and Automatic
 UNIVERSITY of LEON
 ---------------------------------------------------------------------
 framework v:0.9.1      Problem-Solvers Framework Building Utility.

 Usage   : java framework [Options]

 Options : [new |--new=] Framework-Name   Make an empty Framework.
           [-h  |--help]                  Show environment info.
           [-n  |--no-doc]                Don't install help-documents.
           [-l  |--no-lib]                Don't install libraries.

 Examples: java framework  new  app51
           java framework  -h
 Note:
           On Windows family: set    CLASSPATH=./*;lib/*;
           On Linux or OSX  : export CLASSPATH=./*:lib/*:
           Before running.

 ---------------------------------------------------------------------
 Author     : Francisco J. Aguayo                  (Copyright)-2017
 Directed by: Ph.D. Isaias Garcia     Revised by: Ph.D. Hector Alaiz
 ---------------------------------------------------------------------
```

Código 5.2: Opciones de la utilidad de construcción de Directorios de aplicación

5.4.2 Utilidad: psnode (Production System Node)

Esta pieza de software permite crear nuevos *nodos*, empleados por los agentes que compondrán la aplicación. En cada *nodo* se localizan los archivos de configuración del propio *agente* y los archivos del *sistema de producción* que serán asignados al agente. Para un correcto funcionamiento de la utilidad `psnode` es necesario ubicarse sobre la raíz del *Directorio de aplicación* que se ha creado en el apartado anterior.

[Elaboración propia]

```
[~/pruebas/]#  cd app51
[~/pruebas/app51/]#  java  psnode
```

```
-----------------------------------------------------------------
Department of Electrical Engineering, Systems and Automatic
UNIVERSITY of LEON
-----------------------------------------------------------------
psnode v:0.9.1     Problem-Solvers Agent Template building utility.

Usage   : java psnode [Options]  [Engine-Model]

Options : [new |--new=] [name-of-Node]  Make new Node empty space.
          [-h  |--help]                        View extended info.

Examples: java psnode --new=robot74  CLIPS
          java psnode new Bot501,Bot511,Bot521,Bot591
-----------------------------------------------------------------
Author     : Francisco J. Aguayo                   (Copyright)-2017
Directed by: Ph.D. Isaias Garcia     Revised by: Ph.D. Hector Alaiz
-----------------------------------------------------------------
```

Código 5.3: Utilidad 'psnode' para construcción de la estructura de un nodo

Se muestran ejemplos de cómo emplear `psnode` para generar uno, o varios *nodos*, que serán creados dentro del interior del *Directorio de aplicación*, e.g. `Yarvis`, o una secuencia de *nodos* denominados `RobotN`, desde el [1 hasta el 10]. Las sentencias necesarias serían:

```
[~/pruebas/app51/]#  java psnode --new=Yarvis
[~/pruebas/app51/]#  java psnode  new Bot501,Bot511,Bot521,Bot591
[~/pruebas/app51/]#  for i  in  `seq  1  10`
                        do
                          java psnode --new=Robot$i
                       done

[~/pruebas/app51/]#  ls -la  nodes/
```

Código 5.4: Ejemplo de ejecución de la utilidad 'psnode'

Para realizar una comprobación de funcionamiento, pueden lanzarse sobre la plataforma los *agentes* con los nombres correspondientes a los *nodos* generados, como por ejemplo: `Yarvis` y analizar el mecanismo que utilizan los *agentes* para establecer su conexión con la estructura del *nodo*. Para ello, se emplea la utilidad de lanzamientos `launcher`:

```
[~/pruebas/app51/]#  java launcher platform localhost
[~/pruebas/app51/]#  java launcher node     localhost Yarvis CLIPS
```

Código 5.5: Despliegue de agente en plataforma JADE con utilidad launcher

5.4.3 Utilidad: launcher (Lanzadera genérica)

La pieza de software *launcher* es una utilidad de línea de comandos que facilita la apertura de la plataforma, el despliegue de un agente, de un grupo de *agentes*, de un agente en su *Entorno de Desarrollo*, etc. Esta utilidad comprueba el nombre del *hosts* (mediante una consulta realizada al servidor DNS por defecto) y la disponibilidad de los *puertos* solicitados (para ubicar el Servicio adecuadamente), tanto a nivel local como en localizaciones remotas:

[Elaboración propia]

```
[~/pruebas/app51/]# java launcher -h
 -----------------------------------------------------------------
 Department of Electrical Engineering, Systems and Automatic
```

```
UNIVERSITY of LEON
------------------------------------------------------------------
launcher  v0.9.1          Agent and JADE-Platform Launcher Utility

Usage   : java launcher [Objective] [Server] [Name] [Engine] [port]

Examples:
==========::                          [[JADE Platform]] ::=======================================
java launcher platform                                    (Open JADE Platform in localhost:1099)
java launcher platform  NASA21                         (Open JADE Platform in server NASA21:1099)
java launcher platform  localhost 4455                                    (idem. on port:4455)

==========::   [[Collection of Problem-Solvers agents]] ::=======================================
java launcher nodes     localhost CLIPS            (Open all nodes in localhost:1099 with clips)
java launcher nodes     localhost jess 4455         (Open all nodes in localhost:4455 with jess)

==========::                           [[TicketBoard agents]] ::================================
java launcher board                           (Open Default TicketBoard Agent in localhost:1099)
java launcher board     localhost 8877                                    (idem. on port:8877)
java launcher board     localhost ts02           (Open TicketBoard named ts02 in localhost:1099)
java launcher board     localhost ts04  4455     (Open TicketBoard named ts04 in localhost:4455)

==========::                [[PSNode-Agents on Test-Environment]] ::============================
java launcher pocket                               (Empty Test-Environment with CLIPS on :1099)
java launcher pocket    localhost 4455                                        (idem. in :4455)
java launcher pocket    localhost Pocket jess       (Empty Test-Environment with Jess  on :1099)
java launcher pocket    localhost Pocket jess  5566  (Empty Test-Environment with Jess  on :5566)
java launcher pocket    localhost Bot25 CLIPS                    (Open Bot25 with CLIPS Engine)
java launcher pocket    NASA21    Bot24 jess       (Open Bot24 with JESS Engine on NASA21 server)
java launcher pocket    localhost Bot22 CLIPS 5544                    (Bot22 on localhost:5544)

==========::                               [[PsNode-Agent alone]] ::============================
java launcher node      localhost Bot25 CLIPS                    (Open Bot25 with CLIPS Engine)
java launcher node      NASA21    Bot24 jess       (Open Bot24 with JESS Engine on NASA21 server)
java launcher node      localhost Bot22 CLIPS 5544                    (Bot22 on localhost:5544)

==========::                          [[Dummy-Agent on Test-Environment]] ::==================
java launcher dummy                                  (Simple Test-Environment for DummyAgent)
java launcher dummy     localhost 4455          (Simple Test-Environment for DummyAgent on :4455)
java launcher dummy     localhost Pluto 6655           (Open DummyAgent named Pluto on port:6655)

==========::                     [[JADE.core.Agent on Test-Environment]]  ::==================
java launcher agent                                      (Empty Agent Test-Environment)
java launcher agent     localhost 4455          (Simple Test-Environment for JADE Agent on :4455)
java launcher agent     localhost Draft 6655          (Open Empty Agent named Draft on port:6655)
------------------------------------------------------------------
Author     : Francisco J. Aguayo                  (Copyright)-2017
Directed by: Ph.D. Isaias Garcia     Revised by: Ph.D. Hector Alaiz
------------------------------------------------------------------
```

Código 5.6: Utilidad 'launcher' para lanzamiento de agentes (ayuda)

La utilidad `launcher` permite diversas opciones para cubrir el espectro de necesidades de administración y de depuración y aporta parámetros adicionales para realizar el despliegue de los *agentes* dentro del *Entorno de desarrollo*. La utilidad *Launcher* interpreta hasta cinco parámetros distintos para desplegar por ejemplo: la plataforma JADE, todos los nodos del *Directorio de aplicación* simultáneamente, agentes en su *Entorno de desarrollo*, un Entorno de Desarrollo en vacío (sin agente de la aplicación asociado), agentes que prestan el *Servicio de repositorio de tickets* o, *agentes de la aplicación* que no poseen conexión a un motor de inferencia y que son de gran utilidad para asistir en la migración de agentes entre contenedores. La utilidad auxiliar `launcher` ha demostrado su gran valor durante el proceso de depuración de las réplicas de la *aplicación* y para el despliegue de agentes en a plataforma JADE.

La utilidad *Launcher* admite las funciones siguientes: *platform*, *nodes*, *node*, *pocket*, *board*, *dummy* y *agent*. La secuencia habitual de parámetros en la utilidad *Launcher* comienza por la función, seguida del servidor sobre se realizará el lanzamiento y, a continuación, del nombre

del agente a desplegar dentro del *Entorno de desarrollo* más el modelo de motor de inferencia y el puerto de conexión a la plataforma multiagente. Se listan a continuación algunos *alias* incorporados por defecto a la utilidad *Launcher*:

[Elaboración propia]

```
Alias:                                Comando completo asociado:
-------------------------------     ---------------------------------------------
launcher pocket               ==    launcher pocket localhost Pocket CLIPS 1099
launcher dummy                ==    launcher dummy localhost Dummy 1099
launcher platform             ==    launcher platform 127.0.0.1 1099
launcher agent                ==    launcher agent localhost Agent 1099
launcher board                ==    launcher board localhost Board 1099
launcher nodes                ==    launcher nodes localhost clips 1099
launcher pocket localhost Yarvis ==  launcher pocket localhost Yarvis CLIPS 1099
```

Código 5.7: Alias utlizados habitualmente con la utilidad 'launcher'

5.4.4 Utilidad: shell (Entornos de ejecución de comandos)

Es una pieza de software construida para facilitar el acceso a las cuatro *shell* más utilizadas durante los procesos de transformación de aplicaciones basadas en *sistemas de producción* al paradigma multiagente. Permite abrir el Intérprete de *Java* con librerías importadas para programar en JADE, CLIPS y la API de Jess. En el caso de emplear la utilidad `shell` para iniciar un intérprete de CLIPS 6.31, denotar que se produce un volcado continuo de las instrucciones tecleadas archivos del directorio [`./logs/`]. El intérprete SQL conectado al motor de bases de datos HyperSonic-SQL facilita la consulta de la base de datos de los agentes que prestan el servicio de almacenamiento de Tickets (objetos analizados en Apdo. 5.5.1.4). Y el intérprete de *Jess*, que integra una llamada directa al objeto *Main* del constructor de sistemas de producción *Jess* en modo *-developer*. Desde el directorio de aplicación se invoca con:

[Elaboración propia]

```
[~/pruebas/app51/]# java shell
 ---------------------------------------------------------------------
 Department of Electrical Engineering, Systems and Automatic
 UNIVERSITY of LEON
 ---------------------------------------------------------------------
 shell v:0.9.1            Local DPS-Framework Shells access utility

 Usage   : java shell  [options] [params]
 Options : [CLIPS |JESS |JAVA |HSQL ]   Shell type
           [-h |--help]                 Full help
 Params. : [-g |-gui|--gui ]            Open in graphic environment
 Examples: java shell Java     or     java shell JAVA  -gui
           java shell CLIPS    or     java shell Jess
           java shell HSQL     or     java shell HSQL  -gui
 ---------------------------------------------------------------------
 Author     : Francisco J. Aguayo                  (Copyright)-2017
 Directed by: Ph.D. Isaias Garcia     Revised by: Ph.D. Hector Alaiz
 ---------------------------------------------------------------------
```

Código 5.8: Utilidad 'shell' para acceder a intérpretes del Framework

Destacar que, si las librerías del *Directorio de aplicación* se configuran en *modo infraestructura* (*véase* 6.2, pág. 118), la utilidad *shell* de CLIPS y la utilidad *shell* de *Jess*, podrán ser ejecutadas y serán plenamente operativas en los directorios `./engine/` de cada uno de los *nodos* (i.e.: /apps51/nodes/Yarvis/engine/). A continuación se muestran los resultados de ejecución de las distintas *Shells* y las sentencias necesarias para el cierre de cada una de ellas:

[Elaboración propia]

```
[~/pruebas/app51/]#  java shell java
   BeanShell 2.0b4 - by Pat Niemeyer (pat@pat.net)

   [framework::/app51/]# > System.exit(0);

[~/pruebas/app51/]#  java shell clips
   Giarratano, Joseph C and Riley, Gary
   ------------------------------------
   Java Native Interface                       (JNI): 0.5
   C++ Language Integrated Production System (CLIPS): 0.5
           CLIPS (6.31 5/19/15)

   CLIPS> (exit)

[~/pruebas/app51/]#  java shell jess
   Ernest Friedman-Hill.  Jess, the Rule Engine for the Java Platform
   ------------------------------------------------------------------
   Licencia: Copyright (C) 2013 Sandia Corporation
   Version : Jess Version 8.0a1 9/25/2013

  Jess> (exit)


[~/pruebas/app51/]#  java shell hsql
   SqlTool v. 5337.
   SqlFile processor v. 5448.
   Distribution is permitted under the terms of the HSQLDB license.
   (c) 2004-2011 Blaine Simpson and the HSQL Development Group.

   sql> \q
```

Código 5.9: Métodos de cierre de Shell: CLIPS, Jess, Java y Hsql

5.5 Relación de *productos-instrumentales* aportados

Los artefactos software de carácter instrumental son: ontología Problem-Solvers, Entorno de desarrollo, Protocolo de interacción de doble secuencia más el Protocolo de interacción del servicio de tickets y Documentación (Tabla 5.1, pág. 65). Los instrumentos permiten planificar, analizar en pantalla los comportamientos y evolucionar la solución distribuida de la aplicación, dotar de significado semántico a los mensajes intercambiados entre los agentes, etc.

La Documentación de las clases *Java* implementadas para dar forma a las utilidades y a los agentes propuestos; asegura una visión atómica de cada componente y un lugar para la consulta de los manuales, las API de CLIPSJNI, Jess, JPL y BeanShell, demostrando ser de gran ayuda en proyectos de reconversión de aplicaciones. La Documentación se ha incluido en el subdirectorio `/var/doc/index.html`, para cada *Directorio de aplicación* creado sin la opción [`--no-doc`]. Se detallan seguidamente los artefactos instrumentales implementados.

5.5.1 Instrumento: Ontología *Problem-Solvers*

La ontología *Problem-Solvers* es la propuesta, ajustada a directrices de FIPA, destinada a definir *conceptos* y *acciones* para el *dominio* asociado al despliegue de aplicaciones basadas en *sistemas de producción* sobre plataforma multiagente. Esta ontología es utilizada pues, para incorporar significados en el contenido de los *mensajes* intercambiados entre agentes.

La implementación que aporta la plataforma JADE de objetos tipo *Mensajes ACL*, permite incorporar una ontología adicional a la que incorporan *por defecto* los *Mensaje ACL* (*véase* Apdo. 3.2.3, pág.37). Al emplear, como codificación del contenido del mensaje, el formato de *cadenas FIPA*, los parámetros y valores son inteligibles para humanos y quedan expuestos en texto plano. Con ese tipo de codificación es posible depurar y analizar los ejemplos siguientes; observando cuál es la estructura del contenido de los mensajes ACL y cuando incorporan objetos de la ontología *Problem-Solvers* en el interior de los mismos:

```
(INFORM
   :sender    (agent-identifier :name robot220@POSEIDON:1099/JADE )
   :receiver  (set  ( agent-identifier :name robot260@POSEIDON:1099/JADE ) )
   :content  "((done (action (agent-identifier :name robot220@POSEIDON:1099/JADE )
                (process :language CLIPS
                         :method (sequence TEST_ACTIVATIONS \"\")))))"
   :encoding  fipa.acl.rep.string.std
   :language  fipa-sl0
   :ontology  Problem-Solvers
   :protocol  fipa-request
   :conversation-id  Basal_Behaviour_Agenda
   :X-psStrEv "innEng.eval((agenda))"
   :X-psOutValue 2
   :X-psSCode TEST_ACTIVATIONS
   :X-psAction Process
   ...
   :X-psConversationID Basal_Behaviour_Agenda )
   (Envelope  :to (sequence  ( agent-identifier :name robot260@POSEIDON:1099/JADE ))
              :from ( agent-identifier :name robot220@POSEIDON:1099/JADE )
              :comments 102
   ) )
```

Código 5.10: Ejemplo de Mensaje ACL con objetos de ontología Problem-Solvers

En el Código 5.10 del ejemplo, la acción (`done`) pertenece a la ontología `fipa-acl`, e incorpora el significado de *conclusión con éxito* de una petición anterior, formada por una acción tipo *Process* (*procesamiento*) de la ontología *Problem-Solvers*. Se utilizan así, ambos dominios de conocimiento (Problem-Solvers y JADEOntologyManagement), en un mismo mensaje ACL para informar al solicitante que una acción requerida fue culminada.

5.5.1.1 *Esquema de la ontología Problem-Solvers*

Paolo Cancedda, *Giovanni Caire* y *David Cabanillas* en [Caire & Cabanillas, 2010], [Cancedda & Caire, 2010] establecen los instrumentos en JADE para que los agentes puedan realizar comprobaciones semánticas de las expresiones inmersas en el *contenido* de un mensaje mediante la clasificación de los posibles elementos del dominio del discurso, esto es; de los elementos que pueden aparecer dentro de sentencias válidas y con significado una vez que son incorporadas en los *Mensajes ACL*. La ontología *'Problem-Solvers'* que se propone, contiene los conceptos y las acciones mínimas necesarias para la comunicación: entre *agentes* de un *Directorio de aplicación*, entre el *agente* y su *motor de inferencia* y entre *agentes software* y el *Actor-humano* cuando éste interactúa sobre la consola de comunicación asíncrona (en la interfaz gráfica del *Entorno de Desarrollo*) redactando y recibiendo respuestas del motor de inferencia.

En JADE, las ontologías se expresan mediante objetos simples de *Java*. El código de la ontología *'Problem-Solvers'* se encuentra en las clases del paquete [`examples.psnode.domain`] y

queda definida en la clase `ProblemSolversOntology`. Para facilitar la inspección se ha incorporado el método `main()` a la clase `ProblemSolversOntology` para obtener el volcado por pantalla de su estructura con las propiedades de cada uno de los *conceptos*, *acciones* y *predicados* que posee implementados (Código 5.11, a continuación):

```
[~/pruebas/app51]# >  java examples.psnode.domain.ProblemSolversOntology
```

Código 5.11: Comprobación de ontología Problem-Solvers desde consola

[Elaboración propia]

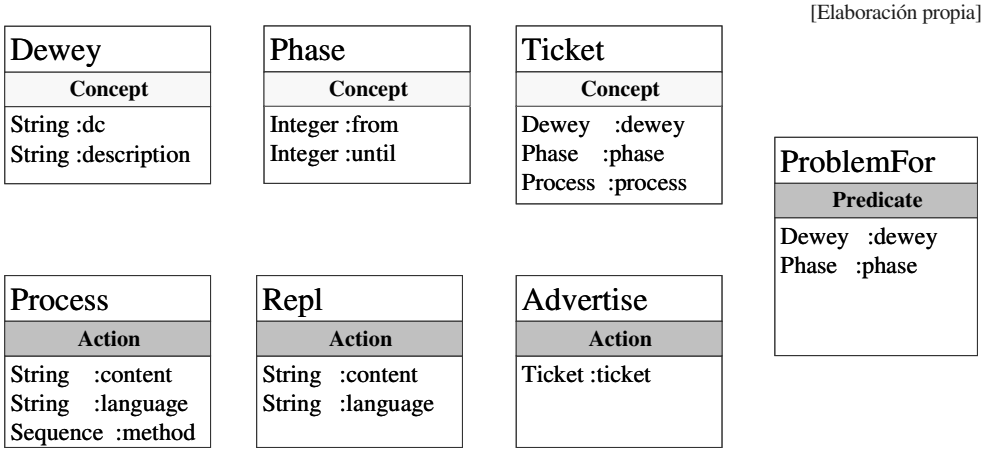


Figura 5.2: Ontología para comunicación entre agentes del Directorio de aplicación

5.5.1.2 *Concepto 'Dewey' en la ontología*

El concepto *Dewey* es un objeto formado por dos atributos, [`:dc`] y [`:description`] ambos de tipo texto. El atributo [`:dc`] es el código que identifica el *campo de conocimiento* donde el agente es competente y puede resolver. Para realizar la clasificación del atributo [`:dc`] se adopta la notación decimal de ordenación de temáticas publicada por *Melvil Dewey*[34] conocida como DDC[35] (de uso habitual en bibliotecas y universidades). El primer atributo [`:dc`] es obligatorio y el segundo atributo [`:description`] es opcional.

La clasificación DDC [Benito, 2005], empleada en el atributo [`:dc`], tiene potencia suficiente para distinguir con fiabilidad capacidades de resolución de problemas asociados a temáticas tan cercanas como por ejemplo: *problemas de lógica matemática tipo Sudoku* [`:dc 793.74`] de problemas específicos de *Puzzles y Juegos de puzzles* [`:dc 793.73`], sin que se produzcan colisiones o errores al determinar cuál es el *dominio de conocimiento* que dicho agente es capaz de acometer.

---

[34] ***Melvil Dewey:*** publica en 1876 el sistema de clasificación de bibliotecas conocido en la actualidad como *Sistema Decimal de Dewey*, aplicado en todo el mundo para la ordenación de las temáticas de todas las publicaciones asociadas mediante un código de indexación decimal. Hoy es mantenido por la OCLC *Online Computer Library Center*.

[35] ***DDC:*** Dewey Decimal Classification.

5.5.1.3 *Concepto 'Phase' en la ontología*

El concepto *Phase* es un objeto que contiene el rango de posibles etapas de resolución en las que el agente puede participar para colaborar en la resolución de problemas. La *Phase* consta de dos atributos: [`:from`] y [`:until`] ambos declarados *Integer*. Se ha establecido un convenio para interpretar el significado de dos valores singulares:

`[:from 0]` Indica que el agente tiene capacidad para resolver problemas, de dicho dominio de conocimiento desde su planteamiento inicial.

`[:until -1]` Indica que el agente tiene capacidad para resolver problemas, de dicho dominio de conocimiento, hasta su conclusión; entregando el resultado final.

Cuando un *agente* o el *Actor-humano* generan un objeto **Ticket** (definido a continuación) y establecen el valor del atributo [`:from 0`] con valor igual a `0`, denotan con ello que los datos asociados corresponden al planteamiento inicial del problema a solventar.

5.5.1.4 *Concepto 'Ticket' en la ontología*

El concepto *Ticket* representa la *unidad de almacenamiento de información* para resolver problemas entre agentes. Contiene la identificación del *dominio de conocimiento* asociado al tipo de problema a solventar mediante un objeto [`dewey`], la *fase* en la que se encuentran los datos de resolución asociados a dicho problema expresada en un objeto tipo [`phase`] y el conjunto de datos e información adicional necesarios para iniciar, o para continuar, con el problema en curso de resolución en un objeto de tipo [`process`].

Un *Mensaje ACL* con un *Ticket* (Figura 5.2) posee 7 atributos en su contenido: dos derivan de [`dewey`], dos de la [`phase`] y tres más incorporados por el objeto [`process`]. Todos esos objetos incorporan la información y los *datos* del problema (en ese instante de resolución), y aportan metainformación sobre el *lenguaje nativo* del motor de inferencia en el que se pueden interpretar dichos datos, más una *lista* adicional de instrucciones para acometer situaciones complejas de resolución de problemas. Además, el *Ticket* es el objeto almacenado en la *base de datos* ubicada en el subdirectorio `/var/ticketDB`, de los agentes *PsBoardAgent* (*véase* Apdo. 5.7.4, pág. 105) que ofrecen el *Servicio de Tickets* en la plataforma JADE.

Los *agentes* con *motor de inferencia* integrado que requieren colaborar con otros agentes para solventar problemas complejos, requieren que dichos problemas sean descompuestos en fases e incorporar, en sus archivos de propiedades, el rango posible de participación en la resolución. Una vez establecido ese escenario de colaboración, dichos agentes pueden crear objetos de tipo *Tickets* y remitirlos al *Servicio de Tickets* para que otros agentes puedan capturarlos y avanzar en su resolución. Por tanto, los agentes que colaboran en la resolución de problemas son simultáneamente generadores y consumidores de objetos *Tickets*.

5.5.1.5 *Predicado 'ProblemFor' en la ontología*

El objeto de tipo predicado *ProblemFor* permite construir *Mensaje ACL* a los agentes para interrogar al *Servicio de Tickets*, y averiguar la existencia de objetos *Tickets* que cumplan sus condiciones para colaborar. Cuando los agentes modelan y construyen estos mensajes, emplean sus propiedades como valores de descripción de las consultas.

Desde el punto de vista de los agentes *PsBoardAgent*, esos mensajes con objetos *ProblemFor* contienen valores precisos sobre el identificador del *dominio de conocimiento* y del *rango de competencias*, ambos necesarios para componer la consulta SQL y extraer objetos *Tickets* que son remitidos como respuesta a la consulta recibida.

```
(QUERY-IF
 :sender       ( agent-identifier        :name nodoA@POSEIDON:1099/JADE)
 :receiver     (set ( agent-identifier   :name Board@POSEIDON:1099/JADE ))
 :content      "((problem-for
                      (dewey  :dc  793.74    :description "Sudoku 5x5")
                      (phase  :from 2        :until 99)                  ))"
 :encoding  fipa.acl.rep.string.std
 :language  fipa-sl0
 :ontology  Problem-Solvers
```

Código 5.12: Ejemplo de mensaje ACL con predicado ProblemFor

En el Código 5.12, los conceptos: [dewey] y [phase] son tratados como los elementos que construyen las expresiones de la consulta de identificación de registros de *Tickets*. Los mensajes con predicados *ProblemFor* podrían utilizar el *acto comunicativo* tipo INFORM pero, se empleó finalmente QUERY-IF para eliminar confusiones, como muestra el ejemplo.

5.5.1.6 *Acción 'Advertise' en la ontología*

Una acción *Advertise* incluida en un *Mensaje ACL*, tiene la misión de provocar el desencadenamiento en el agente que presta el *Servicio de Tickets* del proceso de catálogo del *Ticket* que acompaña en su interior. Para FIPA, esos mensajes de solicitud han de ser de tipo REQUEST.

Un efecto similar podría implementarse con las funcionalidades del *Módulo de Notificaciones* de la plataforma JADE. Sin embargo, es un módulo no activado por defecto cuando se lanza la plataforma y por ese motivo, unido al consumo elevado de recursos, no fue empleado; supliendo con comportamientos en los agentes la remisión de mensajes para publicar los *Tickets* en el *Servicio de Tickets*.

5.5.1.7 *Acción 'Repl' en la ontología*

Las acciones *Repl* incluidas en un *Mensaje ACL* denotan la interacción de un *Actor-humano* desde la *Shell Asíncrona* del *Entorno de Desarrollo*, en donde se encuentra realizando: consultas, solicitudes, programación, análisis o depuración del *motor de inferencia* del agente. El término *Repl*, hace referencia al acrónimo (*Read-Evaluation-Print-Loop*) y está íntimamente asociado en Tecnologías de la Información al término *Shell*. La misión de los mensajes con esta acción, es declarar la intención de procesar una sentencia en el *motor de inferencia* del agente y ser informado tan pronto el motor responda con datos o soluciones.

La acción *Relp* posee dos atributos: [:content] y [:language] (Figura 5.2, pág. 77). Las sentencias redactadas por el *Actor-humano* en el Lenguaje de programación del *sistema experto* son introducidas dentro del atributo [:content] y, cuando el *Mensaje ACL* llega a su destino, esas sentencias son interpretadas y evaluadas por el *motor de inferencia*. Esa manera de actuar representa un problema de seguridad grave y motivo suficiente para considerar que, no deben ser empleados estos *Mensaje ACL* en escenarios de producción.

```
(REQUEST
 :sender      (      agent-identifier :name nodoA@POSEIDON:1099/JADE)
 :receiver    (set ( agent-identifier :name nodoA@POSEIDON:1099/JADE)
 :content     "((action
                   (agent-identifier :name nodoA@POSEIDON:1099/JADE)
                     (Repl   :content \"(facts)\"
                             :language CLIPS)
               ))"
...
```

Código 5.13: Fragmento de mensaje ACL con acción Repl

Sin embargo, estos *Mensaje ACL* resultan extremadamente útiles durante la depuración y en las fases previas de construcción de los agentes con motor de inferencia integrado. Los mensajes con acciones *Repl* (Código 5.13) emplean *actos comunicativos* de tipo REQUEST y *protocolo de interacción* tipo *fipa-request* como se muestra a continuación. El agente que inicia esos mensajes es el agente que ofrece, al *Actor-humano*, su interfaz gráfica en modo *Entorno de desarrollo* para facilitar la redacción de las sentencias que han de ser evaluadas por su *motor de inferencia*. Y es, el mismo *agente*, el que responde al *Actor-humano* a través de su interfaz gráfica, aquello que su motor de inferencia ha resuelto entregar como respuesta.

5.5.1.8 *Acción 'Process' en la ontología*

Las acciones de tipo *Process* incluidas en *Mensaje ACL* tienen una misión similar a las acciones *Repl*, no obstante, se trata de mensajes generados por el propio agente-software para transmitir *hechos* o *reglas* a si mismo, o a *agentes* de la misma *aplicación*.

```
(REQUEST
 ...
 :content  "((action  (agent-identifier :name test24@POSEIDON:1099/JADE)
                      (process
                             :language CLIPS
                             :method  ( sequence  LOAD_FILE  test.clp ))
           ))"
 :language    fipa-sl0
 :ontology    Problem-Solvers
```

Código 5.14: Volcado de un mensaje ACL con acción Process en su contenido

Además, las acciones *Repl* incluidas en *Mensajes ACL* permiten evaluar cualquier sentencia redactada por el *Actor-humano* sobre la consola de la *Shell Asíncrona* contra el *motor de inferencia*; pero las acciones de tipo *Process* emplean un *catálogo predefinido* de posibles procedimientos que se especializan en función del *motor de inferencia* seleccionado por el agente. Eso elimina la necesidad de evaluar las sentencias redactadas a 'mano' y abre el camino a emplear las funciones específicas de la API de cada tecnología de *motor de inferencia*.

Utilizar acciones de tipo *Process* genera una barrera de protección natural, un mecanismo de mejora del rendimiento al eliminar la 'evaluación' de sentencias y establecer un modelo normalizado para que los *agentes* puedan *propagar* sus *hechos* hacia otros agentes o, para que los agentes puedan *coordinar* sus memorias de *hechos* y de *reglas* con otros agentes sin intervención del *Actor-humano* y de forma segura y eficiente. En resumen: con acciones *Process* en los *Mensajes ACL* se consigue fiabilidad y seguridad al interponer una relación estricta de posibles actividades para ser ejecutadas por el *motor de inferencia*. Seguridad que se

ve acompañada de la portabilidad que supone contar con un catálogo cerrado de posibles interacciones entre motores de cualquiera de las tecnologías consideradas y la posibilidad de diseñar procesos estándares para la coordinación de *hechos* o de *reglas*.

[Elaboración propia]

Códigos de procedimiento en atributo (:method) de acciones Process

Código	*Descripción de sucesivos parámetros*
LOAD_FILE	Nombre de archivo a cargar.
LOAD_FACTS	Nombre de archivo de hechos a cargar.
LOAD_FROM_RESOURCE	Nombre de archivo de recursos.
LOAD_FROM_STRING	Nombre de archivos de datos CSV.
LOAD_ASSERT_STRING	Carga directa de hechos desde Strings.
LOAD_BLOAD	Restaurar memoria desde el archivo Bin.
LOAD_SLOAD	Restaurar memoria desde archivo plano.
RUN_INFINITELY	Ejecutar hasta el final de activaciones
RUN_NUMBER_OF_CYCLES	Ejecutar un número de ciclos concreto
RUN_ONCE_THEN_BATCH	Ejecutar y devolver el control al Shell
RUN_INNER_SHELL	Ejecutar la Shell interna.
MAKE_RESET	Realizar un reset.
MAKE_CLEAR	Realizar un clear.
MAKE_MEMORY_DUMP	Realizar una copia de seguridad.
MAKE_ASSERT_STRING	Incorporar un hecho desde una cadena.
MAKE_BUILD	Compilar una consulta.
EVAL_COMMAND(*)	Evaluar una sentencia. (por compatibilidad)
SET_INPUT_BUFFER_COUNT	Solicita el número de caracteres tecleados.
APPEND_INPUT_BUFFER	Incorporar a la sentencia.
SET_UNWATCH	No analizar (debugger).
SET_WATCH	Analizar (Facts, Modules, etc.).
GET_FACT_SLOT	Capturar el valor de un Slot.
FACT_INDEX	Desplazar el cursor en la base de hechos.

Tabla 5.4: Códigos empleados en la definición de Acciones Process

La interface `RBEngine` implementada por las clases `RBEngineCLIPS` y `RBEngineJESS`, contiene la descripción del *catálogo predefinido* de posibles peticiones de procesamiento identificadas mediante una etiqueta y una lista de parámetros adicionales (Tabla 5.4, pág. 81).

Los objetos *Process* poseen: [`:language`], [`:content`] y [`:method`]. El atributo [`:content`] no es necesario y se mantiene en la implementación de *Process* para facilitar la conversión directa de sentencias tecleadas manualmente en ensayos previos sobre la *Shell* que finalmente, son reconvertidos a objetos tipo *Process* codificados.

Existen además, *Mensajes ACL* con acciones *Process* que son generados por comportamientos para llevar a cabo tareas de mantenimiento interno (a intervalos regulares) como por ejemplo: generar una copia íntegra de la WM y la KB del *motor de inferencia*, cargar los archivos del *sistema de producción* en la memoria, comprobar *reglas activadas* en la *agenda* del *motor de inferencia* para proceder a su disparo, etc.

5.5.2 Instrumento: Entorno de desarrollo (agente `pocket`)

El *Entorno de desarrollo* es considerado un instrumento al servicio del *ingeniero de software* y del *administrador de sistemas*. Ha sido implementado como un *agente* que extiende la clase [PsNodeAgent], que es la clase empleada para generar los agentes con *motor de inferencia* integrado. Posee una interfaz gráfica extensa, diseñada para permitir el ajuste, la depuración de los *comportamientos* y la gestión de los archivos del *sistema de producción* que se asignan a cada uno de los *agentes*. Durante el diseño de su interfaz gráfica se han incorporado un número elevado de funcionalidades con objeto de facilitar la intervención directa en tiempo de ejecución sobre el *agente* y la observación de la conducta del agente.

Desde el *Entorno de desarrollo* se obtiene una visión de los comportamientos; del orden de compilación, carga y ejecución de los comportamientos; de los efectos que producen dichos comportamientos en el agente al ser incorporados; de las respuestas generadas al activar el *modo depuración* y el *modo verbose* del agente en análisis; de los resultados de la ejecución de los *test* lanzados sobre el agente, etc. El *Entorno de desarrollo* se ha dividido en dos grupos de pestañas que contienen a las pantallas siguientes:

Pestaña de Gestión del agente:

- Pantalla: *Shell-JADE*.
- Pantalla: *Editor de comportamientos* y archivos de inicialización.
- Pantalla: Editor de mensajes y traza de cola de mensajes del agente (sólo gráficos).

Pestaña de Gestión del motor de inferencia:

- Pantalla: *Editor de archivos* del sistema experto gestionado por el agente.
- Pantalla: *Shell Síncrona* entre agente y motor de inferencia local.
- Pantalla: *Shell Asíncrona* (entre agente y motor de inferencia Local o Remoto pertenecientes a agentes de la misma aplicación).

5.5.2.1 *Pantalla: Shell JADE*

[Elaboración propia]

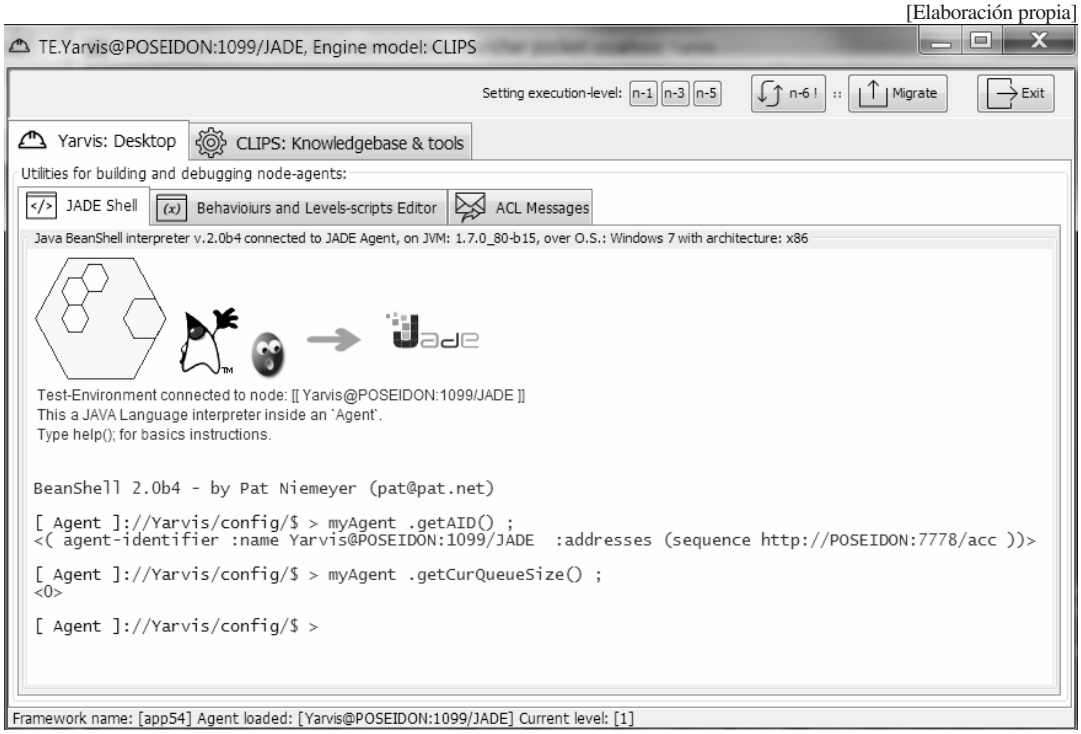


Figura 5.3: Pantalla de Shell – JADE en el Entorno de desarrollo

Ocupa la primera pantalla del *Entorno de desarrollo* (Figura 5.3, pág. 83). Es el componente BeanShell [Niemeyer, 2014] conectado al *agente* para permitir al *Actor-humano* programar en Lenguaje *Java* y tener acceso directo a; los objetos de la API de JADE, métodos y propiedades del *agente*, métodos y propiedades del motor prototipo, etc. Permite realizar cualquier actuación y construir nuevas clases *Java* y por tanto, instanciar nuevos objetos que pueden incorporarse al *agente*, revisar la ayuda *en-línea*, realizar y lanzar *test unitarios* o por componentes, comprobar la cola de mensajes, construir nuevos mensajes, enviarlos, comprobar comportamientos, crear nuevos comportamientos, detenerlos, reactivarlos, etc.

Se han incorporado las siguientes variables: `myAgent` para acceder al objeto *agente*, `innEngine` permite acceso a la API del motor de inferencia, `myBOB` a la lista de comportamientos presentes en el agente, `myEngine` da acceso a la interface pública del motor de inferencia y, `myNode` que permite una visión completa de las propiedades incorporadas al *agente-nodo* para conectarse al sistema de ficheros y a su motor de inferencia. Esas variables se encuentran documentadas en la ayuda *en-línea* accesible mediante la llamada a la función `help();`

La *Shell JADE* posee activadas las funciones de completado automático de comandos, accesible mediante presión de la tecla *Tabulador* a partir del tercer carácter tecleado. Los archivos de sintaxis de los Lenguajes se encuentran en el directorio `/config/` perteneciente al *Directorio de aplicación*. Las librerías de *Java* y de la API JADE son importadas desde el archivo `level.00.bsh` sobre el intérprete y también, se encuentra en dicho archivo; la definición del símbolo del *prompt* empleado en la *shell JADE* más el histórico de comandos tecleados sobre la *shell*, que es accesible con las teclas de dirección arriba y abajo del teclado estándar.

[Elaboración propia]

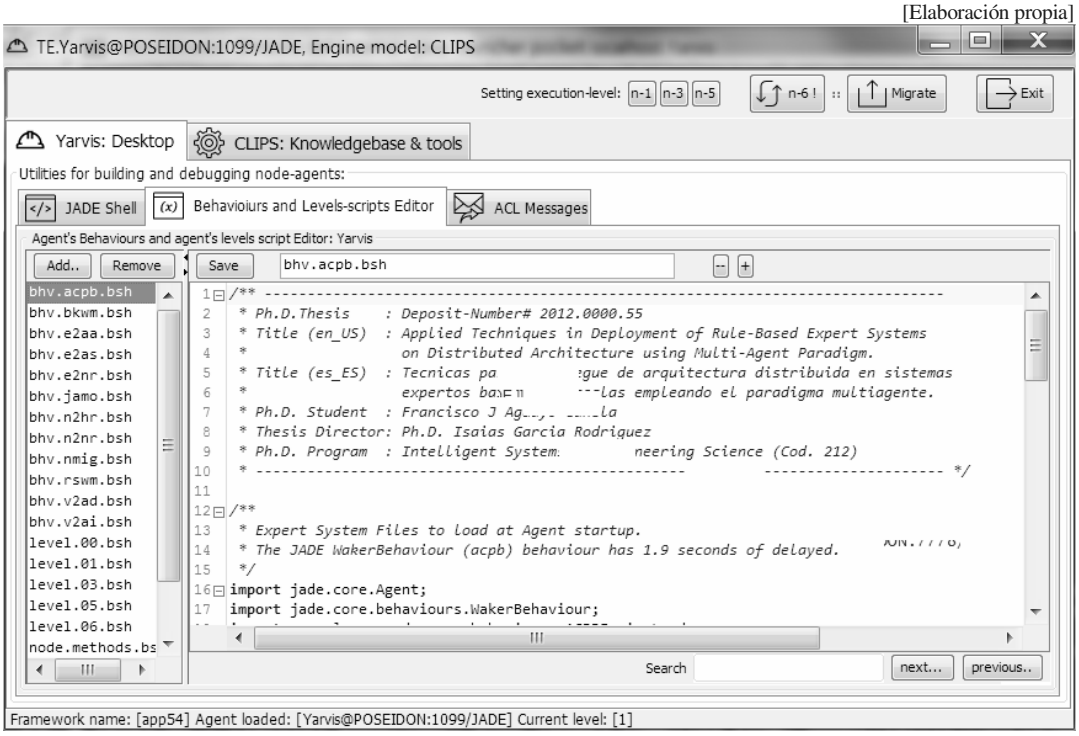


Figura 5.4: Pantalla de Editor de comportamientos y configuraciones

5.5.2.2 *Pantalla: Editor de comportamientos y configuraciones*

La pantalla del editor de los archivos de comportamientos externalizados y de los archivos de configuraciones de cada *agente-nodo* (Figura 5.4, pág. 84) es una pieza de software versátil que emplea el editor *RSyntaxTextArea* para aumentar la comodidad y la experiencia de usuario durante las labores de análisis y de modificación de los comportamientos sobre el mismo *Entorno de desarrollo*. Aporta resaltado de sintaxis y completado automático de funciones, etc. La zona izquierda contiene un listado de los archivos del directorio `/config/` del *agente-nodo*. Una vez seleccionados se cargan en el editor en la zona derecha.

5.5.2.3 *Pantalla: Editor y Traza de mensajes ACL*

Esta pantalla del *Entorno de desarrollo* permite el seguimiento visual de los acontecimientos en la cola de mensajes del agente o construir, manualmente, los mensajes mediante la interfaz de componentes de Acklin B. V., en concreto ACLGui y ACLTrace, que se encuentran en el agente *testAgent* del paquete `jade.tools.testagent` que acompaña a la distribución estándar de JADE. Las funciones de esta pantalla se encuentran parcialmente implementadas.

[Elaboración propia]

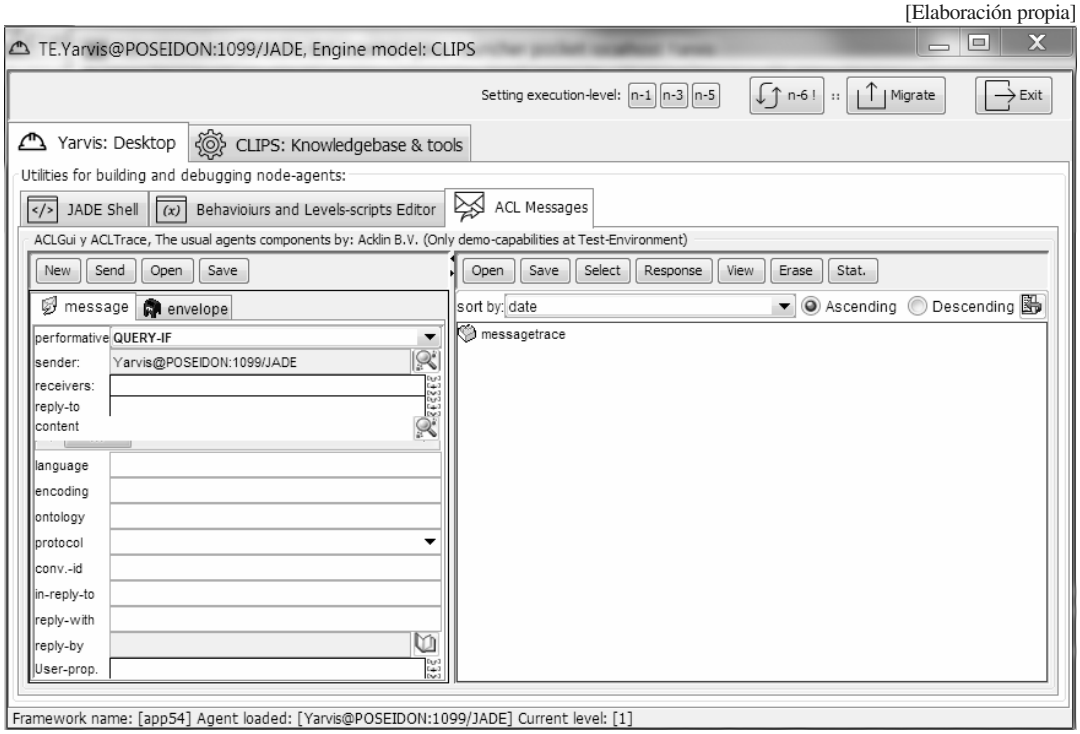


Figura 5.5: Pantalla de Gestión y Traza de mensajes ACL

5.5.2.4 *Pantalla: Shell Asíncrona*

Implementa el modelo de comunicación entre *agente* y su *motor de inferencia*. De manera simultánea, permite la comunicación entre los *agentes* de la aplicación para llevar a cabo labores de control, propagación, comprobación o coordinación de inferencias inter-agentes. Emplea mensajes ACL con la ontología *Problem-Solvers* para construir los contenidos con *acciones* de tipo *Repl* que el *agente* remite hacia si mismo contra su motor local o, hacia agentes que pertenecen a la *Directorio de aplicación*, una vez seleccionados en la lista que aparece a la izquierda de la pantalla (Figura 5.6, pág. 86).

Se ha denominado *Shell Asíncrona* porque su actuación es diferida, en el sentido que, tanto las sentencias dirigidas al *motor de reglas* como las respuestas que dicho motor devuelve, son transmitidas mediante mensajes tratados por los agentes participantes. Esta pieza de software utiliza el instrumento denominado *protocolo de interacción interna de doble secuencia* (PII2S) y permite al *Actor-humano* (ingeniero de conocimiento, desarrollador o experto del dominio), acceso al *motor* desde el *agente* en tiempo de ejecución sin generar bloqueos; ni en la ejecución de los comportamientos del agente, ni en la ejecución del motor.

En la zona inferior aparecen las respuestas emitidas que son devueltas por el *motor de inferencia* y capturadas por el *agente-nodo*. Desde el *Entorno de desarrollo* se dialoga con el motor integrado en el *agente* seleccionado en la lista del área de agentes. La opción por defecto es remitir las sentencias hacia el propio motor del agente-nodo que está representado por [`NombreDelAgente (itself)`]. Para facilitar la escritura, se ha asociado la combinación de teclas Mayúsculas+*Ctrl*+*Enter* como final de la redacción de las sentencias.

[Elaboración propia]

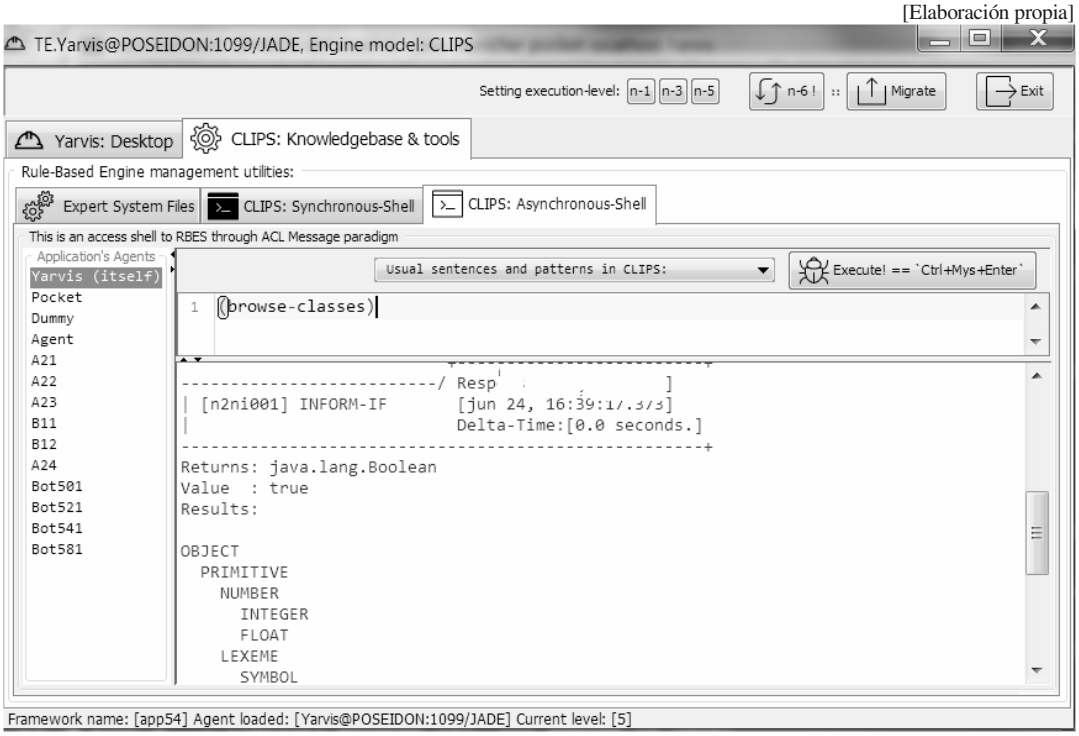


Figura 5.6: Pantalla de Shell Asíncrona

Con esa combinación, (o mediante presión del botón [Execute]) se dispara el evento de evaluación y, la sentencia redactada sobre el área de redacción, es enviada al agente seleccionado en forma de *Mensaje ACL*. Una vez recibido por el agente correspondiente, remitido a su motor interno y concluido dicho procesamiento en el motor de inferencia; es devuelta la respuesta como un nuevo mensaje ACL de tipo Notificación, asociado al *hilo de la conversación* iniciada por el agente original de la petición.

5.5.2.5 *Pantalla: Editor de archivos del sistema de producción-local*

Desde el *Entorno de desarrollo* se incorpora esta pantalla para facilitar el tratamiento, la incorporación y la edición de los archivos que componen cada fragmento de *sistema de producción* que se asigna a cada *agente*. Permite llevar a cabo la edición de los archivos que componen el *sistema experto* original, lo que facilita el proceso de fragmentación de las capacidades de inferencia de cada agente y abre el camino hacia la especialización del agente y la distribución de las funciones del *sistema de producción* original en módulos. Utiliza la librería *Java* de edición RSyntaxTextArea, que soporta auto-completado, funciones y patrones para edición inteligente, resaltado de la sintaxis en CLIPS y *Jess*, marcas para replegar código, numeración de líneas, etc.

A la izquierda se listan los archivos del directorio `/engine/` que pueden ser editados en la parte derecha de la pantalla una vez seleccionados. Los archivos de resaltado de sintaxis del Lenguaje CLIPS y *Jess* se encuentran, por convenio, en el directorio `/config/` del *Directorio de aplicación*.

[Elaboración propia]

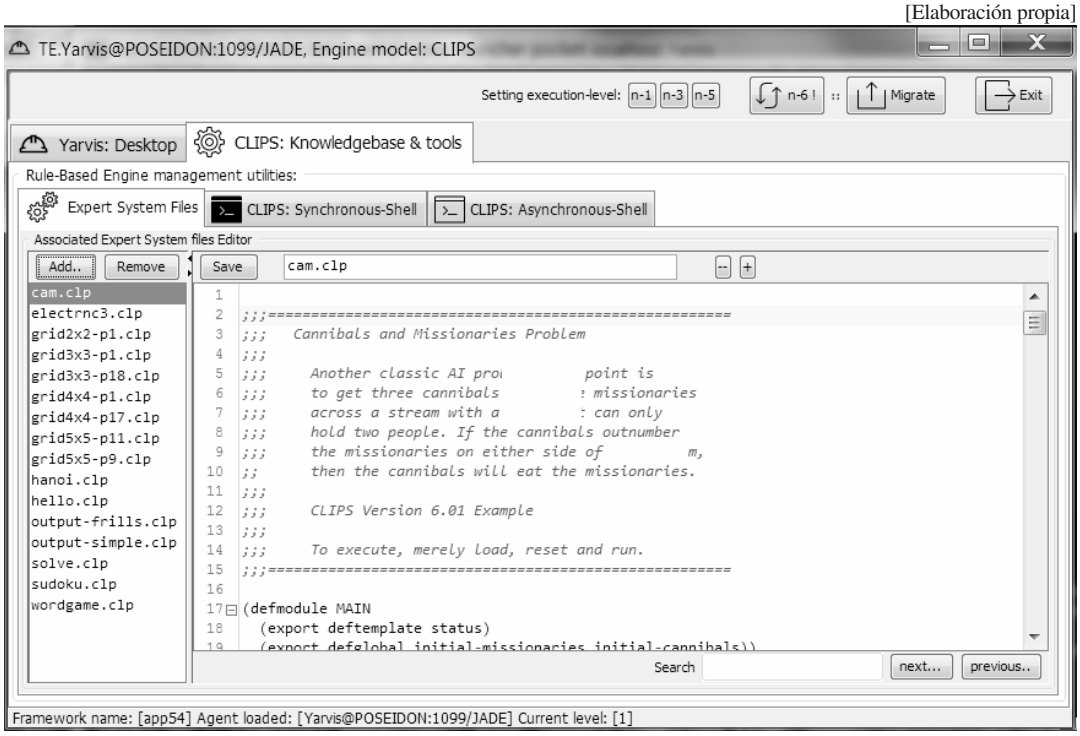


Figura 5.7: Pantalla de gestión de archivos del sistema experto local

5.5.2.6 *Pantalla: Shell Síncrona*

Permite al *Actor-humano*, llevar a cabo una interacción directa con el *motor de inferencia* CLIPS o *Jess* del *agente* (*véase* Figura 5.9, pág.89). Con esa afirmación se denota que, la conexión entre el hilo de ejecución del entorno gráfico *Java*, o *Dispatcher* de *Eventos* de *Java*, es el elemento que mantiene en espera la ejecución del *agente* hasta recibir la respuesta del motor de inferencia. Su código, en el caso de emplear un motor tipo CLIPS, es una adaptación de los trabajos de la *shell de consola gráfica* de CLIPS conectada a CLIPSJNI realizados por *Gary Riley* [Riley, 2008] y disponibles en *sourceForge* [Riley, 2016b].

La *Shell Síncrona* es una conexión directa desde la interfaz gráfica del *Entorno de desarrollo* hasta el *motor de inferencia* interno del agente. Su funcionamiento se basa en utilizar el *Router* de salida de respuestas del motor de inferencia y redirigir sus resultados hacia los eventos del componente gráfico `javax.swing.JTextArea` para emular una *Shell*.

Destacar que, en estados iniciales de la investigación (*véase* Figura 5.8, pág.88), se llegó a considerar a esta *Shell Síncrona* como la *pieza clave* para la comunicación agente-motor pero, fue un error. Esa conexión entre *agente* y *motor* funciona en realidad porque se explota el hilo de ejecución del *gestor de eventos* interno de *Java* y porque se bloquea la interacción hasta que el motor devuelve el control de la ejecución al interfaz gráfico. Lo que representa un modelo de integración *agente-motor* que no es admisible en un *entorno distribuido*. En cualquier caso, posee demostradas capacidades para la depuración en estados iniciales durante el modelado del *agente* cuando aún, no se requiere interactuar con otros agentes de la aplicación. Esta pantalla puede activarse o desactivarse del *Router* del motor de inferencia, desde

la pantalla de *Shell JADE* con las sentencias del Código 5.15, motivo por el que se ha mantenido como parte del *Entorno de desarrollo* para agentes con motores tipo CLIPS o *Jess*.

[Elaboración propia]

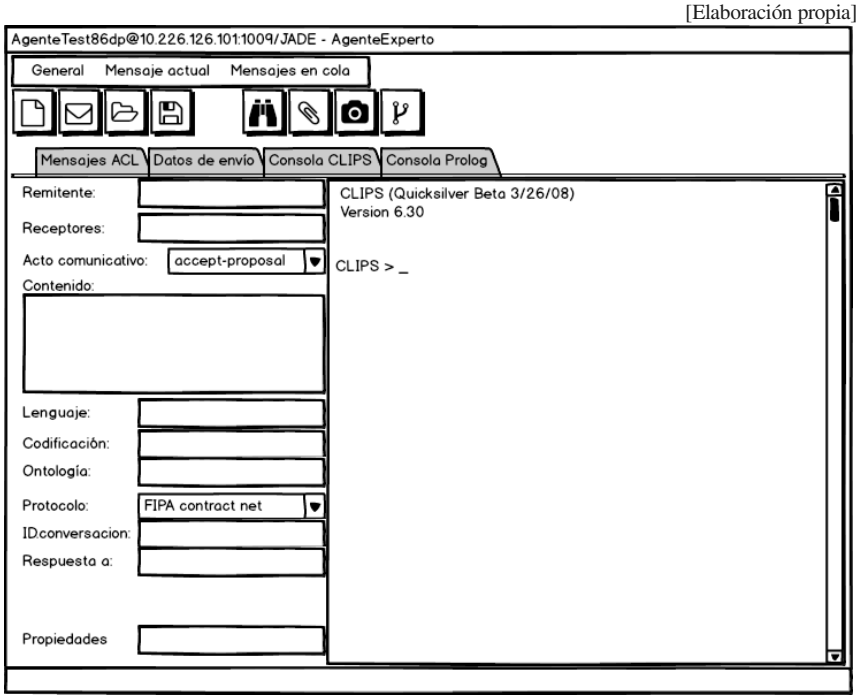


Figura 5.8: Primer boceto de la interfaz gráfica de agente con motor integrado

[Elaboración propia]

```
myEngine .addRouter( myShell );      Se redirige hacia la Shell Síncrona.
myEngine .addRouter( myEngine );     Se redirige hacia el propio objeto Interface.
```

Código 5.15: Técnica empleada para conectar/desconectar la Shell Síncrona

5.5.3 Instrumento: *Protocolos de interacción*

FIPA especifica la estructura recomendada para las secuencias seguidas en los diálogos entre agentes en la recomendación de la Librería de *Protocolos de Interacción* [FIPA00025]. El objetivo de lograr una *solución para la sincronización* entre los motores de inferencia de dos o más agentes, se alcanza mediante la utilización de un protocolo de interacción.

En las especificaciones FIPA sobre agentes inteligentes se establece que, un agente puede verse envuelto en múltiples diálogos con otros agentes de manera simultánea. A ese concepto de *diálogo* se le conoce con el término *conversation* y es utilizado para referirse a cada una de las instancias (tomadas como conjuntos independientes) de las interacciones.

En el transcurso de la investigación destacaron dos modelos de conversación. (***a***) El primero compuesto por dos secuencias; una entre los *agentes* participantes y otra oculta entre esos agentes y sus motores de inferencia integrados. Ese modelo ha dado lugar a la implementación del *protocolo de interacción de doble secuencia*. (***b***) El segundo modelo de conversación es mantenido entre *agentes* y el agente que presta el *Servicio de Tickets*, empleando un protocolo estándar aportado por JADE. Ambos tipos de conversación pertenecen al conjunto que FIPA denomina protocolos de interacción de tipo *FIPA-Request*.

[Elaboración propia]

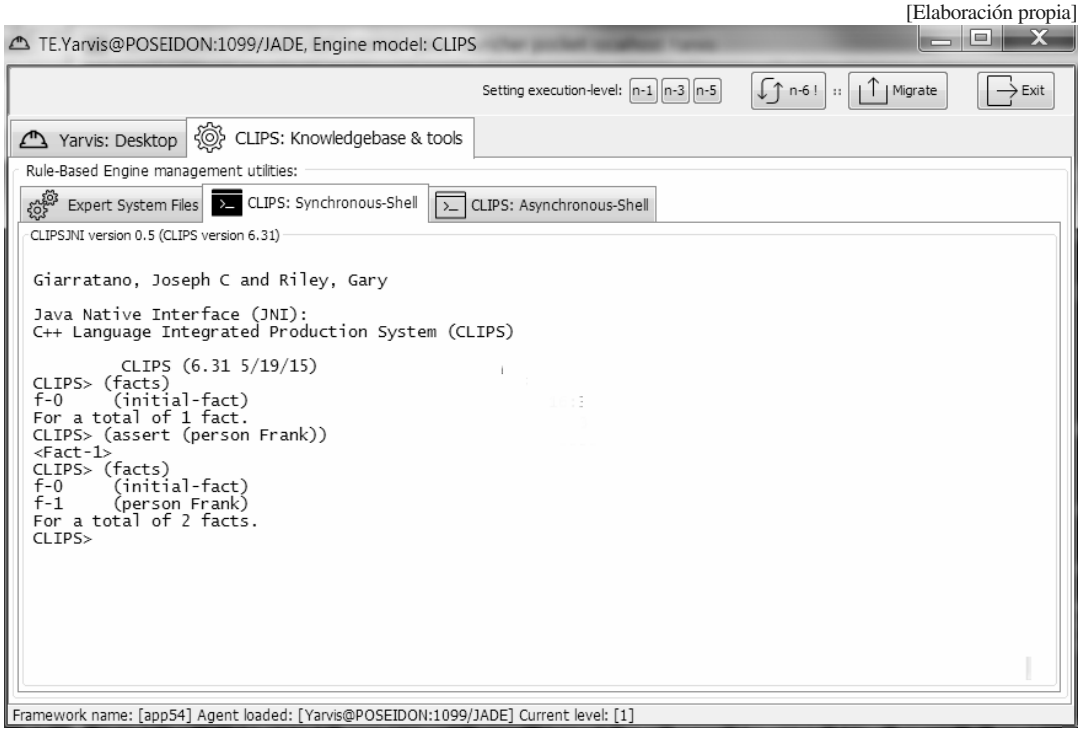


Figura 5.9: Pantalla de Shell síncrona a motor CLIPS

5.5.3.1 *Protocolo de Interacción Interna de doble secuencia*

Cuando el *agente* recibe una petición para ser procesada por el motor de inferencias, el *agente* decide dentro del mismo acto comunicativo, si es pertinente esa petición respondiendo por un lado con un mensaje AGREE al remitente por y por otro, con un REQUEST-WHEN que remite hacia si mismo y que posteriormente será evaluado por su motor interno.

Se inicia así, la primera secuencia de un protocolo desdoblado internamente pero, que presenta el formato externo de un protocolo de interacción estándar *FIPA-Request*. La siguiente respuesta es una *notificación* con los resultados devueltos por el motor dentro de un mensaje INFORM dirigido al remitente.

Las abreviaturas N2NI y N2NR (Figura 5.10, pág. 90), son los nombres de las variables asignadas a los comportamientos que dan forma a este protocolo de interacción secuenciado. Representan los roles de inicio y de respuesta que asumen los *agentes* durante las conversaciones. El término **N2NI** es acrónimo de *Node to Node Initiator*, y corresponde a cada una de las instancias ($N2NI_{001}$, $N2NI_{002}$… $N2NI_{n}$) de comportamientos, creadas por el agente para dar comienzo a nuevas conversaciones con los *motores de inferencia* de otros *agentes*. Cuando el *Actor-humano* emplea la *Shell Asíncrona* para dialogar con el motor interno del propio *agente* se produce un *Caso de Uso* particular de interacción en el que se cumple que:

$$Engine\ from\ Agent_1 \equiv Engine\ form\ Agent_2.$$

En tal caso, el agente establece conversaciones consigo mismo siguiendo el mismo protocolo de interacción de doble secuencia. Esta particularidad fue la que posteriormente llevó a in-

corporar sobre la pantalla de *Shell Asíncrona* (Figura 5.6, pág. 86), la lista con *agentes-nodo* del *Directorio de aplicación* el término (*Itself*) para destacar con claridad que se dialoga con el motor del *agente* abierto en el *Entorno de desarrollo*.

[Elaboración propia]

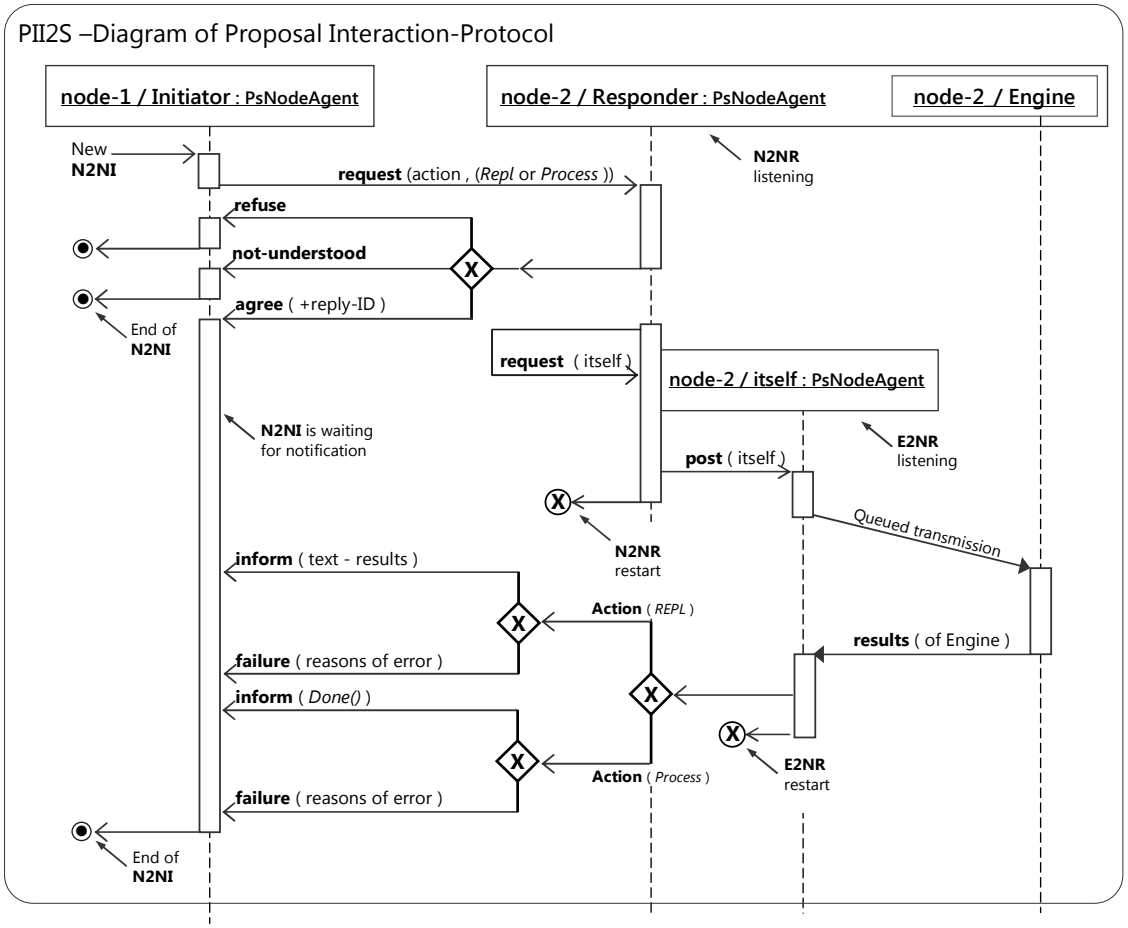


Figura 5.10: Diagrama protocolo de interacción PII2S

El término **N2NR** (*Node to Node Responder*) (Figura 5.10, pág. 90) representa a la única instancia de ese comportamiento que existe en todos los *agentes* y que permanece siempre a la espera de recepción de mensajes ACL emitidos por los comportamientos N2NI iniciadores. El comportamiento N2NR o *respondedor*, aporta a los *agentes* la conducta de *Respuesta Racional* (RE) frente a las propuestas de conversación emitidas por otros *agentes*. Con ese modelo se obtiene el patrón propuesto por FIPA para protocolos de tipo *FIPA-Request* mantenido entre los agentes.

Para capturar los mensajes ACL emitidos por el *agente-nodo* hacia su motor de inferencia se incorpora un comportamiento respondedor denominado **E2NR** (*Engine to Node Responder*) como parte del protocolo. Ese comportamiento en el diagrama, representa a la única instancia de comportamiento de tipo *Respondedor* que permanece siempre a la escucha de los mensajes ACL que un *agente* puede dirigir a su propio motor. Esos mensajes aparecen en la cola interna de mensajes y son gestionados por E2NR, evaluados por el *motor de inferencia* y, una

vez capturadas las respuestas generadas por el motor, es el comportamiento E2NR el que compone el mensaje de notificación adecuado de tipo INFORM o FAILURE y lo remite al agente solicitante, dando por concluida la conversación.

Al implementar el protocolo de interacción de *doble-secuencia* se desencadenan los siguientes efectos en los *agentes* que son trascendentes, tanto en lo relativo al mantenimiento de sus propiedades originales de agentes desplegados en una plataforma multiagente, como en lo relativo a su condición de *objetos Activos* de *Java* que poseen un objeto *motor* integrado:

- El *agente* puede seguir recibiendo y despachando mensajes de todo tipo, incluidos los mensajes con *acciones Repl* o *Process*, porque la recepción de mensajes es gestionada por el comportamiento N2NR que responde de inmediato al remitente y delega la notificación de la respuesta en el segundo comportamiento respondedor E2NR, volviendo a quedar disponible. Esto hace posible iniciar y mantener múltiples conversaciones entre *agentes* debido a que el comportamiento respondedor N2NR, siempre es capaz de despachar nuevas consultas y gestionar los códigos de identificación de cada conversación.
- El *agente* no entra en estado de bloqueo porque, las peticiones que llegan al motor se transforman en mensajes ACL, y son colocadas en la propia *cola de mensajes* del *agente* donde permanecen a la espera hasta su gestión y tratamiento por parte del *respondedor* E2NR. Este mecanismo no bloquea al *agente* y no genera disminución en el rendimiento del agente.
- El *agente* no entra en estado de bloqueo porque, la ejecución del disparo de reglas o la realización de cálculos por descartes de *reglas* acontecidas en el *motor de inferencia* es gestionada dentro de uno de los estados pertenecientes al comportamiento E2NR. Ese estado fue denominado *E2NRProcessState* y es ejecutado sobre un *ThreadedBehaviour* de JADE, que libera el *programador de tareas* interno del agente y permite que se ejecute el próximo comportamiento siguiendo el modelo *cooperativo*. Además, el *motor de inferencia* se ejecuta a impulsos gestionados por objetos RBEngineCLIPS o RBEngineJESS sobre un hilo de ejecución *subordinado* al hilo de ejecución del *agente*.
- Al emplear este protocolo de interacción entre *agentes*, es posible establecer la comunicación con el *motor de inferencia* interno o con el *motor de inferencia* de un *agente* localizado en un contenedor remoto con la misma pauta de interacciones. Para que este efecto ofrezca una mayor seguridad se incorpora un concepto adicional denominado *lista de confianza.* Ese concepto ha sido implementado en el *protocolo de interacción* y permite comprobar si el *agente* solicitante (*Iniciador*), pertenece a los agentes del *Directorio de la aplicación*. Para ello se utiliza la variable `ps.framework.node.serie` ubicada en el archivo `framework.properties`.

5.5.3.2 *Protocolo: FIPA-Request aplicado al Servicio de Tickets*

Los diálogos posibles entre *agentes-nodo* y el *Servicio de Tickets* son tratados mediante la extensión del protocolo de interacción estándar FIPA denominado protocolo *FIPA-Request* como se observa a en el diagrama de interacción de la Figura 5.11, (*véase* pág.92).

[Elaboración propia]

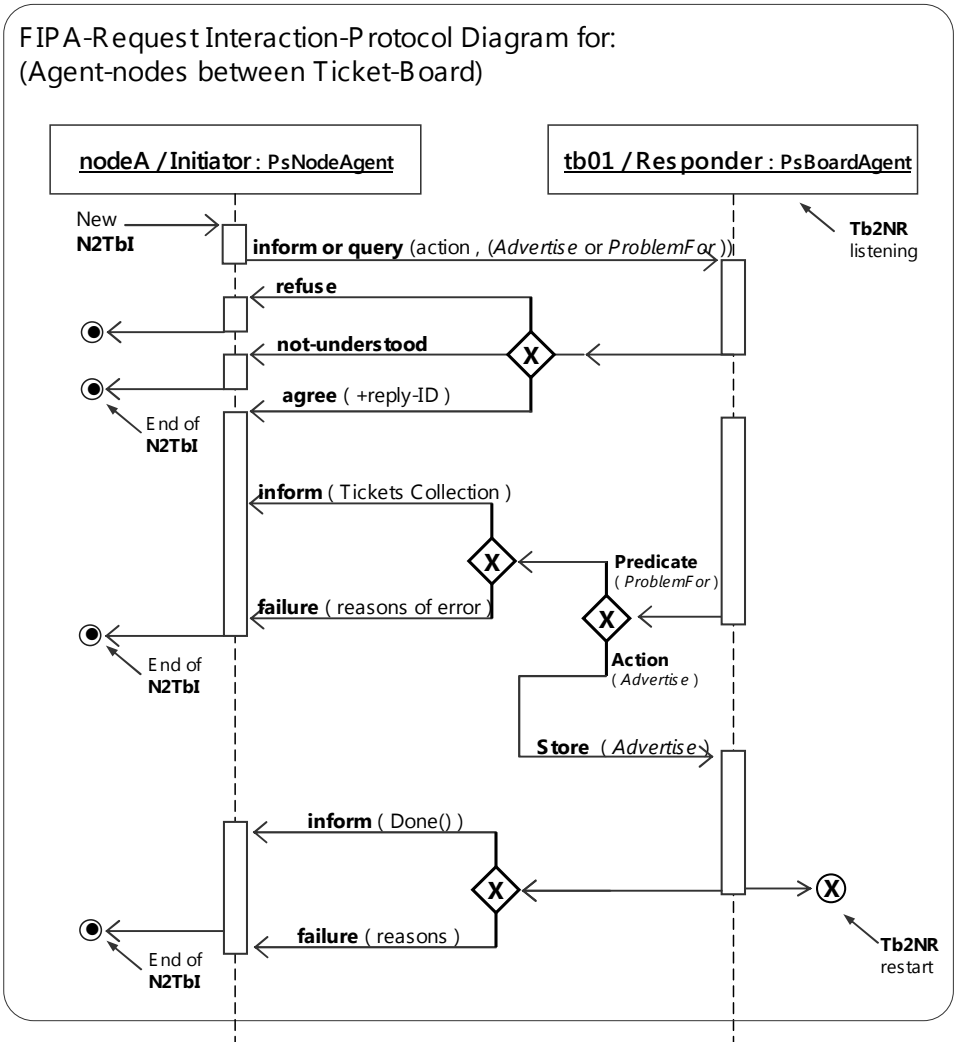


Figura 5.11: Diagrama de protocolo de interación con agentes TicketBoard

Los términos N2TbI y Tb2NR corresponden con instancias de comportamientos que implementan el protocolo de interacción *FIPA-Request*. N2TbI (*Node to Ticket-Board Initiator*) son instancias ($N2TbI_{001}$, $N2TbI_{002}$… $N2TbI_n$) de comportamiento creadas por el *agente-nodo* para dar comienzo a nuevas conversaciones pautadas con el agente que presta el *Servicio de Tickets*. El comportamiento denominado Tb2NR (*Ticket-Board to Node Responder*) es una instancia única del comportamiento respondedor que permanece activa en el agente *Ticket-Board*, destinada a ofrecer *respuestas racionales* a las solicitudes de consulta emitidas por *agentes-nodo* para localizar *Tickets* con datos sobre problemas que están pendientes de resolución o solicitudes de catálogo de *Tickets* (con la *acción Advertise*) con datos de las soluciones (parciales o finales) que son alcanzadas por los *agentes*.

La implementación del comportamiento Tb2NR respondedor permite discriminar, en función de la acción solicitada por el *agente-nodo*, si ha de ejecutar un proceso de catálogo en su base de datos de objetos *Tickets* o, sólo tiene que interrogar al repositorio de *Tickets* y responder

con un mensaje ACL que incluye el listado con los objetos *Tickets* presentes que cumplen las condiciones solicitadas.

5.5.4 Instrumento: Documentación en formato JavaDoc

[Elaboración propia]

Figura 5.12: Portada inicial de la documentación incorporada al Framework

Como se emplean varias tecnologías y Lenguajes de programación de manera simultánea en el modelado y ajuste de los *agentes*, se adopta la decisión de incluir *por-defecto* la documentación mínima e imprescindible en el directorio `/var/` de cada *Directorio de aplicación* en el instante de su creación (*ver* Código 5.2, pág. 71). Esa opción puede eludirse con la indicación de parámetro adicional [`--no-doc`].

```
[ ~/pruebas/]#  java -jar dps-frameworkBuilder-0.9.1.jar --new=app51  --no-doc
```

El volumen de la documentación incorporada no supera los (14 MBytes) y puede resultar excesivo en comparación con el volumen que ocupan las librerías, pasarelas, plataforma de agentes y los propios agentes que componen un *Directorio de aplicación* en su estado inicial

(*aprox.* 5 MBytes). Sin embargo, los beneficios que aporta la documentación, en prácticamente todas fases de análisis, diseño y despliegue de las aplicaciones a transformar, justifican su inclusión como elemento de carácter instrumental.

Se encuentra por convenio en: `directorio-de-aplicación/var/doc/dps/index.html` (véase, Tabla 5.6, pág. 103). Contiene la información normalizada en formato *javadoc* sobre las *utilidades auxiliares*, los instrumentos, las soluciones propuestas y los *agentes* implementados. Incluye también las Guías de usuario y la documentación de las API de CLIPSJNI, *Jess*, SWI-Prolog y del intérprete *Java* BeanShell, al que se une el manual de CLIPS, *Jess* e HyperSQL, este último para acceder a la base de datos *de Tickets* ubicada en el subdirectorio `var/data/` de los nodos asociados a agentes [`PsBoardAgent`]. El método para acceder a este material es mediante un navegador Web estándar.

5.6 Relación de *técnicas* aportadas

Las *Técnicas* engloban a los segmentos de código que han sido creados para ofrecer soluciones concretas a problemas software en los agentes. Son entonces, productos que no se ejecutan de manera independiente —como lo hacen las *utilidades auxiliares* o como algunas de los elementos instrumentales— pero que se encuentran integrados en el código del *agente*. Las *técnicas* (Columna 3ª, Tabla 5.1, pág. 65) se han clasificado por su funcionalidad en:

(**1**) Técnicas para integración de un intérprete *Java* en el agente.

(**2**) Técnicas para externalización de los comportamientos del agente.

(**3**) Técnicas para control de la inicialización del agente.

(**4**) Técnicas para elaboración de *pruebas unitarias* en el agente.

(**5**) Técnicas para integración del motor de inferencia en el agente.

(**6**) Técnicas para migración del *agente* a contenedores remotos de la plataforma.

5.6.1 Técnica: Integración de un *Intérprete de Java*

Para facilitar la incorporación de capacidades específicas en cada instancia de *agente*, se incorpora un Intérprete de *Java* en los agentes con motor de inferencia integrado. Esas capacidades de compilación se obtienen de la librería *BeanShell* [Niemeyer, 2008] como sugieren los autores *Beneventi* y *Poggi* en [Beneventi et al., 2004].

BeanShell posee tres modos de funcionamiento. Un modo *No-gráfico* donde las sentencias en Lenguaje *Java* son interpretadas con llamadas al método *.eval()* del objeto *BeanShell*. Ese modo es empleado por el *agente* cuando es desplegado para operar en entornos de producción porque, ya no es necesaria una interfaz gráfica pero sí, la capacidad de interpretación del Lenguaje *Java* para leer e interpretar los comportamientos externalizados.

Un segundo modo *Gráfico*, que posee una *consola* para permitir la interacción del usuario y redactar en pantalla el código que es interpretado línea a línea por el intérprete de *Java*. Ese modo ha sido utilizado en el *Entorno de desarrollo*, presentando una amplia zona de edición,

destinada a facilitar la *programación orientada a agentes* y a simplificar el ciclo de análisis, ensayo, error y modificación sobre el código ejecutado por los *agentes.*

Y un tercer modo que permite conectar al intérprete *BeanShell* a la utilidad Shell, vista en apartados anteriores, e inyectar, sobre dicho intérprete del terminal, las variables trascendentes para el desarrollo de aplicaciones como: `frameworkVarPath`, `frameworkLogPath`, `frameworkRootPath`, `frameworkConfigPath`, etc., a las que se suman las sentencias `import` de la API de JADE y la ubicación en función del hardware de las librerías de pasarela *Java* ofrecidas por CLIPS.

También se ha cuidado en lo posible, la apariencia del *prompt* de las *consolas* del Intérprete de *Java* con técnicas incorporadas en los archivos de configuración. Con esas soluciones, el *agente* desplegado en el *Entorno de desarrollo* (Figura 5.13, I y II) o en la pestaña homóloga que presenta el agente que asume el *Servicio de Tickets*, informan al usuario de cuál es la identidad del agente desplegado seleccionado en cada instante, desde la consola *Shell JADE*.

[Elaboración propia]

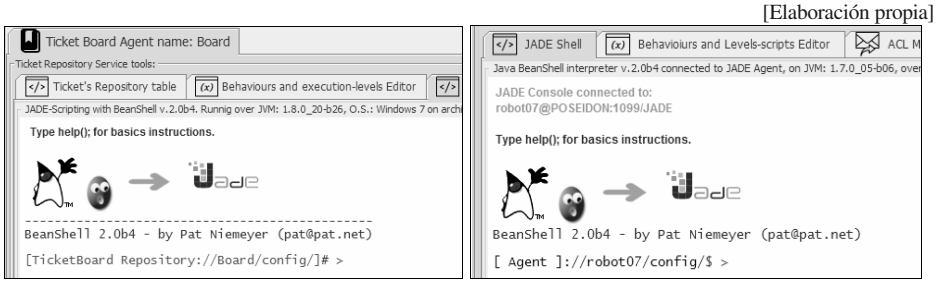


Figura 5.13: Prompt de consola empleados en Shell-JADE por agentes

5.6.2 Técnica: Externalización de los comportamientos

Las técnicas de externalización de comportamientos son la solución software propuesta para afrontar la necesidad de especialización de cada *agente* que compone la *aplicación*. No sería viable construir clases distintas para generar cada instancia particular de agente con sus conductas concretas para componer la *aplicación* basada en *sistemas de producción*. Además, los trabajos de investigación dedicados a emplear archivos *.properties* externos para diferenciar y gestionar a cada instancia de agente, no aportaron la flexibilidad suficiente para permitir al *desarrollador* ejercer un control flexible sobre los comportamientos de los *agentes*.

En resumen, como los escenarios que se producen cuando se acomete la transformación de una aplicación al paradigma multiagente requieren gran flexibilidad, las técnicas implementadas tenían que orientarse a ofrecer *potencia de modelado* en las manos de los arquitectos de software y los desarrolladores. La solución propuesta, para incorporar esa flexibilidad, son las técnicas de externalización de comportamientos.

Consisten en dotar al intérprete de *Java*, ya incorporado en esos agentes siguiendo la técnica descrita en el apartado anterior, de procedimientos y referencias al *sistema de ficheros*, para que los *comportamientos* puedan ser compilados desde archivos en tiempo de ejecución del agente y puedan ser cargados en el *planificador de tareas* en los instantes apropiados para evitar colisiones o incoherencias.

[Elaboración propia]

Ejemplo de comportamiento externalizado en archivo de clase, compilado e incorporado al agente:

```
//////////////////////////////////////
// BeanShell Class:
source(   "class.BKWMFSMBehaviour.bsh"  );

// Without second parameter, by default = 15 minutes
// bkwm = new FSMBackupWorkingMemory( myAgent  );

bkwm    = new BKWMFSMBehaviour( myAgent, 60L );
bkwm.setBehaviourName( "bkwm");
...
myAgent .addBehaviour( bkwm );
```

Código 5.16: Cargador de comportamiento externalizado en tiempo de ejecución

Estas técnicas generaron una parte importante del conjunto de *convenios* adoptados para el *Directorio de aplicación*, en donde existe un subdirectorio `/nodes/AgentName/config/` en todos los agentes, que contiene los archivos de construcción de clases y las declaraciones de objetos comportamiento externalizados que se incorporan en tiempo de ejecución. La nomenclatura finalmente adoptada para referirse a los archivos de clases y a los archivos de carga de objetos comportamiento del agente se listan en la Tabla 5.5 adjunta:

[Elaboración propia]

Clases de comportamientos convertidas a archivos de script externos	
Tipo de agente afectado:	***Nombre del script adaptado y objetivo del comportamiento***
Instancias de clase: `PsNodeAgent`	`bhv.acpb.bsh` Agent Capabilities Loader Behaviour `bhv.bkwm.bsh` Backup Working Memory Behaviour `bhv.rswm.bsh` Restore Working Memory Behaviour `bhv.jamo.bsh` JADE-Management-ontology Actions Manager `bhv.nmig.bsh` Node Migration Behaviour `bhv.n2nr.bsh` Node To Node Responder Behaviour `bhv.e2nr.bsh` Engine To Node Responder Behaviour `bhv.v2ad.bsh` Visitor To Agenda Daemon Behaviour `bhv.v2bd.bsh` Visitor To Board Daemon Behaviour
Instancias de clase: `PsBoardAgent`	`bhv.bcpb.bsh` Board Capabilities Loader Behaviour `bhv.bktb.bsh` Backup TicketBoard Behaviour `bhv.rstb.bsh` Restore TicketBoard Behaviour `bhv.b2nr.bsh` Board To Node Responder Behaviour

Tabla 5.5: Nomeclatura empleada en archivos de carga de comportamientos

5.6.3 Técnica: Control de inicialización de los comportamientos

Cada instancia de *agente* en una aplicación distribuida es singular por las funciones y por el *rol* que asume frente a los demás agentes que conforman la aplicación. Esas diferencias se modelan mediante la selección de aquellos *comportamientos* que han de activarse durante la carga y mediante la modificación de sus archivos de configuración, alterando y ajustando [`node.properties`] hasta que se logra que el agente exponga la conducta buscada.

Debido al aumento del número de *comportamientos* que modelan la conducta de un *agente-nodo*, se hizo necesario un *sistema de control* que facilitara el modelado de la conducta. Y más importante si cabe, era necesario fragmentar la inicialización de los comportamientos en los agentes para poder depurar por un lado, a los propios comportamientos y por otro, para

lograr la configuración de la conducta final expuesta por cada instancia de *agente* al observar el efecto real de sus interacciones con los otros agentes de la *aplicación*.

Las técnicas propuestas para fragmentar la inicialización se encuentran en los archivos siguientes; (1) archivo de funciones de utilidad [`node.methods.bsh`]; (2) en el archivo [`framework.methods.bsh`] y (3), en los archivos [`level.00.bsh`] hasta [`level.06.bsh`] que contienen procesos a ejecutar en cada nivel de ejecución. Se detallan seguidamente los significados asociados a cada *nivel*:

Nivel-0: El *agente* acaba de ser incorporado a la plataforma JADE y su estado es *activado*. Sólo entonces, se carga y se ejecuta, el archivo [`level.00.bsh`].

Nivel-1: El *agente* ha concluido el nivel-0 y ejecuta el archivo [`level.01.bsh`] destinado a cargar comportamientos de carácter basal para el agente, como por ejemplo, cargar las capacidades o restaurar la *memoria de trabajo*.

Nivel-3: El *agente* ha concluido el nivel-1 y procede a la *activación* de los comportamientos previamente cargados. Se ejecuta entonces [`level.03.bsh`].

Nivel-5: El *agente* ha concluido el nivel-3 y procede con la *activación* de los comportamientos solicitados. Se ejecuta entonces [`level.05.bsh`]. Al concluir, el *agente* se considera operativo y *en-servicio*.

Nivel-6: Se trata de un nivel ejecutado sólo a petición del usuario. Consiste en un conjunto de instrucciones contenidas en el archivo [`level.06.bsh`], que provocan *reinicio en caliente* (`restart`) del agente, esto es, el agente no es eliminado de la plataforma, tan sólo se detienen sus comportamientos *activos* y a continuación, son destruidos. Finalizado ese proceso, se reinician los *scripts* de inicialización desde el *nivel-0* donde, vuelven a ser creados y cargados los comportamientos en cada uno de los niveles superiores según la configuración que se haya establecido.

[Elaboración propia]

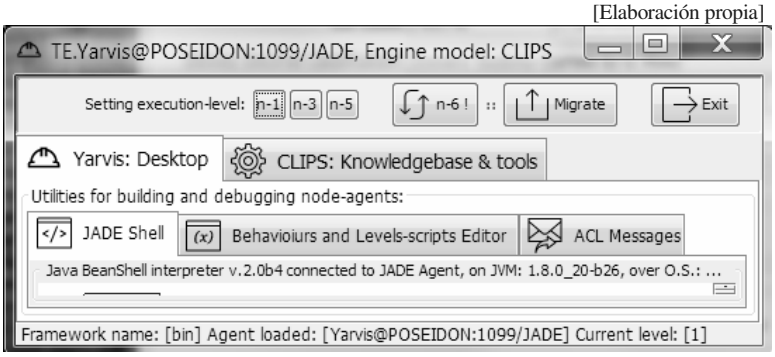


Figura 5.14: Detalle del nivel de ejecución en barras; de botones y de estado

Desde el *Entorno de desarrollo* se puede modificar el *nivel de ejecución* en la *barra de botones* de la pantalla principal del agente. También se refleja sobre la *barra de estado*, donde se actualiza el valor del *nivel de ejecución* con carácter informativo. En la Figura 5.14, se observa un agente denominado [`Yarvis`], en nivel de ejecución igual a [`1`]. El botón [`n-6!`] ejecuta un [`reset`], eliminando todos los comportamientos y mensajes del agente.

5.6.4 Técnica: Elaboración de *pruebas-unitarias*

La técnica propuesta para realizar pruebas unitarias en *agentes* se incorporan desde la pantalla *Shell JADE*, accesible cuando el *agente* se encuentra desplegado en el *Entorno de desarrollo*. Con excepción de los agentes que asumen el *Servicio de Tickets* [`PsBoardAgent`] y de los agentes creados a partir de instancias de la clase [`PsDummyAgent, PsAgent`] ya que incorporan *de facto* la pantalla *Shell JADE* para realizar pruebas a medida.

5.6.5 Técnica: Integración del motor de inferencia en el agente

Se adoptaron finalmente las *decisiones de diseño* siguiente para la implementación:

1. El *agente* usa a su *motor de inferencia* a través de una interface y por ese motivo, es independiente de la tecnología del *sistema experto* seleccionado.
2. El *agente* es responsable de la ejecución ordenada de los acontecimientos y de la interpretación de las respuestas emitidas por el *motor de inferencia*.
3. La coordinación entre *motores de inferencia* es gobernada sólo por los *agentes participantes* y es realizada sólo mediante paso de Mensajes ACL.

~

4. El *motor de inferencia* y el *agente* están interconectados mediante *comportamientos* JADE, que gestionan las *actuaciones* en el motor.
5. El agente restablece y persiste el estado de conocimientos de su *motor de inferencia* cuando se incorpora a, o es eliminado de, la plataforma.
6. El *comportamiento* que gestiona las *actuaciones* sobre el *motor de inferencia* es de tipo *Threaded Behaviour* de JADE.
7. La *ejecución* de ciclos de *motor de inferencia* tiene lugar en un *Thread* que es subordinado respecto al *Grupo de Threads* del *agente*.

Los tres primeras decisiones (1-2-3), se corresponden con *conclusiones de diseño* originadas a partir de la aplicación de buenas prácticas en desarrollo de aplicaciones en Lenguaje *Java* [Felleisen & Friedman, 2000] y de aplicaciones basadas en agentes [Shoham, 1993]. Las cuatro últimas (4-5-6-7) son requisitos *no-funcionales* obtenidos por las recomendaciones de desarrollo sobre plataforma JADE realizadas en [Bellifemine et al., 2010] y en los refinamientos sufridos durante el proceso de desarrollo de la integración *agente-motor*.

5.6.6 Técnica: Migración de '*agentes-nodo*'

La técnica aportada para llevar a cabo la migración del *agente* y su *nodo* de datos entre contenedores de la plataforma es la solución a uno de los requisitos de carácter crítico de la presente investigación. Los *agentes* con estas capacidades de inferencia no pueden ser serializados y la plataforma JADE, no puede realizar por si misma el proceso de migración entre contenedores con estos tipos de agente.

Con la técnica de migración se recupera la capacidad intrínseca de los *agentes* [Helin, 2003] y, gracias a esa recuperación, se restaura el mecanismo natural para obtener el *reparto de carga computacional* al ser viable desplazar los *agentes-nodo* hacia *contenedores* gestionados por la plataforma JADE donde existen recursos, de hardware y de memoria, disponibles.

Es importante destacar que sólo es posible migrar *agentes* hacia *contenedores* de la plataforma que hayan sido desplegados desde alguna de las *réplicas* del *Directorio de aplicación*. Esas *réplicas* permiten a los *agentes* encontrar el directorio `/nodes/` de manera relativa sobre el servidor de destino y por tanto, reubicar los archivos del fragmento del *sistema experto* que gestionan y que conectan a sus motores.

Queda entonces justificada finalmente, en este apartado, la importancia crítica que suponía la implementación de una *utilidad de creación de Directorios de aplicación* (Apdo. 5.4.1, pág. 70) para ser ejecutada en los distintos servidores[36] hacia los que poder migrar los agentes, la trascendencia de implementar las *utilidades auxiliares* (Apdo. 5.4.2, pág. 71), así como la necesidad de adoptar un conjunto de *convenios* (Apdo. 5.7.2, pág. 102) en la estructura del *Directorio de aplicación* para recuperar las capacidades de migración de estos *agentes*.

La técnica de *migración* implementada, que ha aportado unos resultados óptimos de rendimiento, emula el proceso seguido por un náufrago al lanzar un mensaje en una botella:

- El agente con intención de migrar hacia un contenedor remoto, prepara un mensaje dirigido a un agente de su misma *aplicación*, con la esperanza de que se encuentre ubicado en dicho contenedor remoto e indicándole su intención de migrar.
- En esos instantes, cuando el agente emite el mensaje (el náufrago), prepara un backup de su Memoria de trabajo y de su Base de conocimientos. Y empaqueta todos sus archivos del directorio /nodes/NombreDelAgente/ en un archivo .ZIP, a la espera de alguna respuesta positiva a su mensaje de socorro (el mensaje de la botella).
- Si el agente destinatario existe en ese momento en la plataforma multiagente, recibe el mensaje y comprueba que: (**1**) el agente solicitante pertenece a su *aplicación*, y (**2**) el agente solicitante de la migración no se encuentra en el mismo contenedor. Sólo entonces, inicia el diálogo de *rescate*.
- Al emitir el mensaje de confirmación de la viabilidad de migración, el agente (náufrago) responde de inmediato con un mensaje AGREE, al que incorpora en su contenido, el archivo .ZIP y en los parámetros adicionales del mensaje, información útil para que se reproduzca en contenedor destino la adecuada descompresión del mismo.

---

[36] ***Migración hacia servidores***: sería más correcto indicar migración hacia contenedores de la plataforma. Pero, la referencia a migración hacia contenedores, trata de poner en evidencia que la migración se producirá hacia contenedores creados en servidores remotos, lanzados contra la plataforma JADE desde *Directorios de aplicación* que poseen el mismo nombre.

- Concluido el proceso, el agente con intención de migrar procede a auto-eliminarse de la plataforma pero, emite antes de su eliminación un último mensaje de tipo INFORM, a su agente (rescatador), informando de esa actuación final.
- En el contenedor destino, el agente que ejecuta el proceso de migración procede con la descompresión de los datos recibidos en formato .ZIP y lanza una instancia del agente solicitante sobre su contenedor; dando por finalizado el proceso de migración.

5.7 Relación de *propuestas* y clases de agentes aportadas

La *Línea de trabajo* de propuestas aportó un espacio donde gestionar los esfuerzos dedicados a la construcción de agentes con motor de inferencia integrado, la redacción desarrollo y comprobación de los convenios adoptados para construir las réplicas físicas de los *Directorios de aplicación* (que pueden estar presentes en los servidores gestionados por la plataforma JADE), y la construcción de los agentes que ofrecen el *Servicio de Tickets*. Por lo que puede resumirse que la investigación aporta las propuestas siguientes:

(**1**) Convenios aplicables al *Directorio de aplicación.*
(**2**) *Agentes conectados al nodo de archivos del sistema de producción.*
(**3**) *Agentes que ofrecen el Servicio de Tickets.*

5.7.1 Propuesta: Directorio de aplicación

El *Directorio de aplicación* es generado por la *utilidad auxiliar* de instalación *framework*, invocada por defecto con [`java -jar dps-frameworkBuilder-0.9.1.jar`]. Utiliza métodos de la clase [`PsService`] para realizar la construcción del *Directorio de aplicación*, que es una estructura de subdirectorios y de convenios adoptados que permite, a los *agentes* desplegados, tener capacidades para localizar los archivos de código del *sistema de producción* que tienen que ejecutar y los archivos de texto en Lenguaje Java que contienen sus comportamientos y configuraciones específicas para modelar su conducta.

Una *Aplicación* puede tener tantas réplicas de *Directorios de aplicación* como *contenedores* gestione la *plataforma multiagente*. Los objetivos del *Directorio de aplicación* son múltiples; (**a**) permitir el despliegue de los *agentes*, (**b**) servir de punto de localización empleado para la ubicación de archivos del sistema de producción gestionado por cada agente, (**c**) elemento para facilitar la localización relativa en el sistema de archivos de los ficheros de comportamientos, etc.

La entidad *Directorio de aplicación* fue introducida en los apartados 5.4 y 5.4.1 (pág.70). Este apartado muestra los convenios propuestos para la nomenclatura de archivos y directorios, su estructura y la función asociada a cada uno de esos elementos:

1. El nombre del *directorio raíz* es empleado de manera relativa por los *agentes* para localizar los subdirectorios bajo su control. El nombre del *directorio raíz* coincide con el nombre de la *aplicación*.

2. El *Directorio de aplicación* ofrece el *Entorno de ejecución* desde donde desplegar los *agentes*, ejecutar *utilidades auxiliares* y desplegar el *Servicio de Tickets*.

3. El *Directorio de aplicación* facilita un *escenario para replicar ensayos* y permitir, al *equipo de investigación*, contrastar los resultados en cualquier arquitectura física y sistema operativo que cuente con una máquina virtual *Java* versión 7.0 ó superior.

4. El *Directorio de aplicación* aporta la estructura para permitir la *Migración de agentes* entre *contenedores* de la plataforma JADE, incluidos los conocimientos acumulados y su configuración íntegra antes de la migración.

Las propiedades de un *Directorio de aplicación* se encuentran en `/config/`, en el archivo [`framework.properties`]. Su contenido expresa los conceptos siguientes: *nombre* de la aplicación, *lista de confianza* o *lista de agentes-nodo* que componen la aplicación (tanto los locales como los remotos) y que pueden intercambiar mensajes dirigidos a sus motores de inferencia, *password*, *clave de cifrado* para envíos dirigidos a *Servicios de Tickets* securizados, etc. Le siguen, un bloque de propiedades con los *metadatos* y la ubicación del directorio de librerías nativas; necesaria para facilitar a los *administradores de sistemas* la configuración en *modo-infraestructura* para ejecutar múltiples *aplicaciones* centralizando las librerías, tanto de *Java* como las pasarelas JNI.

[Elaboración propia]

```
# ------------------------------------------------
# Problem-Solvers Framework Settings
# ------------------------------------------------
ps.framework.name          = app51
ps.framework.node.serie    = Pocket,Robot1,Robot2,Robot3,Robot3,nodoB,nodoC,nodoD
ps.framework.keypasswd     = theSecretPassword.
ps.framework.publickey     = asfjksd98843.9692adsf
# ------------------------------------------------
# Problem-Solvers Framework Application Meta-data
# ------------------------------------------------
ps.framework.app.title     = Titulo de la aplicacion original basasa en Sistemas expertos
ps.framework.app.created   = Fecha lanzamiento
ps.framework.app.author    = Su nombre/Empresa, aqui...
ps.framework.app.objective = Puede simular varias lineas con\\n antes de la nueva linea.

# ------------------------------------------------
# Problem-Solvers Framework Application native libraries location (relative o absolute)
# ------------------------------------------------
## By default, example for autonomous service:
ps.framework.app.nativelib =

## Example for MsWindows absolute path:
## ps.framework.app.nativelib = C:\\home\\unileon\\prototipos\\var\\lib
## Example for MsWindows absolute path:
## ps.framework.app.nativelib = /home/unileon/dps/bin/lib

## Example for Mac/Linux absolute path:
## ps.framework.app.nativelib = /opt/var/local/lib

## Example for all operating systems relative path:
## ps.framework.app.nativelib = ../var/lib
```

Código 5.17: Propiedades por defecto de un Directorio de aplicación (vista parcial)

5.7.2 Propuesta: Convenios en el Directorio de aplicación

Los convenios propuestos para los *Directorios de aplicación* son los siguientes:

[Elaboración propia]

Convenios y significados adoptados en el Directorio de la aplicación:

Directorio	*Descripción / función / contenidos:*

`Directorio raíz/`
 	
`Un nombre de ejemplo de Directorio de aplicación:`
 	
`app51/`
	Contendrá las réplicas distribuidas de la *aplicación* que se trata de transformar. Posee las propiedades siguientes:
	• Es el lugar empleado como punto de arranque de las *utilidades auxiliares*: `psnode`, `shell` y `launcher`, correspondientes al mecanismo de creación de *agentes*, la utilidad de acceso a los compiladores e intérpretes y la lanzadera de la plataforma y de los agentes.
	• Es el lugar empleado para crear nuevos *agentes-nodo*.
	• Contiene un archivo `app51.properties` para lanzar en bloque mediante: `java jade.Boot -conf app51.properties` una réplica de la aplicación en este servidor.
	• Contiene el archivo `sniffer.properties` para configurar las opciones del agente Sniffer de JADE.
	• Puedes ser trasladado a otro punto de montaje del sistema de ficheros y mantener funcional y operativa la *aplicación*.

`./config/`	• Es el subdirectorio donde se almacenan archivos de configuración. • Los archivos `framework.properties` y `framework.methods.bsh` exponen propiedades que afectan globalmente al funcionamiento de todos agentes de la *aplicación*. • Los archivos `help.xxxx.txt` contienen las ayudas *en-línea* cuando se ejecuta el agente en el Entorno de desarrollo.

`./nodes/`	Es el subdirectorio dedicado al almacenamiento de los datos de los agentes que componen la *aplicación*:
	• La estructura de `/nodes/` es similar, por ejemplo, a:
	`~/app51/`
	`    |`
	`    +---/nodes`
	`    |       +---/Robot_AA`
	`    |       |       +---/config`
	`    |       |       +---/engine`
	`    |       |       +---/var`
	`    |       +---/Robot_BB`
	`    |       |       +---/config`
	`    |       |       +---/engine`
	`    |       |       +---/var`
	• Cada subdirectorio de /nodes/ representa a un *agente*. El nombre de dicho subdirectorio queda vinculado en *tiempo de ejecución* al agente.
	• En el subdirectorio `/nodes/RobotAA/config/` se almacenan los comportamientos del agente.
	• En el subdirectorio `/nodes/RobotAA/engine/` se almacenan los archivos del *sistema experto* asignados al agente.
	• En directorio `/nodes/RobotAA/var/` se almacenan los volcados de la me-

Convenios y significados adoptados en el Directorio de la aplicación:	
Directorio	***Descripción / función / contenidos:***
	moria de trabajo y producción (WM+PM) sometidos a ciclos de rotación.
`./lib/`	• Es el subdirectorio de Librerías *Java*.
`./var/`	• Contiene el subdirectorio `/doc/` con la documentación.

Tabla 5.6: Nomeclatura y téminos propuestos en el Directorio de apliación

5.7.3 Propuesta: Agente-Nodo

La investigación concluye con la construcción de la clase de a*gente* [`PsNodeAgent.java`] que puede generar instancias de agentes desplegables sobre la plataforma JADE conectados a un motor de inferencia gracias a la estructura denominada *nodo*.

[Elaboración propia]

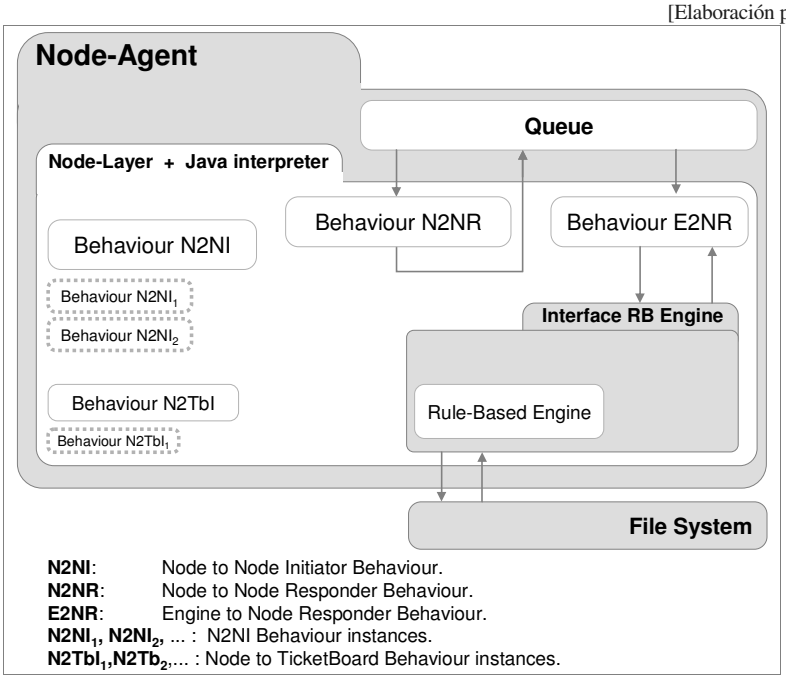


Figura 5.15: PsNodeAgent, esquema general de distribución de componentes

El *Directorio de aplicación* aporta información suficiente a estos agentes para que puedan localizar: (**1**) archivos de configuración, (**2**) archivos de comportamientos, (**3**) archivos del sistema experto y (**4**) archivos con el respaldo de las memorias de *reglas* y *hechos* para restaurar su estado de conocimientos tras los períodos sin existencia en la plataforma multiagente. Por tanto, un *agente PsNodeAgent*, está vinculado a las réplicas de los *Directorios* de la *aplicación* porque, son el lugar donde pueden existir como *nodo* y agente simultáneamente.

5.7.3.1 *Agente-Nodo: Distribución de componentes*

La distribución de componentes de un *agente-nodo* se representa en la Figura 5.15. El *agente* gobierna la interface [`RBEngine`] de su *motor de inferencia* mediante el comportamiento [EngineToNodeREResponder (`E2NR`)] y posee un *espacio de nombres* de objetos y de variables gracias a objeto `Node-Layer` que le permite contar con un amplio conjunto de propiedades

inicializadas. Los comportamientos y el motor de reglas conocen dicho *espacio de nombres* y obtienen la información necesaria para acceder al *sistema de ficheros*.

5.7.3.2 *Agente-Nodo: capa de propiedades 'Node-Layer'*

Estos agentes 'conocen' en todo momento la localización de los archivos del sistema de producción utilizados por el *motor de inferencia*. Otra misión del objeto *Node-Layer* es permitir la gestión de los archivos *comportamiento* del agente. El objeto *Node-Layer* es una instancia de la clase [`PsNodeSettings`] y puede ser analizado en detalle mediante en el *Entorno de desarrollo* con la llamada al método [`myNode.perspective();`].

5.7.3.3 *Agente-Nodo: Interfaz Gráfica de Usuario*

Estos *agentes* poseen una interfaz gráfica mínima (*véase,* Figura 5.16) donde reflejan el valor de sus variables principales al usuario. Cuando son arrancados con `debug` y `verbose` en su archivo de propiedades: `/nodes/Agente-nodo/config/node.properties` con valor 0, la interfaz gráfica de usuario se oculta a los 15 segundos. Esa interfaz gráfica sigue los criterios de diseño marcados para agentes destinados a operar en entornos reales, donde la gestión de los recursos debe ser óptima y la intervención del usuario nula [Bellifemine et al., 2010]. Los botones de `Exit`, `Select Board` y `Hide`, así como el [Área de redacción de texto] para componer una bitácora de los cambios adoptados, son incorporados para facilitar labores de documentación cuando el agente es desplegado dentro de *Entorno de desarrollo*.

[Elaboración propia]

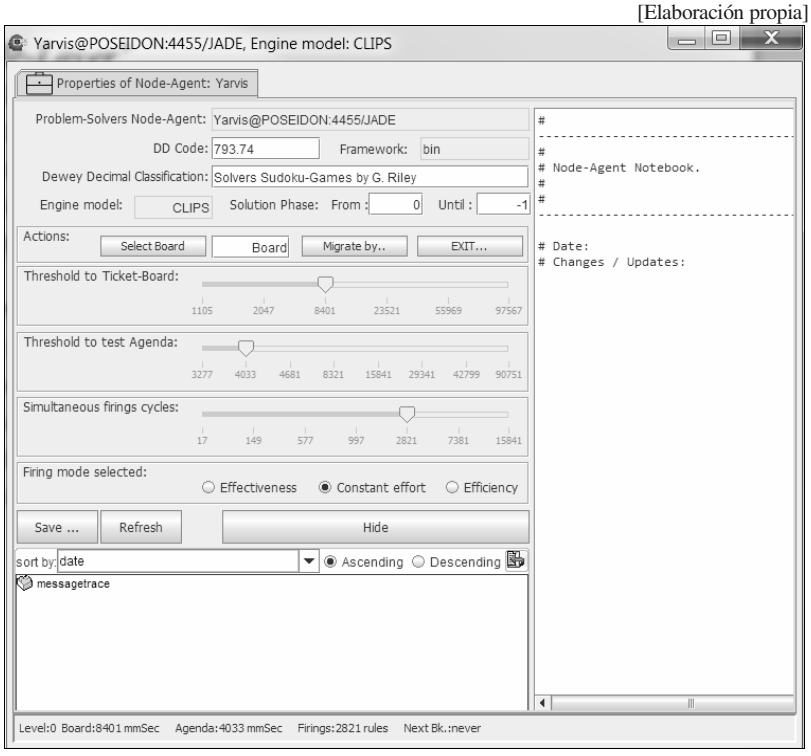


Figura 5.16: Interfaz gráfica de usuario de agentes [PsNodeAgent]

5.7.4 Propuesta: Agente-TicketBoard

Un a*gente TicketBoard* ofrece un espacio desatendido de intercambio de información, similar al que proporciona un *tablón público* donde es posible dejar o capturar anuncios breves (denominados *Tickets*). La clase [`PsBoardAgent`] hereda de *jade.core.Agent* y permite crear agentes que prestan el *Servicio de Tickets*. Emplean un nodo formado por directorios y en su archivo de propiedades se puede establecer el nombre público del servicio *'ticket-board'*:

[Elaboración propia]

```
# Default Ticket Board Agent Properties File
# ------------------------------------------
ps.board.agent.debugging.mode = 1
ps.node.agent.verbose.mode = 1
ps.node.agent.framework-board-service = ticket-board
```

Código 5.18: Archivo board.properties de un agente PsBoardAgent (vista parcial)

Los agentes generados a partir de [`PsBoardAgent`] se registran sobre el *directorio de páginas amarillas* de la plataforma en el [`Nivel-5`], utilizan el directorio `/config/` para ubicar y gestionar sus archivos de configuración y de comportamientos, y el directorio `/var/` para alojar la base de datos HSQLDB donde persisten los objetos *Tickets* recibidos. Gracias a ello, los agentes *TicketBoard* pueden reanudar su prestación del servicio tras períodos sin existencia sobre la plataforma multiagente, al reconectarse de nuevo a su base de datos local.

5.7.4.1 *Agente-TicketBoard: Distribución de componentes*

[Elaboración propia]

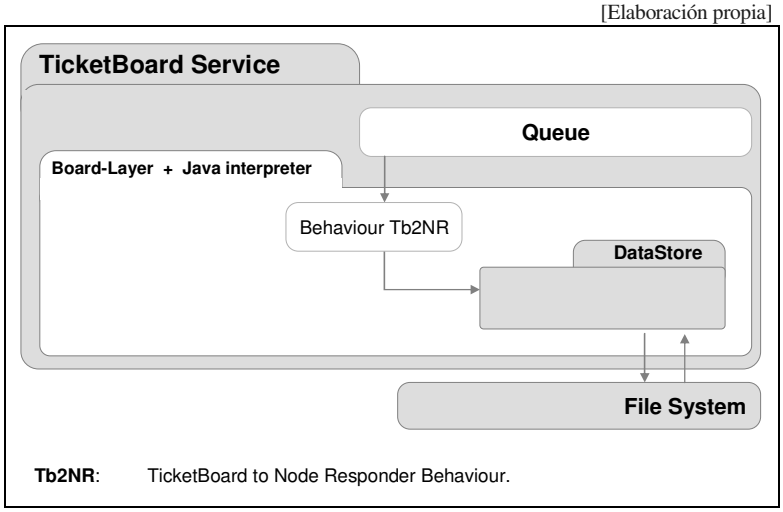


Figura 5.17: PsBoardAgent, esquema de distribución de componentes

Cuando se crea un *Directorio de aplicación* se genera por defecto un espacio de almacenamiento para un agente TicketBoard con el nombre: 'Board', concretamente en el directorio `/nodes/Board/`. La estructura de directorios y los archivos que posee no están orientados a integrar ninguna tecnología de sistemas expertos porque su misión, es almacenar objetos de la ontología *Problem-Solvers* denominados *Tickets*.

Los *Directorios de aplicación* incorporan este agente para facilitar las fases iniciales del período de desarrollo, sin embargo, no es necesario que existan en los despliegues sobre *entornos reales* donde, un único agente puede ofrecer el servicio de tickets a decenas de *agentes-*

nodo pertenecientes a distintas *aplicaciones*. Los componentes del esquema de la Figura 5.17, representan la combinación de objetos que permite externalizar los comportamientos, ofrecer respuestas a consultas realizadas por los *agentes-nodo* y ejecutar los métodos para almacenar y gestionar los objetos *Tickets.*

5.7.4.2 *Agente-TicketBoard: Interfaz Gráfica de Usuario*

[Elaboración propia]

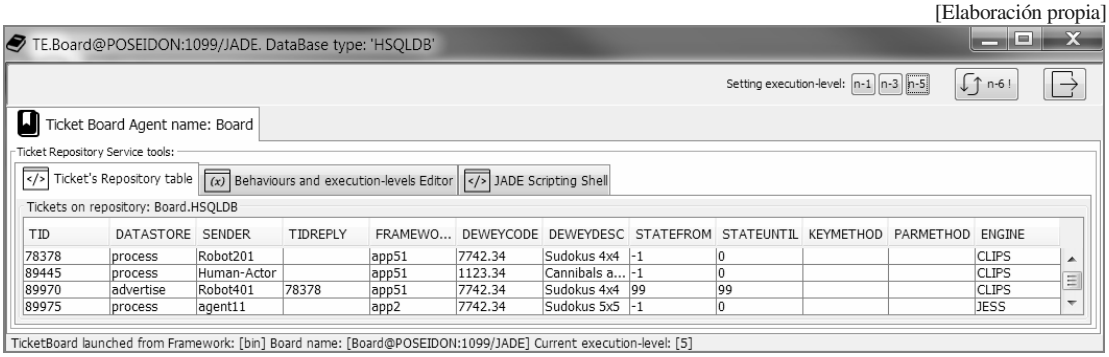


Figura 5.18: Interfaz gráfica de usuario de agentes [PsBoardAgent]

La *interfaz gráfica* de estos agentes se ha optimizado para facilitar al *desarrollador* la depuración y acumulación de los objetos *Tickets*, mostrados como registros en un componente tipo `JTable` de `javax.swing` como se muestra la Figura 5.18:

[Elaboración propia]

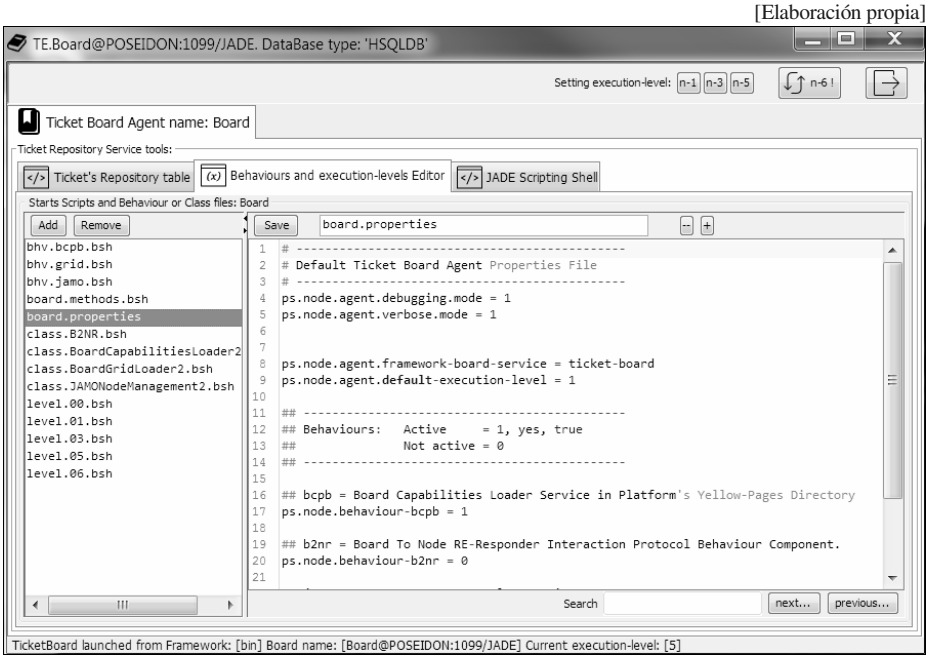


Figura 5.19: Pestaña de edición de archivos de agentes [PsBoardAgent]

Su diseño completo, consta de tres pantallas (Figura 5.18, Figura 5.19 y Figura 5.20) que permiten la intervención directa sobre la *base de datos* de *Tickets*, los comportamientos externalizados y las comprobaciones del propio servicio ejecutando *pruebas unitarias* sobre la consola de *Shell JADE*. El código de *interfaz* se encuentra en la clase `PsBoardAgentGui`. Una vez ajustados sus parámetros, estos agentes no requiere de interfaz gráfica alguna para operar

en entornos de producción por lo que, su interfaz gráfica queda oculta al establecer la variable `ps.board.agent.debugging.mode` igual a *false*.

[Elaboración propia]

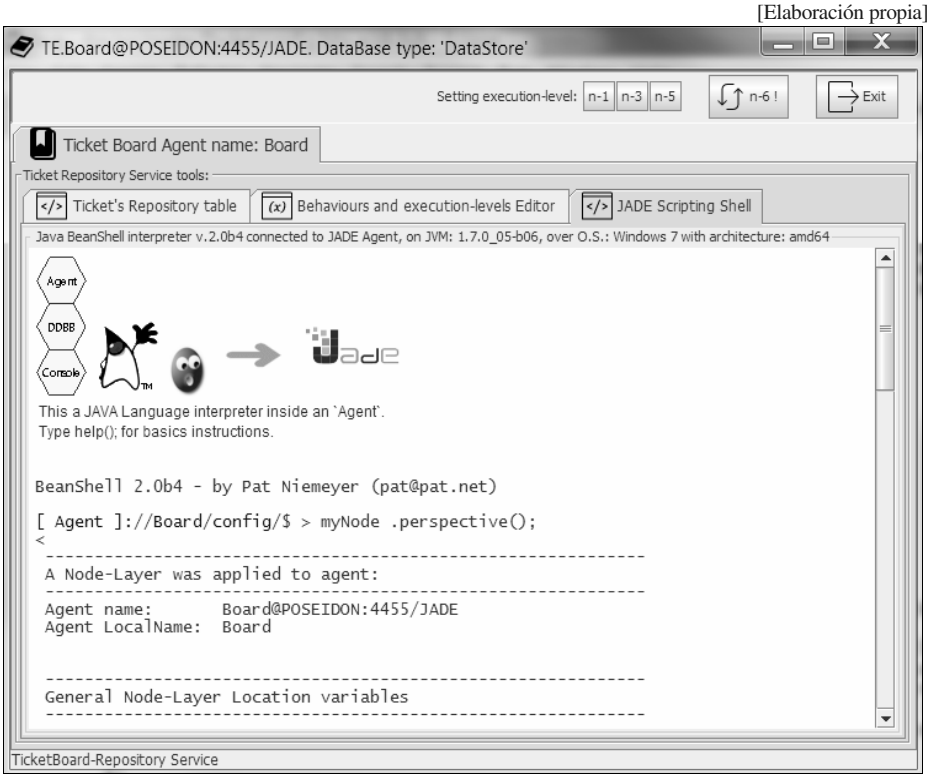


Figura 5.20: Shell-JADE en agentes [PsBoardAgent]

Cuando no es necesario encriptar los mensajes, o cuando el contenido de los *Tickets* no requiere mayor seguridad, un único agente puede prestar este servicio. En los casos más complejos, la pantalla de acceso a la consola *Shell JADE* (Figura 5.20) permite ensayar, programar, cargar nuevos comportamientos, modificar el proceso de arranque del agente y adaptar y comprobar los comportamientos para prestar servicios seguros a grupos de *agentes-nodos* pertenecientes a *aplicaciones* con exigencias de seguridad elevadas.

5.8 Síntesis de los resultados: factores críticos de éxito

A continuación se presentan los elementos clave que han determinado el éxito de las soluciones diseñadas en la presente investigación.

5.8.1 Comunicación agente-motor

El factor crítico de éxito determinante en la investigación fue la implementación del *protocolo de interacción* entre el agente y su motor de inferencia. La idea original que permitió avanzar en ese camino fue tratar, a ambas entidades de software, como dos agentes de la plataforma que conversan uno con el otro siguiendo un modelo similar al aconsejado por FIPA para llevar a cabo comunicaciones pautadas entre los agentes.

Cuando el agente [A2] de la Figura 5.21, recibe el mensaje de solicitud emitido por el agente [A1], el agente [A2] inicia una conversación interna con su *motor de inferencia* al que trata como un agente más de la plataforma y, con el que puede intercambiar mensajes. El valor [t_e] representa el *tiempo de interpretación* consumido por el agente [A2] para descifrar la petición del agente [A1] y comenzar una conversación con su motor de inferencia.

El valor [t_p] es el *tiempo de proceso* consumido en el interior del motor de inferencia para obtener una respuesta a la consulta. Concluidos ambos procesos, el agente [A2] aplica el formato a la información generada por el *motor de inferencia* y envía la respuesta al agente emisor [A1] considerando atendida la solicitud y finalizadas ambas conversaciones; la interna con su *motor de inferencia* y la externa con el agente [A1].

Para lograr una comunicación fluida entre ambas entidades software (agente y motor), el agente legítimo [A2] emite las solicitudes envueltas en Mensajes FIPA-ACL hacia un agente ficticio del cual, solamente conoce su *Identificador de Agente* (AID). Ese *agente falso* es el *motor de inferencia* integrado. Cuando esos mensajes FIPA-ACL aparecen en la bandeja de entrada, son capturados por el comportamiento de *respuestas del motor*, a continuación son interpretados por el motor, procesados y finalmente notificados mediante un nuevo Mensaje FIPA-ACL como respuesta dirigida al agente que inició la consulta.

[Elaboración propia]

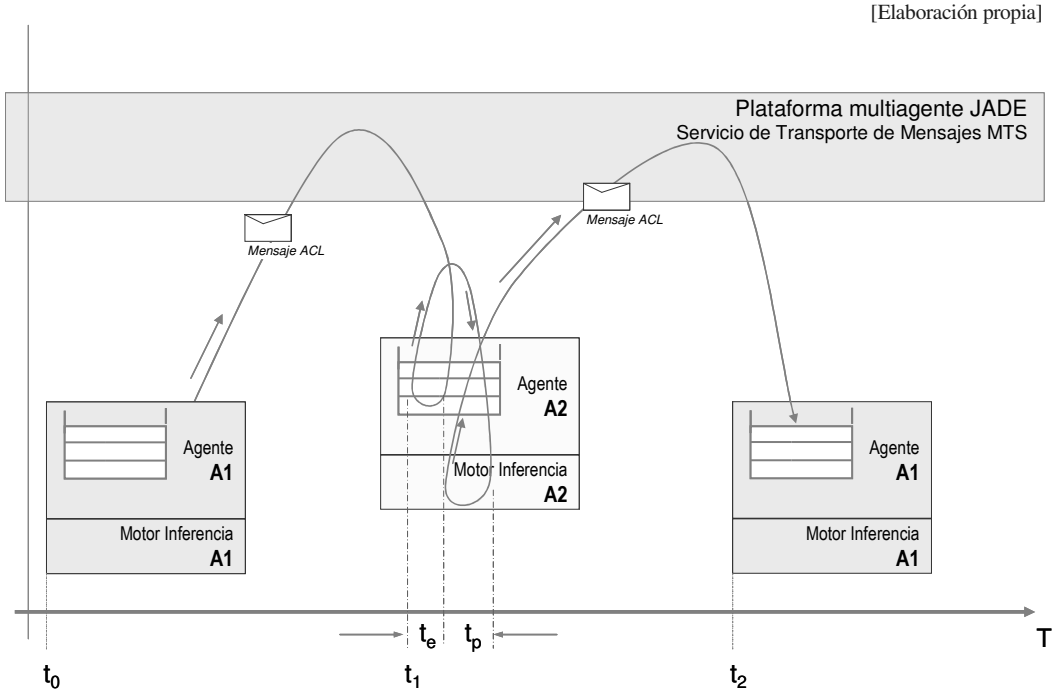


Figura 5.21: Comunicación agente-motor mediante Protocolo de Interacción

La implementación de ese mecanismo adoptó el formato de un comportamiento que se mantiene siempre alerta a las solicitudes: su nombre es `Engine To Node Responder` abreviado en el nombre de variable (e2nr) e implementado en la clase [E2NRFSMBehaviour]. Consta de tres estados; uno de captura, otro de procesamiento y un tercero de notificaciones, marcados en el diagrama como (A), (B) y (C), y está incorporado por defecto en los agentes [PsNodeAgent].

```
* ---------------------------------
* e2nr (Engine to Node Responder)
* ---------------------------------
*  +--------+         ThBhv      (D)
*  |        |   +--0--+/         /
*  |       0|   |     |        7/
*  +---(A)--+---+--(B)+---(C)--+
*       |    1         1   |
*       |                  |
*       +----by-default----+
*
* (A) Catcher Engine's Message
* (B) Processing Action Content
* (C) Send Results like Notification
```

Ese mecanismo permite, no sólo entablar diálogos de un agente con su *motor de inferencia* sino, entre los motores de inferencia de los agentes de la misma *aplicación*. Se obtuvieron entonces los primeros éxitos de estabilidad en los agentes con *sistemas de producción* integrados porque, quedaba liberado el *planificador de tareas* del agente y podían ser despachados los nuevos mensajes entrantes: convirtiendo en viables las futuras investigaciones relacionadas con razonamientos coordinados entre dos o más agentes.

Uno de los exponentes más versátiles que derivó de la implementación del *protocolo de interacción* entre agente y motor de inferencia, es la *Shell Asíncrona* (Figura 5.22) que realiza un uso intensivo de este mecanismo de comunicación.

[Elaboración propia]

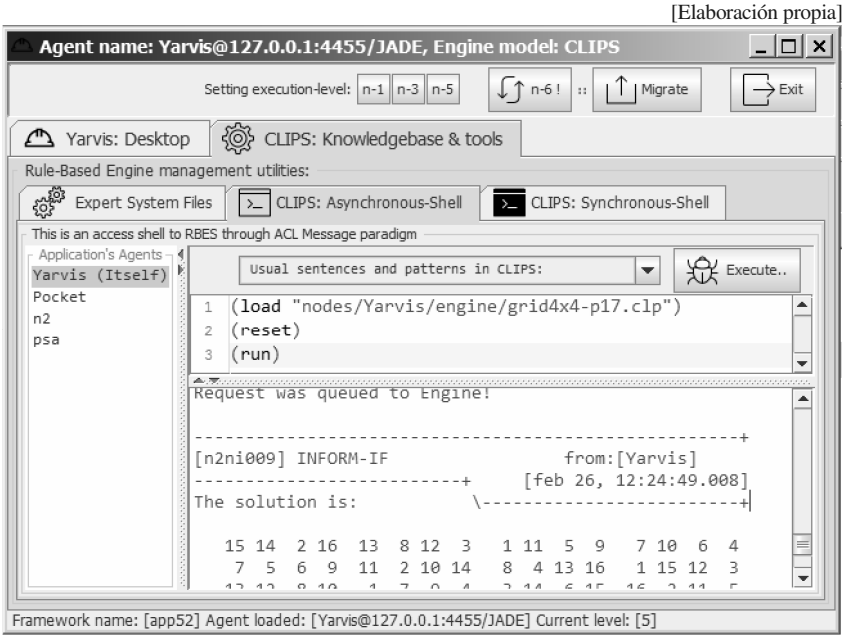


Figura 5.22: Entorno de desarrollo, Shell Asíncrona de un agente sobre sí mismo

5.8.2 La estructura Nodo

La segunda de las claves de éxito ha sido el *Nodo*, una estructura de directorios y archivos que el agente software localiza e incorpora a sus propiedades y que le permite gestionar a su motor de inferencia. El *Nodo* es el mecanismo principal que permite adaptar y repartir las competencias de cada agente y establecer el diseño modular del *Sistema de Producción* ori-

ginal sobre cada uno de los agentes que compondrán la versión distribuida de la aplicación desplegada en la plataforma multiagente.

```
/nodes/Agent.Name
          |
          +--config
          +--engine
          +--var
```

5.8.3 La estructura Framework

La tercera de las claves fue el *Framework*, que es el objeto que permite conectar las distintas réplicas que puede tener una *aplicación* desplegada en los distintos servidores de la plataforma JADE. Es además, para los agentes de una misma aplicación, un perímetro imaginario que informa a los agentes de cuáles son todas las entidades de software que conforman el mismo *sistema de producción* y con las que pueden compartir *hechos*, *reglas* y *sentencias* entre sus motores de inferencia. Ese perímetro establece una relación de confianza fiable mediante filtros de recepción de mensajes entre los agentes desplegados en la plataforma.

```
Framework.Name
          |
          +--config
          +--nodes
          +--logs
```

5.8.4 La externalización de comportamientos

El cuarto de los factores críticos es el mecanismo de externalización, creado como consecuencia de las exigencias de remodelar y adaptar al paradigma multiagente, el modo de ejecución natural de un *sistema de producción* pues, de lo contrario, se obligaría a los *ingenieros de software* a construir nuevos comportamientos específicos para cada instancia de cada agente en función de las necesidades particulares de cada aplicación.

```
* ------------------------------------------------------------ *
* myBOB (Bundle Of Node-Agent Behaviours proposal)             *
* ------------------------------------------------------------ *
* [0] = A label to ordering                                    *
* [1] = Level when Behaviour is initialized                    *
* [2] = Level when Behaviour is activated                      *
* [3] = Behaviour reference variable name                      *
* [4] = A label to behaviours' group                           *
* [5] = Externalized Behaviour definition file name            *
* === Level of  ====   =====   ============   =====  ======= *
* ID  Make Actv Bhv    Area    File name      Name   Actived *
* === ==== ==== ====== ======= ============   =====  ======= *
*  00 : 0 : 1 : jamo : basal : bhv.jamo.bsh : jamo : yes     JADE Mngmt. Ontology
*  0a : 0 : 0 : acpb : basal : bhv.acpb.bsh : acpb :  -      Agent Capabilities
*  2b : 1 : 3 : n2nr : RE_re : bhv.n2nr.bsh : n2nr : yes     Node To Node Responder
*  2c : 1 : 3 : e2nr : RE_re : bhv.e2nr.bsh : e2nr : yes     Engine To Node Responder
*  6a : 3 : 5 : e2as : Human : bhv.e2as.bsh : e2as : yes     Engine To Agent Shell
*  6b : 1 : 3 : e2aa : Robot : bhv.e2aa.bsh : e2aa :  -      Engine To Agent Analyzer
*  8a : 3 : 5 : v2ad : demon : bhv.v2ad.bsh : v2ad :  -      Visitor To Agenda Daemon
*  8b : 6 : 7 : v2bd : demon : bhv.v2bd.bsh : v2bd :  -      Visitor To Board Daemon
*  fa : 3 : 5 : bkwm : Mngmt : bhv.bkwm.bsh : bkwm :  -      Backup Engine WM + KB
*  fb : 3 : 5 : rswm : Mngmt : bhv.rswm.bsh : rswm :  -      Restore Engine WM + KB
*  fc : 0 : 1 : nmig : Mngmt : bhv.nmig.bsh : nmig : yes     Node Migration Mngmt.
```

La solución óptima encontrada para incorporar nuevos comportamientos y modelar la conducta de estos agentes, fue integrar un *Intérprete Java* en el agente, de forma que tuviese autonomía para instanciar nuevos objetos de comportamiento e incorporarlos a su *planificador de tareas* sin necesidad de alterar su diseño original de *clase*. Y, una evolución lógica de esas ideas y soluciones fue externalizar los comportamientos en archivos en Lenguaje Java sobre el directorio `/config/` del *Nodo* del agente.

Posteriormente, esta capacidad de Interpretación del código Java por parte del agente, concluyó en la implementación del modelo de arranque fragmentado en *niveles de ejecución* que facilitó sobremanera la depuración y el análisis de los comportamientos del agente. Y terminó por provocar la especialización de cada comportamiento y la elevación del número de comportamientos de carácter *Basal*, *Demonios*, *de Interacción* con el Actor-humano, de *Gestión*, etc., como muestra el arranque por defecto de los agentes [`PsNodeAgent`].

5.8.5 Un *Entorno de desarrollo* para los agentes

El quinto factor crítico de éxito ha sido la implementación de un *Entorno de desarrollo* para los agentes [`PsNodeAgent`], con capacidad de modificar los comportamientos externalizados, seleccionar el *nivel de ejecución* del agente, acceder a la Shell del *motor de inferencia* interno, contar con una Shell Asíncrona conectada a cualquiera de los demás motores de los agentes del mismo *Directorio de aplicación*, y la capacidad de modelar y editar los archivos del *sistema de producción* que se encuentran asignados a cada agente; todo ello en tiempo de ejecución. El *Entorno de desarrollo* se ha implementado como un agente que extiende la clase [`PsNodeAgent`] y que muestra entonces la configuración real del agente pero, con interfaces de usuario enriquecidas para facilitar las labores de adaptación, programación, comprobación y depuración sobre los agentes (*véase* Figura 5.22, pág. 109).

5.9 Comparativa con modelos de integración

5.9.1 Integración: JADE - Jess

El modelo de integración JADE-*Jess* propuesto por *Kamyab* y *Bellifemine* [Kamyab & Bellifemine, 2004], emplea un comportamiento de tipo *CyclicBehaviour* para alojar una instancia del motor de inferencia *Jess* desde el instante de la inicialización del comportamiento. El código se encuentra en el paquete [`examples.jess`] en los archivos de clases [`BasicJessBehaviour`] y [`JessAgent`], y en el archivo de Lenguaje *Jess* [`JadeAgent.clp`]:

```
public class BasicJessBehaviour extends CyclicBehaviour {
    Rete jess;                  // holds the pointer to jess
    Agent myAgent;              // holds the pointer to this agent
    int m_maxJessPasses = 0;    // holds the maximum number of Jess passes for each run
    int executedPasses = -1;    // to count the number of Jess passes in the previous run
    Hashtable AIDCache;         // holds a local cache to map agent names to AID

    public BasicJessBehaviour(Agent agent, String jessFile) {
        myAgent = agent;
        AIDCache = new Hashtable();
        jess = new Rete();
```

Código 5.19: Integración JADE-Jess de [Kamyab & Bellifemine, 2004] (parcial)

Para *Lopes Cardoso* [Cardoso, 2007], la integración de *Jess* en agentes JADE puede ser implementada mediante la técnica de un comportamiento [JessBehaviour] que extiende la clase *CyclicBehaviour* con gestión del número de ciclos de ejecución del propio comportamiento:

```
class JessBehaviour extends CyclicBehaviour {
 private jess.Rete jess;
  // maximum number of passes that a run of Jess can execute before giving c
 private static final int MAX_JESS_PASSES = 1;

 JessBehaviour(Agent agent, String jessFile) {
   super(agent);
   // create a Jess engine
   jess = new jess.Rete();
   // load the Jess file
   try {
```

Código 5.20: Integración JADE-Jess de [Cardoso, 2007] (parcial)

En la presente investigación, los agentes [PsNodeAgent] logran la integración entre agente y motor de inferencia mediante la interface *RBEngine* declarada como:

```
package examples.psnode.core.engines;

public interface RBEngine {
...
public int examples.psnode.core.engines.RBEngine.getState()
public java.lang.StringBuffer examples.psnode.core.engines.RBEngine.getBuffer()
public java.lang.String examples.psnode.core.engines.RBEngine.getVersion()
public java.lang.Object examples.psnode.core.engines.RBEngine.getRBEngine()
public boolean examples.psnode.core.engines.RBEngine.setRouter(java.lang.Object)
public void examples.psnode.core.engines.RBEngine.setThGroup(java.lang.ThreadGroup)
public void examples.psnode.core.engines.RBEngine.setThName(java.lang.String)
...
```

Y se incorpora, en el método [setup], la referencia al objeto *myEngine*, mediante objetos de tipo RBEngineCLIPS o RBEngineJESS que extienden la interface *RBEngine* con:

```
case "JESS":
// ////////////////////////////////
// Engine JESS 8.01a
        myEngine = new RBEngineJESS(thFWGroup, getLocalName() + myNode.engineSelected);
        if ( args != null ){
          if ( args.length > 1 ){
           if ( args[1] instanceof jess.Rete ){
                 myEngine = new RBEngineJESS( args[1], thFWGroup, getLocalName() +
                         myNode.engineSelected );
                 outLogger(new Object[] { "setup", myNode.nodeName,
                         "JESS Engine reference from outside this node-agent:" +
                         args[1].toString()});
        }}}
        break;
case "CLIPS":
...
```

La Tabla 5.7 (pág.113) resume, de manera gráfica, las diferencias entre los modelos de integración de agentes JADE con motores tipo *Jess*, publicados por *Kamyab* y *Bellifemine* en [Kamyab & Bellifemine, 2004] y *Cardoso* en [Cardoso, 2007], respecto al modelo de integración propuesto en la presente investigación. Los comportamientos han sido representados por *Triángulos*. Y el *motor de inferencia* y el *agente* se representan como *Hexágonos* con una etiqueta para diferenciar ambas entidades.

Los modelos de integración JADE-Jess citados, consideran al *motor de inferencia* (Engine) como un objeto que pertenece al comportamiento (figura a la izquierda). La investigación concluye con una propuesta que trata de retrasar la incorporación del tipo de *motor de inferencia* mediante un objeto que cumple la interface *RBEngine*. Y considera que los comportamientos son gestionados por un Intérprete de *Java* y almacenados en archivos externalizados, que no requieren cargarse en el agente para gestionar la instancia de *motor de inferencia*.

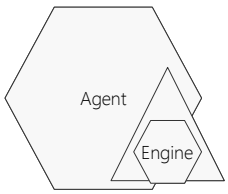
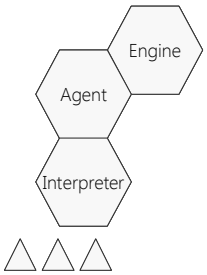
1º Integración: **JADE** - **Jess**		

Tabla 5.7: Comparación gráfica de *modelos de integración* JADE-Jess

Por ese motivo, en la imagen a la derecha de la Tabla 5.7, aparecen los comportamientos en una posición externa al *agente*. Y tanto el *agente*, como el *motor de inferencia* y el *intérprete de Java* son entidades que presentan una integración controlada por objetos interface y son representadas como entidades interconectadas pero, independientes.

5.9.2 Integración: JADE - CLIPS

No se han encontrado investigaciones destinadas a desplegar aplicaciones basadas en sistemas de producción CLIPS sobre agentes JADE. Los agentes [PsNodeAgent] emplean el mismo mecanismo para integrar motores tipo *Jess* y tipo CLIPS:

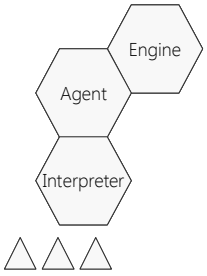
2º Integración: **JADE** - **CLIPS**	No se encuentran investigaciones sobre el despliegue en JADE, de aplicaciones completas basadas en sistemas de producción CLIPS.	

Tabla 5.8: Comparación gráfica de *modelos de integración* JADE-CLIPS

La implementación ha buscado la simetría entre cualquier tipo de tecnología de motor de reglas conectada a agentes [`PsNodeAgent`]. Por ello, aunque CLIPS quede integrado mediante la pasarela CLIPSJNI, el objeto agente [`PsNodeAgent`] no es sensible a ello. Las librerías nativas de CLIPS en C++, se encuentran ordenadas por arquitectura en subdirectorios de `/lib/`.

5.9.3 Integración: JADE - Jess - Protégé

Oliver Hoffmann, en colaboración con *Fabio Bellifemine* y *Friedman-Hill* [Hoffman et al., 2001] establecieron el *modelo de integración* para agentes JADE, con el motor *Jess* conectado a la pestaña del plugin *JessTab* [Eriksson, 2003] en Protégé [Protégé Community & Stanford University, 2014], teniendo en cuenta para la implementación que:

- La instancia del motor *Jess*, empleada por el plugin `JessTab`, se conectaría al motor del agente mediante una *clase estática* común de ambas aplicaciones (JADE y Protégé).
- Con un comportamiento (`JadeJessBehaviour`) se alcanzaría el acceso a la instancia del motor *Jess* gestionado por el plugin `JessTab` en Protégé.
- La implementación se encuentra disponible en el paquete: [`examples.JadeJessProtege`] en los archivos Java y Jess siguientes: [`JadeJessAgent.java`], [`JadeJessBehaviour.java`], [`JessHashtable.java`], [`Jess.clp`], [`JessProtege.clp`].

La Tabla 5.9 (pág.114) resume, de manera gráfica, las diferencias entre el modelo de integración que emplea una *clase estática* (imagen izquierda) para intercambiar datos entre el agente con motor *Jess* y el plugin de *JessTab* de Protégé; respecto al nuevo modelo de integración basado en la API de programación de JADE denominada *inProcess* (imagen a la derecha).

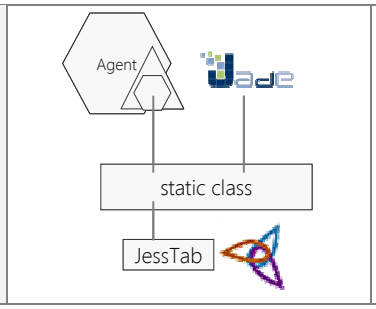
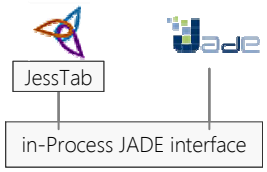
3º Integración: **JADE - Jess - Protégé**		

Tabla 5.9: Comparación gráfica de *modelos de integración* JADE-Jess-Protégé

Con el modelo propuesto, la instancia del plugin *JessTab* de Protégé puede construir una nueva instancia de agente [`PsNodeAgent`] conectada al mismo motor de inferencia que *JessTab*, mediante el Código 5.21 mostrado a continuación: (*véase resultado final* en Figura 5.23)

```
(bind ?rt2     (call jade.core.Runtime instance) )   ;;; Runtime of JADE
(bind ?prof2   (new jade.core.ProfileImpl) )         ;;; By default localhost and port = 1099
(bind ?cc      (call ?rt2 createAgentContainer ?prof2 ))
(bind ?args    (list "jess" (engine) ) )       ;;; Parameters List with reference to [Rete]
;; (bind ?eva2    (call ?cc createNewAgent "Eva" "examples.psnode.PsNodeAgent" ?args))
(bind ?eva2    (call ?cc createNewAgent "Eva" "examples.psnode.util.PsPocket" ?args))
(bind ?proxy2  (call ?eva2 start ) )
;; (bind ?killer2  (call ?eva2  kill))
```

Código 5.21: Nuevo modelo para la integración JADE-Jess-Protégé

[Elaboración propia]

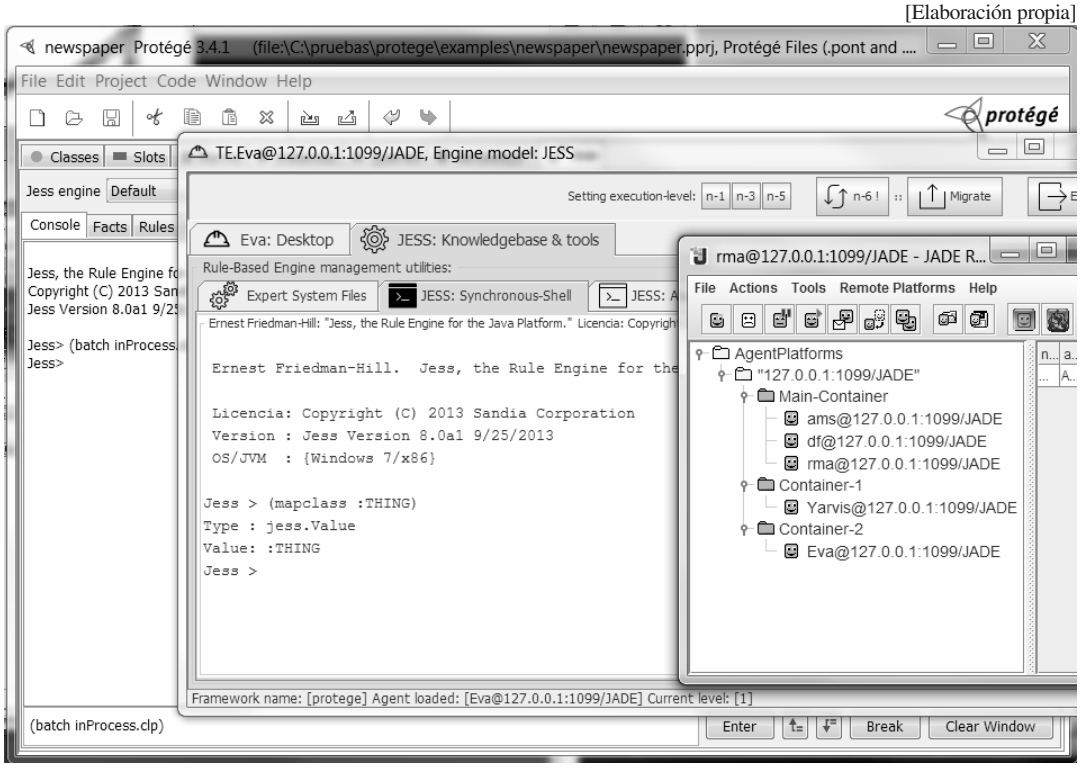


Figura 5.23: Evolución propuesta para integración JADE-Jess-Protégé

En la Figura 5.23, el agente `Eva`, se ha creado y lanzado en la plataforma JADE (*véase* agente RMA, *container-2* en la Figura 5.23) desde la línea de comandos del plugin *JessTab* con las sentencias indicadas en el Código 5.21. Como el agente `Eva`, está utilizando su *Entorno de desarrollo*, resulta posible acceder a la pestaña *Shell-Jess* y comprobar que, el motor de inferencias empleado por el *agente* es la misma instancia de objeto que el motor de inferencia que posee la estructura de clases generada por *JessTab* en su memoria de trabajo. Para ello, se revisan las clases de la *ontología de periódicos* mediante la llamada a `(mapclass: THING)`.

5.10 Resumen

- Los resultados aportados por la investigación se han clasificado en función de la *Línea de trabajo* en la que fueron construidos. Esas Líneas de organización fueron: *utilidades*, *instrumental*, *técnicas* y *propuestas* (*véase* Tabla 5.1, pág. 65):

  - Las *utilidades auxiliares* forman el grupo de herramientas estáticas, destinadas a preparar el *Directorio de aplicación* sobre el *sistema de archivos*, la construcción de agentes nodo, o facilitar el despliegue de nodos sobre el *Entorno de desarrollo*.

  - Los *instrumentos creados* aseguran una utilización: correcta, escalable y ajustada a la normalización vigente de las tecnologías aportadas por el *paradigma multiagente* para la construcción de aplicaciones basadas en agentes.

- Las *técnicas* aportadas agrupan las soluciones y los mecanismos ideados para satisfacer los requisitos originales de la presente investigación.

- Las *propuestas* aportan las piezas de software que formarán la estructura modular empleada para el despliegue de las *aplicaciones* sobre la plataforma.

6 Pruebas de concepto

Todo debe simplificarse lo máximo posible, pero no más.

Albert Einstein

Objetivos del capítulo:

- *Describir las pruebas de concepto.*
- *Mostrar cómo configurar las aplicaciones con librerías centralizadas.*

6.1 Introducción

Se han preparado dos pruebas de concepto para detallar paso-a-paso cómo iniciar el proceso de transformación de sendas aplicaciones hasta alcanzar sus despliegues en la plataforma multiagente JADE; y se incluyen aclaraciones adicionales a lo largo del proceso para destacar capacidades intrínsecas de las herramientas aportadas por la investigación.

La primera de las pruebas de concepto muestra cómo obtener la versión distribuida de la aplicación de resolución de puzzles matemáticos de lógica tipo *Sudoku*, aplicación publicada en la distribución oficial de CLIPS por Gary Riley [Riley, 2016a]. Es un *Sistema de Producción* desarrollado en CLIPS, que resuelve ese tipo de crucigramas de lógica y que aporta una colección amplia de ejemplos de distintos niveles de complejidad. Con dicha colección pueden obtenerse medidas comparables de tiempo consumido en su resolución mediante *agentes*. Además, esa aplicación permite ser empleada tanto con tecnología de motores de inferencia CLIPS como con motores *Jess* sin alterar el código, propiedad que resultaba muy interesante para ser utilizada como mecanismo de demostración de la capacidad de los *agentes* para permutar el tipo de motor de inferencia sin modificar sus propiedades.

La segunda de las pruebas presentadas en el capítulo está dedicada al *Sistema de Producción* denominado O-DEVICE [Meditskos & Bassiliades, 2008], [Meditskos & Bassiliades, 2011] disponible también en SourceForge para su descarga[37]. Se trata de un sistema experto que

[37] **odevice_v2.rar**: https://ayera.dl.sourceforge.net/project/o-device/odevice_v2.rar

contiene la base de conocimientos (*Knowledge Base*) en formato orientado a objetos. Es un sistema desarrollado en Lenguaje CLIPS que transforma los archivos de ontologías expresadas en formato OWL o RDF, en esquemas orientados a objetos en Lenguaje COOL de CLIPS. Durante el proceso de transformación convierte los conceptos de la ontología OWL, en las clases correspondientes sobre COOL, las propiedades de OWL en *slots* de las citadas clases y las instancias de OWL, en objetos COOL. En palabras de *Georgios Meditskos* [Meditskos & Bassiliades, 2006] «El propósito de esa transformación es múltiple: (**a**) explotar las ventajas de la *representación del conocimiento* en formato orientado-a-objetos y acceder a todas las propiedades de las instancias en un sólo paso, (**b**) contar con la posibilidad de emplear un lenguaje de razonamiento basado en reglas y orientado a objetos para poder interrogar y generar vistas de las instancias de OWL, las cuales descansan sobre esquemas orientados a objetos de CLIPS y, (**c**) responder con una rapidez mucho mayor a las consultas pues, la riqueza de los significados de OWL ha sido previamente analizado y transformado».

Para la transformación, los agentes requieren acceso a librerías del proyecto Apache-JENA y librerías de analizadores sintácticos para interpretación previa del archivo OWL/RDF de la ontologías (*véase*, Tabla 6.2, pág. 121 para mayor detalle). Esas exigencias particulares que plantea O-DEVICE fueron consideradas de gran interés para demostrar cómo incorporar dichas librerías al núcleo de un *Directorio de aplicación* porque, la transformación y adaptación de aplicaciones basadas en *Sistemas de Producción* al paradigma multiagente, presentarán similares circunstancias de forma habitual con librerías específicas que permitan su funcionamiento. La dificultad de transformación de archivos de ontologías y el consumo de CPU que supone las transformaciones en *reglas*; convirtió entonces al Sistema O-DEVICE en una aplicación candidata y doblemente atractiva como *prueba de concepto*.

Este sexto capítulo arranca con una exposición de la configuración en *modo-infraestructura*: una característica que fue introducida en la etapa final de esta investigación, en la *Línea de trabajo* de *utilidades software* y en la *Línea de propuestas de agentes*. Originalmente, el *Directorio de aplicación* generado por el instalador de Frameworks [`dps-frameworkBuilder-0.8.1.jar`], era 'un bloque compacto y autónomo' que servía exclusivamente como *Plataforma de Desarrollo* destinada a expertos. Sin embargo, a la vista de los resultados preliminares se incluyó un importante *Requisito Funcional* para el *Directorio de aplicación*, de manera que fuese capaz de convertirse en el pieza clave para el despliegue de aplicaciones en instalaciones a gran escala, incorporando mecanismos para tener un control completo de las librerías compartidas por el servidor gestionado por la plataforma multiagente.

6.2 Configuración en *Modo-infraestructura*

En la versión final del instalador [`dps-frameworkBuilder-0.9.1.jar`], el *Directorio de aplicación* generado por defecto, se muestra en Tabla 6.1 de la página siguiente. A ese despliegue *por-defecto* del instalador se le denomina *Modo-entorno-de-desarrollo* y es la pieza perfecta para investigadores en AOP, desarrolladores *scouts* o para equipos de desarrollo profesionales, al tratarse de una solución preparada para comenzar a analizar y desarrollar nuevos agen-

tes, con el único requisito de incorporar a la variable CLASSPATH el valor [`lib/*:`] y desplazarse al *Directorio de aplicación* recién creado.

El contenido y las funciones de las librerías del directorio [`lib/`] se detalla en la (Tabla 6.2, pág.121). Dicho directorio incluye la plataforma JADE, las librerías de motores de inferencia CLIPS y *Jess*, el Intérprete de Java BeanShell, la librería de edición RSyntaxTextArea y un cliente para bases de datos HSQLdb. Posee además, los subdirectorios [`jena`] y [`w32, w64, ux32, ux64, mac`], para localizar las pasarelas CLIPSJNI y las librerías Apache-JENA.

El *Modo-entorno-de-desarrollo* es el que se ha venido empleando a lo largo del capítulo 5º para exponer las funcionalidades básicas de utilidades, cómo crear nodos de *agentes*, ajustar archivos de configuración del *Directorio de aplicación*, configurar los agentes, etc.

[Elaboración propia]

Estructura generada por-defecto para un Directorio de aplicación

Subdirectorios de un Directorio de aplicación	***Contenidos***	***Requerido***	***Infraestructura***
├──config	Archivos de configuración del *Directorio de aplicación*.	**Sí**	**No**
├──lib │ ├──jena │ ├──mac │ ├──ux32 │ ├──ux64 │ ├──w32 │ └──w64	Librerías Java, pasarelas JNI de CLIPS, Librerías JENA	**No**	**Sí**
├──logs	Archivos temporales y secuencias de comandos.	**Sí**	**No**
├──nodes │ ├──Agent │ ├──Board │ ├──Dummy │ └──Pocket	Nodos por defecto para la creación y depuración de *agentes*. **Agent**: *agente-nodo* sin conexión a motor de reglas. **Board**: *agente-TicketBoard*. **Dummy**: *Entorno de desarrollo* de agentes *DummyAgent*. **Pocket**: *Entorno de desarrollo* de *agentes-nodo*.	**Si**	**No**
└──var └──doc ├──bsh ├──dps ├──engines └──hsql	Documentación: JavaDoc, Manuales y Componentes.	**No**	**No**

Tabla 6.1: Estructura por defecto de un Directorio de aplicación

Los subdirectorios que definen un *Directorio de aplicación* se han marcado en la columna del campo [*Requerido*] de la Tabla 6.1. En [`/config/`] los archivos [`framework.properties` y `framework.methods.bsh`] son imprescindibles pues definen las propiedades y las funciones necesarias en los agentes para compilar los comportamientos. Los demás, son archivos de

apoyo al desarrollador: la serie de archivos de ayuda [help_xxx.txt], los archivos de autocompletado de sintaxis de los Lenguajes de programación *Jess*, *Java* o CLIPS, etc. El subdirectorio [/lib/] no forma parte del *Directorio de aplicación* y podrá ser desplazado a una ubicación centralizada, como se verá a continuación. El subdirectorio [/nodes/] debe existir, pero no requiere contener ningún ejemplo ni plantilla de agentes para que el *Directorio de aplicación* sea operativo en *modo-infraestructura*. Finalmente, el directorio [/var/] tampoco es necesario y puede ser eliminado tanto, de las versiones desplegadas en servidores de producción con las librerías centralizadas, como de los despliegues en local.

Por ello se muestra, al directorio [/lib/] en la Tabla 6.1, como el elemento a extraer de un *Directorio de aplicación* que puede ser trasladado a una ubicación alternativa para lograr la centralización de las librerías de todas las aplicaciones presentes en el servidor. Destacar que las pasarelas de CLIPSJNI son operativas y localizables de manera centralizada, al establecer el valor de la propiedad [ps.framework.app.nativelib] para las distintas aplicaciones que empleen agentes con el motor CLIPS y que deban operar de manera centralizada.

Esa propiedad se encuentra en el archivo: [/config/framework.properties] y los valores que pueden asignarse a dicha propiedad se resumen a continuación:

***Técncias para centralización de Librerías nativas de CLIPS*:**	
ps.framework.app.nativelib =	(Descripción de funcionamiento)
=	***Valor por defecto***, construye la **Ruta relativa** a partir del *Directorio de aplicación* e incorpora el subdirectorio ./lib Es válida para todos los sistemas opertivos.
= ../var/lib	**Ruta relativa** a partir del *Directorio de aplicación* más la cadena indicada. Válida para todos los sistemas opertivos.
= C:\\var\\dps\\lib	**Ruta absoluta**. Sólo válida en servidores con sistema opertivo *Microsoft Windows*.
= /var/local/lib	**Ruta absoluta**. Válidad para todos los sistemas operativos y para servidores con *Microsoft Windows* con la instalación del *Directorio de aplicación* en la unidad C:\

Valores que deben ser incorporados de manera coordinada entre la variable CLASSPATH y la variable de localización de librerías nativas CLIPS de cada aplicación:

CLASSPATH + ps.framework.app.native.lib por tipo de despliegue:		
ps.framework.app.nativelib	**CLASSPATH =**	**Modo**
=	= lib/*;	Entorno-Desarrollo-local
= ../var/lib	= ../var/lib/*;	Infraestructura (relativa)
= C:\\var\\dps\\lib	= C:/var/dps/lib/*;	Infraestructura (Windows Server)
= /var/local/lib	= /var/local/lib/*:	Infraestructura (Linux, OS X)

Se listan a continuación, en la Tabla 6.2, los nombres de las librerías que genera el instalador de *Directorios de aplicación* y las funciones que desempeñan en las *aplicaciones*.

Directorio	*Librerías*	*Tamaño*	*Escenario donde son requeridas:*
`lib/`	commons-codec-1.3.jar	46 K	Cifrar *Byte Stream* en [`content`] de mensajes.
`lib/`	jade.jar	2712 K	Crear la instancia de plataforma JADE.
`lib/`	bsh-2.0b4.jar	276 K	Intérprete *BeanShell* adosado a *agentes-nodo*.
`lib/`	psnode-agents-0.9.1.jar	859 K	Utilidades, herramientas y *agentes-nodo.*
`lib/`	CLIPSJNI.jar	13 K	Emplear motor CLIPS en *agentes-nodo*
`lib/ mac/`	libCLIPSJNI.jnilib	2633 K	
`lib/ ux32/`	libCLIPSJNI.so	1082 K	
`lib/ ux64/`	libCLIPSJNI.so	1153 K	
`lib/ w32/`	CLIPSJNI.dll	1048 K	
`lib/ w64/`	CLIPSJNI.dll	1448 K	
`lib/`	jess.jar	630 K	Emplear motor *Jess* en *agentes-nodo*
`lib/`	JessML1_0.xsd	12 K	
`lib/`	jsr94.jar	15 K	
`lib/`	rsyntaxtexarea-2.5.7.jar	1437 K	Abrir *agentes-nodo* en *Entorno de desarrollo*
`lib/ jena/`	commons-logging-1.1.1.jar	60 K	Desplegar O-DEVICE en *agentes-nodo*.
`lib/ jena/`	icu4j_3_4.jar	3074 K	
`lib/ jena/`	iri.jar	148 K	
`lib/ jena/`	j2cf.jar	27 K	
`lib/ jena/`	jena.jar	2165 K	
`lib/ jena/`	stax-api-1.0.jar	26 K	
`lib/ jena/`	wstx-asl-3.0.0.jar	463 K	
`lib/ jena/`	xercesImpl.jar	1176 K	
`lib/`	sqltool.jar	145 K	Cliente SQL de consola de comandos.
`lib/`	hsqldb.jar	1459 K	*Tickets* en base de datos HSQLdb.

Tabla 6.2: Función de las librerías incluidas en el directorio ./lib/

6.3 Prueba de concepto: Puzzles de tipo Sudoku

Sobre un ordenador personal se enumeran a continuación los pasos mínimos a realizar para la transformación al escenario multiagente del *Sistema de Producción* de Gary Riley, disponible en *sourceForge* para el cálculo de puzzles de lógica de tipo *Sudoku* [Riley, 2016a].

Se han descargado los archivos [`dps-frameworkBuilder-0.9.1.jar`][38] para la creación de réplicas del *Directorio de aplicación* y, el archivo de ejemplos [`examples_630.zip`][39] de CLIPS sobre un directorio denominado [`pruebas`]. Se procede entonces con la construcción del *Directorio de aplicación* para obtener acceso a las *utilidades auxiliares*.

[38] **dps-frameworkBuilder-0.9.1.jar**: https://cekb.unileon.es /MAS/dps
[39] **examples_630.zip**: https:// downloads .sourceforge .net /projec t/clipsrules /CLIPS /6.30/

6.3.1 Introducción: una primera aproximación desde la *Shell*

Una vez creado el directorio de aplicación y antes de desplegar la plataforma JADE, debe ser evaluado inicialmente el correcto funcionamiento de la aplicación CLIPS 6.31 para comprobar que los ejemplos operan correctamente sobre CLIPS en funcionamiento desde una máquina virtual de *Java* en cualquier sistema operativo. Para ello, desde el terminal, se procede con la creación de un nuevo *Directorio de aplicación*:

```
1.   c:\pruebas> java -jar  dps-frameworkBuilder-0.9.1.jar --new=sudokus --no-doc

2.1. c:\pruebas> cd sudokus
2.2. c:\pruebas\sudokus> set CLASSPATH=lib/*;
                                        (NOTA: con estos dos pasos, quedan operativas las
                                        utilidades, como por ejemplo, la Shell en consola).
3. c:\pruebas\sudokus> unzip examples_630.zip         (Extraer Ejemplos de CLIPS 6.30)

4. c:\pruebas\sudokus> java shell clips

CLIPS> (batch* "Examples/sudoku/runbench.bat" )   (NOTA: puede variar en función de la CPU)
       %%%%%%%************
       %%******************************************************
       **********
       *
       7515 rules fired        Run time is 0.545031070709229 seconds.
       13788.2047535767 rules per second.
       656 mean number of facts (1493 maximum).
       1 mean number of instances (1 maximum).
       1226 mean number of activations (5503 maximum).
       8746 rules fired        Run time is 1.99811506271362 seconds.
       4377.12530334571 rules per second.
       685 mean number of facts (1502 maximum).
       1 mean number of instances (1 maximum).
       1684 mean number of activations (5503 maximum).
       TRUE

5. CLIPS> (exit)
```

6.3.2 Creación de nuevos *nodos* para los agentes

Una vez verificado que la aplicación basada en Sistemas de Producción CLIPS o Jess opera sobre entorno *Java*, desde el *Directorio de aplicación* se procede con la creación de nuevos *nodos* con la *utilidad auxiliar psnode* que, se desplegarán en la plataforma JADE como agentes con dicho nombre para que puesan localizar las variables y los recursos creados en los *nodos*. Indicar que, la plataforma JADE, puede quedarse activada durante todo el transcurso de las pruebas.

```
1.1. c:\pruebas\sudokus> java psnode new Robot01
1.2. c:\pruebas\sudokus> java psnode new Robot04
1.3. c:\pruebas\sudokus> java psnode new Alumno1
1.4. c:\pruebas\sudokus> java psnode new Alumno2

2.1. c:\pruebas\sudokus> start java launcher platform localhost

                  (NOTA.1: Se lanza el agente RMA de la plataforma JADE para asistir
                           durante las pruebas de concepto. No es necesario detener
                           la plataforma JADE durante las pruebas).
                  (NOTA.2: Si se posee conexión a la Intranet corporativa o del aula, las
                           pruebas de concepto pueden ser realizadas entonces desde
                           dos o más ordenadores.
```

```
                          Se utilizará el nombre del host-anfitrión para lanzar la
                          plataforma y para lanzar agentes desde los distintos equipos.
                          e.g.: start java launcher platform poseidon  4455)

2.2. c:\pruebas\sudokus>  start java launcher pocket  localhost Robot01  clips

3.1. Presionar botón: [n-5]      (para llevar al agente hasta su estado opertaivo).
3.2. En la pestaña: CLIPS: Asynchronous Shell, teclear:

     (facts)

3.3. Realizar la combinación de teclas: [Ctrl+Myús+Enter] o presionar: botón "Execute!"
  Observar resultados
3.4. Presionar botón: [Exit]                          (Para cerrar y salir del agente)
```

A continuación, se alimentan los agentes con el Sistema de Producción. Para ello es necesario mover el directorio [`Examples`] completo hasta el espacio `/engine/` perteneciente al nodo del agente [`Robot01`], y repetir la prueba desde el interior del *agente-nodo*. (Se propone realizar dicho desplazamiento con *Explorer*).

```
1. c:\pruebas\sudokus> explorer .
2. c:\pruebas\sudokus> Desplazar (arrastrar con ratón) el directorio Examples hasta ...

                    nodes/Robot01/engine/.
```

Realizar el *Test de estrés* anterior pero, sobre el *agente* [`Robot01`]:

```
1.1. c:\pruebas\sudokus> start java launcher pocket localhost Robot clips

2.1. Presionar botón: [n-5]

2.2. En pestaña: CLIPS: Asynchronous Shell, teclear:

     (batch* "nodes/Robot/engine/Examples/sudoku/runbench.bat" )

2.3. Presionar: [Ctrl+Mayús+Enter] o, el botón [Execute!]:

En la consola de respuestas se obtiene:      (NOTA: puede variar en función de la CPU).

CLIPS>
%%%%%%%***********
%%*******************************************************
***********
*
7515 rules fired        Run time is 0.555031776428223 seconds.
13539.7653236379 rules per second.
656 mean number of facts (1493 maximum).
1 mean number of instances (1 maximum).
1226 mean number of activations (5503 maximum).
8746 rules fired        Run time is 2.03611707687378 seconds.
4295.43079783431 rules per second.
685 mean number of facts (1502 maximum).
1 mean number of instances (1 maximum).
1684 mean number of activations (5503 maximum).
TRUE

3. Presionar botón: [Exit] para cerrar el agente.
```

6.3.3 Resumen de los procesos realizados

En los dos apartados anteriores se han empleado los *productos* (Tabla 5.1, pág.65) siguientes:

- La *utilidad de instalación* [`dps-frameworkBuilder.jar`] para construir un nuevo *Directorio de aplicación* que se ha llamado [`sudokus`] donde se ha iniciado el desarrollo.

- La *utilidad auxiliar Shell*, sobre el interior del *Directorio de aplicación* [`sudokus`], para lanzar una Shell de CLIPS sobre una máquina virtual de Java y facilitar la apertura de una sesión de Shell de CLIPS 6.31, en cualquiera de los sistemas operativos donde se haya creado el *Directorio de aplicación* [`sudokus`].

- La *utilidad auxiliar psnode*, para generar varios *nodos* denominados [`Robot01,04, Alumno01, etc.`] sobre la estructura del *Directorio de aplicación*, y donde posteriormente, se ha reubicado el *Sistema de Producción* de cálculo de puzzles de lógica de tipo *Sudoku*.

- La *utilidad auxiliar launcher*, que ha facilitado la llamada a la plataforma JADE con la secuencia de parámetros necesaria para lanzar al *agente* [`Robot01`], sobre un *Entorno de desarrollo* como agente estándar en la plataforma JADE en un nuevo contenedor.

- El *Entorno de desarrollo* de *agentes*, invocado a través de la utilidad *launcher* con el parámetro denominado [`pocket`].

Los pasos realizados han permitido tener una primera toma de contacto con el proceso de transformación de una aplicación al paradigma multiagente, en este caso, para desplegar un *Sistema de Producción* de resolución de Sudokus desarrollado en CLIPS. Sin embargo, son necesarios procesos adicionales para transformar el *agente* a través de la modificación de las propiedades de su *nodo*, para obtener una versión distribuida de esa aplicación sobre JADE; de manera que preste servicios de resolución a distintos agentes y pueda migrar entre distintos contenedores de la plataforma con sus conocimientos, se propone: construir un nuevo *Directorio de aplicación* con el nombre [`sudoku2`] y realizar de nuevo los siguientes procesos:

- Primero, simplificar en lo posible, el contenido del directorio `/engine/` del agente que vaya a convertirse en la pieza responsable de solventar Sudokus, de manera que sólo contenga los archivos necesarios: [`solve.clp`, `sudoku.clp` y `output-frills.clp`].

- Incorporar, en la secuencia de inicialización de dicho agente, el proceso de carga de dichas capacidades de resolución para que permanezcan siempre activas y puedan ser atendidas las demandas de resolución de cualquier puzzle, no solamente del archivo de análisis *Benchmark* lanzado en el apartado de comprobación, sino cualquier crucigrama que reciba. Además, el agente estará operativo desde el instante de su despliegue.

- Realizar múltiples Test de funcionamiento desde otros agentes del mismo *Directorio de aplicación* [`sudoku2`], con consultas al agente [`Robot`], someterlo a máximo estrés con crucigramas realmente complejos (series `grid4x4-pxx.clp` y `grid5x5-pxx.clp`), y verifi-

car cómo, las respuestas que se devuelven; son correctas, están dirigidas a cada solicitante y no bloquean la capacidad de respuesta y diálogo del agente consultado.

- Crear un proceso para construir mensajes ACL que contengan archivos de crucigramas para remitir, desde agentes [solicitantes], hacia alguno de los agentes [`Robot`].

- Y para concluir, integrar dicho proceso, en un *Protocolo de Interacción* de respuesta racional, también conocidos como protocolos tipo *FIPA-Request*, que permita diálogos formados por la pauta de conversación siguiente: (1) Solicitud con archivo adjunto de crucigramas al agente dedicado a resolución de sudokus, (2) Respuesta con la confirmación de la recepción del crucigrama y (3) Respuesta con la solución del crucigrama.

6.3.4 Simplificación del directorio: /engine/

Se crea un nuevo *Directorio de aplicación* para la segunda parte de la prueba de concepto. En primer lugar, se simplificará en lo posible el contenido del directorio `/engine/` del agente responsable de solventar Sudokus. Los archivos necesarios son los siguientes: [`solve.clp, sudoku.clp y output-frills.clp`]:

```
1. c:\pruebas> java -jar  dps-frameworkBuilder-0.9.1.jar --new=sudokus2 --no-doc

2. c:\pruebas> cd sudokus2

3. c:\pruebas\sudokus2\> java psnode new Robot

4. c:\pruebas\sudokus2\> cd nodes\Robot\engine
5. c:\pruebas\sudokus2\nodes\Robot\engine\> [Limpiar contenido y Copiar]:
                                            solve.clp
                                            sudoku.clp
                                            output-frills.clp
6. c:\pruebas\sudokus2\nodes\Robot\engine\> [Copiar dos crucigramas para comprobación]:
                                            grid3x3-p1.clp
                                            grid4x4-p1.clp
```

6.3.5 Incorporar capacidades de inferencia a los agentes

Editar el archivo de carga de *capacidades*, [`bhv.acpb.bsh`], destinado a incorporar en los *agentes* sus capacidades en el instante de su inicialización. Ese archivo puede ser editado en la pestaña de *Edición de comportamientos* cuando el agente [`Robot`] se arranca en el *Entorno de desarrollo*:

```
1. c:\pruebas\sudokus2\> start java launcher pocket localhost Robot clips

2. En la pestaña Editor de comportamientos:  Doble-Clic en archivo: [bhv.acpb.bsh]
   Crear una variable Array de cadenas, de la forma:

initLoad = new String[][] {
    {"MAKE_CLEAR","null"
   ,{"LOAD_FILE",myNode.relativeEnginePath+"sudoku.clp"}
   ,{"LOAD_FILE",myNode.relativeEnginePath+"solve.clp"}
   ,{"LOAD_FILE",myNode.relativeEnginePath+"output-frills.clp"}
   ,{"MAKE_RESET","null"}
   ,{"RUN_INFINITELY","null"}
};
```

```
3. En la pestaña Editor de comportamientos:
   Finalizar la edición con:
   Presionar [Save]

4. En la pestaña Editor de comportamientos. Doble-Clic en archivo: [node.properties]
   Establecer la variable:    ps.node.behaviour-acpb  = 1
   Presionar [Save]

5. En la barra superior del agente-nodo:  [Robot]
   Presionar [n-6!]:
   NOTA: Se re-inicia el agente [Robot] en caliente y se incorporan los cambios realizados.

6. En la barra superior del agente-nodo:
   Presionar [n-5]:
   NOTA: Para llevar al agente a su estado de prestación de servicios.

7. En la pestaña CLIPS: Shell Asíncrona:
   Teclear:
   (load "nodes/Robot/engine/grid3x3-p1.clp" )      y presionar: [Ctrl+Mayús.+Enter]

8 En la pestaña CLIPS: Shell Asíncrona:
   Teclear:
   (reset)                                          y presionar: [Ctrl+Mayús.+Enter]

9 En la pestaña CLIPS: Shell Asíncrona:
   Teclear:
   (run)                                            y presionar: [Ctrl+Mayús.+Enter]

   Verificar resultados.
```

6.3.6 Comprobación de acceso remoto al agente [Robot]

A continuación, para verificar que se puede acceder y obtener soluciones del agente [Robot] desde otros agentes del mismo *Directorio de aplicación*:

```
1. c:\pruebas\sudokus2\> java psnode new Alumno1

3. c:\pruebas\sudokus2\> start java launcher pocket localhost Alumno1 clips

5. Desde la barra superior:
   Presionar [n-5]
   NOTA: Para elevar el nivel de ejecución a estándar y permitir la comunicación.

6. Desde la pestaña CLIPS: Shell Asíncrona de agente Alumno1:
   Seleccionar a agente Robot en la lista de la izquierda y realizar los mismos
   pasos 7, 8 y 9 del apartado anterior, con un crucigrama distinto: grid4x4-p1.clp.

   Comprobar que el agente Robot, reponde adecuadamente al agente Alumno1.

7. Observar sobre agente RMA de la plataforma JADE que:
   Ambos agentes se encuentran en Contenedores separados.
```

6.3.7 Crear un *nuevo mensaje* para remitir crucigramas

Es necesario crear un nuevo método sobre la instancia de agente [Alumno1] que devuelva un objeto tipo mensajes ACL, y que admita los parámetros: (1) nombre de la ubicación del archivo de crucigramas a enviar y, (2) nombre el agente encargado de resolver ese crucigrama.

```
1. Desde la pestaña Editor de Comportamientos [Alumno1]:
   Junto al botón [Save], indicar nombre del archivo en el cuadro de Texto:

   [makePuzzleREQUEST.bsh]
   A. Borrar el contenido que existiese en la pantalla del Editor.
```

B. Incorporar a la pantalla del Editor la siguiente implementación:

```
public ACLMessage makePuzzleREQUEST( String nameOfPuzzle , String agentSolver ) {

     placeOfPuzzle   = myNode.thisNodeEngineDir.getAbsolutePath() + File.separator ;
     File puzzle     = new File( placeOfPuzzle + nameOfPuzzle );
     if ( !puzzle.exists()) { return null; }

     ACLMessage msg  = new ACLMessage( ACLMessage.REQUEST);
     try {
             FileInputStream  fis     = new FileInputStream(puzzle);
             int byteLength           = (int) puzzle.length();
             byte[] fileContent       = new byte[byteLength];
             fis.read(fileContent, 0, byteLength);
             msg.setByteSequenceContent(fileContent);
             fis.close();

             msg.setLanguage( FIPANames.ACLCodec.BITEFFICIENT  );

     } catch (IOException e) {
             msg.setContent (         nameOfPuzzle + ": error on reading!" );
             msg.setLanguage( FIPANames.ACLCodec.STRING );
     }

     AID solver  = new AID( agentSolver, AID.ISLOCALNAME       );
     msg.addReceiver (        solver                           );
     msg.setSender   (        myAgent.getAID()                 );
     msg.setReplyWith(        "SudokuPuzzlesRequest"           );

     msg.addUserDefinedParameter( "nameOfPuzzle", nameOfPuzzle );

     return msg;
 }
```

Presionar en botón [**Save**]
NOTA: ¡ **MUY IMPORTANTE** !

2. Desde la pestaña Shell-JADE del agente [Alumno1]:

```
[ Agent ]://Alumno1/config/$ > source( "makePuzzleREQUEST.bsh" );
[ Agent ]://Alumno1/config/$ > dir("../engine/"); (para comprobar lista de puzzles)
```

Crear un objeto mensaje y almacenar en la variable: c1

```
[ Agent ]://Alumno1/config/$ > c1 = makePuzzleREQUEST( "grid4x4-p17.clp", "Robot");
```

Se genera una salida (a causa del Echo activado en Shell-JADE):

```
<(REQUEST
    :sender    ( agent-identifier :name Alumno1@localhost:4455/JADE
                                  :addresses (sequence http://poseidon:64466/acc ))
    :receiver  (set ( agent-identifier
                                  :name Robot@POSEIDON:4455/JADE ) )
   :X-JADE-Encoding Base64
   :content        "KGRlZnJ1bGUgZ3JpZC12YWx1ZXMKCiAgID9mIDwtIChwaG.....
   :reply-with     SudokuPuzzlesRequest
   :language       fipa.acl.rep.bitefficient.std
   :X-nameOfPuzzle grid4x4-p17.clp
 )>
```

3. Desde pestaña Shell-JADE del agente [Alumno1]:

Remitir el mensaje al agente [Robot]

```
[ Agent ]://Alumno1/config/$ > myAgent .send( c1 );
```

4. Desde pestaña Shell-JADE del agente [Robot]:

Comprobar que el mensaje ha llegado y que puede obtenerse:

```
[ Agent ]://Robot/config/$ > myAgent .getCurQueueSize();
[ Agent ]://Robot/config/$ > r1 = myAgent .receive();
```

Denotar que, si falla la incorporación en formato de *Byte Stream* al contenido del mensaje, se realiza el cambio del valor del campo `language` y se remite una cadena informando del error.

6.3.8 *Protocolo de interacción* entre [AlumnoN] vs. [Robot]

La plataforma JADE, aporta la clase [`SimpeAchieveREInitiator`] para crear con fiabilidad y bajo consumo de recursos, conversaciones que presentan *Rational-Effe*ct que pertenecen al grupo de protocolos de interacción tipo *FIPA-Request*. Antes de crear una clase que extienda a `SimpeAchieveREInitiator` para iniciar los diálogos entre agentes, pueden comprobarse de forma manual sendas clases sobre la Shell-JADE: una que inicia la conversación en el agente [`Alumno1`] y la segunda en el agente [`Robot`] con un objeto tipo [`SimpleAchieveREResponder`] que permanece a la escucha. Destacar que ese, ha sido el modelo seguido para desarrollar agentes y comportamientos; dos agentes abiertos en sus *Entornos de desarrollo* pertenecientes a un mismo *Directorio de aplicación* que intercambiar mensajes entre si mismos.

```
1. Desde la pestaña Shell-JADE del [Alumno1]:          Como primera comprobación:
   [ Agent ]://Alumno1/config/$ > b = new SimpleAchieveREInitiator( myAgent , c1 );

2. Desde la pestaña Shell-JADE del [Alumno1]:
   [ Agent ]://Alumno1/config/$ > myAgent .addBehaviour( b );

3. Desde la pestaña Shell-JADE del [Robot]:            (Se crea un Template para filtrado).
   [ Agent ]://Robot/config/$ > myAgent .getCurQueueSize() ;
   <1>...Esto indica que el mensaje ha llegado a la bandeja de entradas.

   [ Agent ]://Robot/config/$ > mt = MessageTemplate.and(
                                    MessageTemplate.MatchReplyWith("SudokuPuzzlesRequest"),
                                    MessageTemplate.MatchPerformative(ACLMessage.REQUEST));

4. Desde la pestaña Shell-JADE del [Robot]:            (Se crea el Respondedor).
  [ Agent ]://Robot/config/$ > b = new SimpleAchieveREResponder(myAgent, mt);
  [ Agent ]://Robot/config/$ > myAgent .addBehaviour( b );


  NOTA.1: Con esto queda comprobado que funcionará adecuadamente Iniciador y Respondedor
        creados con las clases implementadas por JADE. Ahora, es necesario crear sendas
        clases que las extiendan e incorporar el código necesario en sus métodos
        de manipulación de reacción al AGREE y al mensaje tipo INFORM con los resultados.

  NOTA.2: Establecer la propiedad verbose = 1 en node.properties y reiniciar en caliente al
        agente Robot [n-6!]. Se obtiene así una visión completa del paso de mensajes.
```

Sobre la consola del agente [`Robot`] se puede comprobar con [`myAgent .getCurQueueSize ();`] que el mensaje ha sido transmitido por el [`Alumno1`] y que ha llegado a destino. Una vez que se crea el filtro [`mt`] y se inicializa el objeto [`SimpleAchieveREResponder`] para permanecer a la escucha, todos los mensajes con esas características son captados por agente [`Robot`], respondidos con AGREE si son correctos y notificados con INFORM adjuntando resultados.

Esta prueba de concepto ha tratado de ofrecer una visión general de las distintas herramientas aportadas para llevar a cabo una transformación y, el posterior despliegue, de un *Sistema de Producción* sobre la plataforma multiagente JADE. En el directorio `/config/` se encuentran las dos clases propuestas ya implementadas para mantener el diálogo. Pueden abrirse y edi-

tarse en la pestaña de *Edición de comportamientos* y para realizar una carga manual desde la Shell-JADE pueden seguirse las sentencias siguientes:

```
1. Desde la pestaña Shell-JADE del [Alumno1]:
   [ Agent ]://Alumno1/config/$ > source( "class.SudokuREInitiator.bsh");
   [ Agent ]://Alumno1/config/$ > b = new SudokuREInitiator( myAgent , c1 );
2. Desde la pestaña Shell-JADE del [Alumno1]:
  [ Agent ]://Alumno1/config/$ > b .setBehaviourName( "puzzIN" );
  [ Agent ]://Alumno1/config/$ > myAgent .addBehaviour( b );
3. Desde la pestaña Shell-JADE del [Robot]:
   [ Agent ]://Robot/config/$ > source( "class.SudokuREResponder.bsh");
   [ Agent ]://Robot/config/$ > mt = MessageTemplate.and(
                                    MessageTemplate.MatchReplyWith("SudokuPuzzlesRequest"),
                                    MessageTemplate.MatchPerformative(ACLMessage.REQUEST));

   [ Agent ]://Robot/config/$ > myAgent .removeBehaviour( b2 );
   [ Agent ]://Robot/config/$ > b2 = new SudokuREResponder( myAgent , mt );
   [ Agent ]://Robot/config/$ > b2 .setBehaviourName( "puzzRE" );
   [ Agent ]://Robot/config/$ > myAgent .addBehaviour( b2 );
```

Para automatizar: la carga de esas clases, el template de filtrado y la creación de objetos comportamiento en los agentes durante la inicialización, se propone el mismo método utilizado en los archivos: e.g. [bhv.xxxx.bsh], para crear los objetos e incorporar dichos *Behaviours* al planificador de tareas de cada agente y en el *nivel de ejecución* adecuado.

6.3.9 Cómo cambiar la tecnología de motor de inferencia

Una propiedad del *Sistema de Producción* para el cálculo de *Sudoku* [Riley, 2016a] es su portabilidad, esto es: puede funcionar sobre motores CLIPS y sobre motores *Jess*. Aprovechando dicha capacidad, se muestra cómo lanzar los agentes [Robot] y [Alumno1] conectados a motores *Jess* y cómo realizar *hacking* (de investigación) en el directorio /lib/ y extraer conclusiones para comprender más profundamente la configuración del *modo-infraestructura*.

Para verificar que realmente los agentes pueden intercambiar su motor de inferencia en esta prueba de concepto, se pueden eliminar del directorio /lib/ del *Directorio de aplicación* denominado [sudokus2] los subdirectorios: /jena, /w32, /w64, /mac, /ux32, /ux64, y [CLIPSJNI.jar]. Tras esa operación, ya sólo será posible conectar motores *Jess* a los agentes de ese *Directorio de aplicación* [Sudokus2]. Para indicar a los agentes que empleen este segundo tipo de motor de inferencia se utiliza, durante el instante del lanzamiento, el parámetro con la palabra `jess` como se muestra a continuación:

```
1. c:\pruebas\sudokus2\> start java launcher pocket localhost Robot  jess
2. c:\pruebas\sudokus2\> start java launcher pocket localhost Alumno1 jess
4. Presionar [n-5] en la barra de botones del agente Robot.
5. Presionar [n-5] en la barra de botones del agente Alumno1.
6. Desde JESS: Shell Asíncrona del agente Alumno1:
   Seleccionar el agente [Robot] al que lanzar las consultas:
   Teclear y presionar a continuación [Ctrl+Mayús.+Enter]

   (batch "nodes/Robot/engine/grid4x4-p1.clp" )

7. Desde JESS: Shell Asíncrona del agente Alumno1:
   Teclear y presionar a continuación [Ctrl+Mayús.+Enter]
   (reset)
8. Desde JESS: Shell Asíncrona del agente Alumno1:
   Teclear y presionar a continuación [Ctrl+Mayús.+Enter]
   (run)
```

Destacar que, cuando el *Actor-Humano* es quién interactúa desde la *Consola Asíncrona* del *Entorno de desarrollo*, es necesario utilizar (`batch "nodes/Robot/engine/."`) con motor *Jess* conectado al agente, en lugar de (`load "nodes/Robot/engine/..."`) que se empleaba en el caso de motor CLIPS. Excepto esa diferencia para cargar el archivo de crucigramas en el motor, el código de resolución de crucigramas tipo *Sudoku* es íntegramente portable y operativo en ambos tipos de motores CLIPS y *Jess* y por tanto, ha sido posible emplear agentes con motores de ambas tecnologías en esta prueba de concepto.

Indicar finalmente, que al remitir el archivo de crucigramas adjunto en el contenido de un mensaje al agente, esas diferencias presentes la realizar la carga por el cambio de motor, queda asumida por la clase [`class.SudokuREResponder.bsh`] pues, emite la solicitud al motor de inferencia del agente mediante el método [`LOAD_FILE`], que es genérico para todas las clases que implementan la interface RBEngine y que aplica el método de carga del archivo de ejemplo transmitido, en función de la tecnología de motor de inferencia del agente [`Robot`].

6.3.10 Resumen de la integración de resolución de Sudokus

Como resumen de la prueba de concepto se destacan estas conclusiones preliminares:

1. Los *agentes* [`Alumno-N`] obtienen soluciones mediante *consultas* a un tercero.
2. Las *consultas* se realizan mediante un *protocolo de interacción* del tipo *FIPA-Request*.
3. El *Sistema de Producción* se integra por entero en el agente que recibe las consultas.
4. La capacidad de intercambiar el motor de inferencia es una propiedad de los agentes.
5. La portabilidad del código es una característica del *Sistema de Producción* seleccionado.
6. Los agente tipo [`Alumno-N`], en realidad, no requerirían incorporar ningún motor.
7. Las respuestas del motor son transmitidas y por ello, podrían ser *re-evaluadas*.

6.4 Prueba de concepto: Sistema O-DEVICE

La recomendación de W3C para crear y compartir significados semánticos para la Web es utilizar archivos en formato OWL basados en el *formalismo de la representación de conocimiento* mediante el formalismo de *Lógica Descriptiva* (DL). Ese formato es utilizado por los *Razonadores de Ontologías* para obtener información semántica y deducir nuevo conocimiento. El modelado en OWL se realiza siempre en base a tres elementos: *conceptos*, *propiedades* e *instancias*. Los dos primeros constituyen el conocimiento terminológico empleado por la ontología llamado (*schema*) que, progresivamente se completa con la asignación de contextos de significado en los cuales dichos términos pueden ser empleados.

El *Sistema* denominado O-DEVICE [Meditskos & Bassiliades, 2008], se encuentra disponible en SourceForge[40]. Su diferencia fundamental frente a los *Razonadores de ontologías* es que aporta un modelo completo de *representación del conocimiento*, extraído de una ontología en OWL y transformado al formato de *reglas* y *objetos* sobre la *Knowledge-Base* de un

[40]**O-DEVICE**: https://ayera.dl.sourceforge.net/project/o-device/odevice_v2.rar

Sistema de Producción en CLIPS. Pero eso sí; ese modelo es estático y orientado para la explotación mediante consultas que requieran altos rendimientos de velocidad de respuesta. Para alcanzar y ofrecer dicho modelo, el sistema O-DEVICE aplica una serie de transformaciones, primero en objetos y luego en *reglas* deductivas (o, *producciones*). La arquitectura de O-DEVICE consta de 5 módulos:

1. Programa Cargador de Reglas.
2. Cargador de Tripletas OWL.
3. Traductor en Reglas Deductivas (o, *reglas de producción*).
4. Traductor de Tripletas OWL.
5. Extractor de OWL.

La publicación [Meditskos & Bassiliades, 2008], expone dicha arquitectura modular en la Figura 6.1 donde además, los autores resumen el flujo seguido por los datos desde el formato original en OWL hasta su forma final en *reglas* y *objetos*. Ese flujo consta de 6 procesos seguidos de manera secuencial: (**1**) entrada de usuario dirigida al programa de carga de reglas, (**2**) Analizador sintáctico que traslada el documento en OWL a Tripletas, (**3**) traducción de esas tripletas en clases, propiedades y objetos de COOL, (**4**) conversión a reglas de producción de CLIPS, (**5**) se ejecuta CLIPS para generar los objetos que constituyen el resultado del emparejamiento de las reglas y (**6**) un último proceso destinado a la exportación.

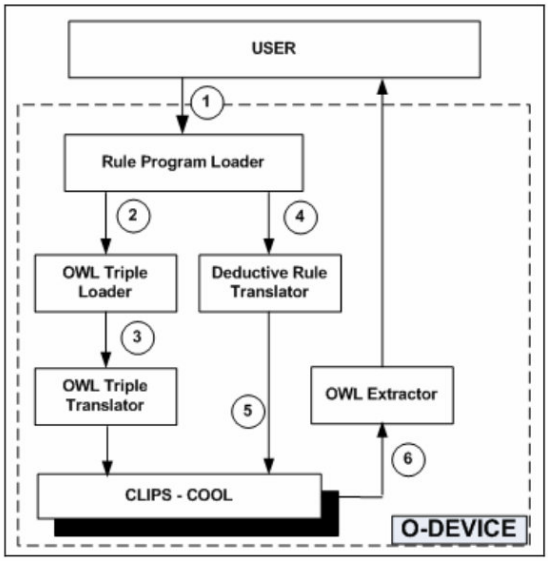


Figura 6.1: Arquitectura de O-DEVICE [Meditskos & Bassiliades, 2008]

En esta *prueba de concepto* se siguen pautas similares al caso anterior: *a*) emplear un ordenador personal y, *b*) proceder con los pasos mínimos para transformar el *Sistema* O-DEVICE de *Georgios Meditskos*, al escenario multiagente empleando las herramientas aportadas en la investigación. Los archivos necesarios son el instalador de *Directorios de aplicación* [`dps-frameworkBuilder-0.9.1.jar`] y O-DEVICE [`odevice_v2.rar`], ambos se descargan sobre un directorio denominado [`pruebas`]. Para iniciar el proceso, se construye un *Directorio de aplicación* para obtener acceso a las *utilidades auxiliares*.

6.4.1 Requisitos para desplegar O-DEVICE en: *java shell clips*

El primer proceso es la demostración de cómo desplegar el Sistema O-DEVICE en una instancia de *Java* con motor CLIPS. Para ello se emplea la *utilidad auxiliar*: *shell clips*. Se puede observar que ese primer proceso es tosco pero, aporta la visión de conjunto sobre: el flujo que sigue la información en el Sistema O-DEVICE, las adaptaciones necesarias para transformar al formato de objetos, las librerías que deben ser utilizadas para que opere en cualquier sistema operativo, una estimación preliminar de los cambios que serán necesarios en los *nodos* de agentes donde quede integrado O-DEVICE, etc. Se procede entonces con la creación de un nuevo *Directorio de aplicación*, e.g.: [reasoners].

```
1.   c:\pruebas> java -jar  dps-frameworkBuilder-0.9.1.jar --new=reasoners --no-doc

2.1. c:\pruebas> cd reasoners
2.2. c:\pruebas\reasoners> set CLASSPATH=lib/*;lib/jena/*;

3.1. c:\pruebas\reasoners> unrar x odevice_v2.rar
                                        (NOTA: Sólo es necesario el directorio odevice
                                        pues, CLIPSDOS.exe y CLIPSWin.exe, no pueden ser
                                        utilizados por los agentes, ni por shell-clips)
3.2. c:\pruebas\reasoners> cd odevice_v2
3.3. c:\pruebas\reasoners\odevice_v2> mv odevice ..
3.4. c:\pruebas\reasoners\odevice_v2> cd ..
3.5. c:\pruebas\reasoners\> rm -rf odevice_v2/

4. c:\pruebas\reasoners> vim  odevice/config.bat        (Editar el archivo config.bat)
   Modificar línea 15:                                  (Por el momento, hard-code!)
   (defglobal ?*odevice-folder* = "c:/pruebas/reasoners/odevice/")

5. c:\pruebas\reasoners> vim  odevice/prepare.bat       (Editar el archivo prepare.bat)
   Crear línea 24:
   (load-ontology "file:///c:/pruebas/reasoners/pizza.owl" )

6. c:\pruebas\reasoners> copy  nodes\Pocket\var\pizza.owl .

7. c:\pruebas\reasoners> vim  odevice/src/functions.clp  (Editar el archivo functions.clp)

   Modificar línea 948:  Descomentar
   Comentar  línea 943: ;(bind ?command (str-cat "java -jar " ?*odevice-folder* "j2cf.jar"))

   No sería adecuado, lanzar: (system ?command) desde un Agente en la plataforma JADE.
   Se propone temporalmente, para que al menos opere en la Shell-clips la solución:

   Crear     línea 944: (bind ?command (str-cat "javaw edu.csd.auth.gr.clips.tools.J2CF "))

8. c:\pruebas\reasoners> java shell clips

   CLIPS> (batch* odevice/config.bat  )
   CLIPS> (batch* odevice/prepare.bat )
   CLIPS> (clear)

   CLIPS> (batch* odevice/config.bat  )
   CLIPS> (batch* odevice/exec.bat    )

   CLIPS> (facts)
   CLIPS> (browse-classes)

9. CLIPS> (exit)

   Observar: contenido de los directorios: odevice/bundle  y  odevice/triple-facts
```

6.4.2 Creación de nuevos *nodos* para los agentes ODEVnnn

Desde el *Directorio de aplicación* se procede con la creación de *nodos* nuevos con la *utilidad auxiliar psnode* que permitirán, a los agentes con dichos nombres, deplegarse en JADE y captar las variables y los recursos contenidos en los *nodos*. Una vez creados, se procede con la reubicación de los archivos del Sistema O-DEVICE sobre el directorio /engine/ siguiendo las siguientes propuestas de reordenación de archivos y directorios:

```
1.1. c:\pruebas\reasoners> java psnode new ODEV101
1.2. c:\pruebas\reasoners> java psnode new ODEV201
1.3. c:\pruebas\reasoners> java psnode new Helper04
```

```
     Desplazar el sistema O-DEVICE hacia el nodo ODEV101
     y limpiar el Directorio de aplicación:

2.1. c:\pruebas\reasoners> cd odevice
2.2. c:\pruebas\reasoners\odevice>  mv * ../nodes/ODEV101/engine/.
2.3. c:\pruebas\reasoners\odevice>  cd ..
2.4. c:\pruebas\reasoners> rm -rf odevice/
2.5. c:\pruebas\reasoners> rm -rf nodes/ODEV101/engine/lib/
2.6. c:\pruebas\reasoners> rm -f  nodes/ODEV101/engine/j2cf.jar

     Aspecto final del nodo destinado a agentes que integran el sistema O-DEVICE.

     reasoners/nodes/ODEV101
                        +---config*
                        ¦
                        +---engine*
                        ¦   +---bundle
                        ¦   +---src
                        ¦   +---triple-facts
                        ¦   +---logs*
                        ¦
                        +---var*

                                    (*) Directorios que forman parte original del nodo.
```

6.4.3 Asignación dinámica del directorio: ?*odevice-folder*

Sobre el subdirectorio /engine/ del *nodo* empleado por el agente, [ODEV101] se encuentra el archivo [config.bat]. En la línea 15, se encuentra el valor de la carpeta empleada por el sistema O-DEVICE para localizar su estructura.

```
1. c:\pruebas\reasoners\nodes\ODEV101\engine\> vim config.bat (Editar el archivo config.bat)

   Comentar la Línea 15:
   ;(defglobal ?*odevice-folder* = "c:/Users/George/Desktop/@work/_code/odevice/")
```

Para integrar al paradigma multiagente el sistema O-DEVICE, su directorio debe ser gestionado por el agente. La variable [?*odevice-folder*] es asignada por el agente durante su proceso de inicialización. Esto se logra incorporando al *comportamiento de carga de capacidades* las siguientes sentencias. Puede comprobarse el proceso sobre el agente [ODEV101] en tiempo de ejecución empleando la consola *Shell-JADE* y el *Editor de comportamientos*:

```
2.0. c:\pruebas\reasoners\> start java launcher platform localhost   (Si no estaba arrancada)
2.1. c:\pruebas\reasoners\> start java launcher pocket   localhost ODEV101 clips
```

```
3. Sobre la Shell-JADE comprobar el resultado de las siguientes sentencias:
```

```
  [ Agent ]://ODEV101/config/$ > oDeviceFolderAgent  = nodeEnginePath .replace("\\","/");
  resultado: <C:/pruebas/reasoners/nodes/ODEV101/engine/>

  [ Agent ]://ODEV101/config/$ > oDeviceFolderGlobal =
                   "(defglobal ?*odevice-folder* = " + "\"" + oDeviceFolderAgent + "\" )";

  resultado: <(defglobal ?*odevice-folder* = "C:/pruebas/reasoners/nodes/ODEV101/engine/" )>
```

```
4. Sobre el Editor de comportamientos:
   Editar                  : bhv.acpb.bsh        (Acrónimo de: agent-capabilities script)
   NOTA: insertar al principio del archivo.

oDeviceFolder  =  nodeEnginePath .replace("\\","/");
oDeviceFolderGlobal   = "(defglobal ?*odevice-folder* = " + "\"" + oDeviceFolder + "\" )";

initLoad = new String[][] {};

switch (myNode.engineSelected) {
       case "CLIPS":
           initLoad = new String[][] {
                  { "MAKE_CLEAR", "null" }
                 ,{ "MAKE_BUILD", oDeviceFolderGlobal}
            };
           break;
 ...
 IMPORTANTE: Presionar botón [Save]
```

```
5. Sobre el Editor de comportamientos:  Editar node.properties y establecer...

   ps.node.behaviour-acpb  = 1
 ...
 IMPORTANTE: Presionar botón [Save]
```

```
6. Desde Pestaña Editor Expert System Files: seleccionar config.bat

   Comentar línea 15:   ;(defglobal ?*odevice-folder* = "c:/pruebas/reasoners/odevice/")
   IMPORTANTE: Presionar botón [Save]
```

```
7.1. Realizar re-inicio del agente ODEV101:

   Presionar: [n-6!]
   Observar la salida de consola:       (Se han ejecutado comandos de inicialización)

7.2. Desde pestaña:  CLIPS:Shell - Síncrona, en el Agente ODEV101...

   CLIPS > (list-defglobals)
   CLIPS > (printout -t ?*odevice-folder*  crlf)
```

De esa forma, el agente al inicializarse, establece la variable global de la carpeta del sistema O-DEVICE respecto a su instancia de motor de inferencia CLIPS. Esa variable será utilizada posteriormente por el Sistema O-DEVICE para ubicar sus archivos de resultados intermedios con las traducciones de la ontología. Como demostración de ello, en la misma pestaña de *Shell Síncrona* del motor CLIPS del agente, puede cargase todo el modelo de objetos COOL, con *hechos* y *reglas* inferidas después de la trasformación del apartado anterior de la ontología [pizza.owl] sobre el motor del agente mediante las siguientes sentencias en el *Entorno de desarrollo* del agente ODEV101:

```
8. Desde pestaña:  CLIPS:Shell - Síncrona en el Agente ODEV101
   CLIPS> (batch* nodes/ODEV101/engine/config.bat  )
   CLIPS> (batch* nodes/ODEV101/engine/exec.bat    )
   CLIPS> (browse-classes)
```

6.4.4 Resumen y evaluación de procesos realizados

Los dos apartados anteriores solventan la primera fase de la integración de O-DEVICE en agentes JADE con motores CLIPS. Esas soluciones han sido producto de las decisiones de diseño siguientes, encaminadas a alcanzar la integración y mantener la autonomía del agente:

- La *agente* debe asignar en tiempo de ejecución el valor a la variable global utilizada por el sistema O-DEVICE sobre su motor de inferencia CLIPS. Esa variable debe contener un valor String con el camino completo hasta el directorio /`engine`/, donde se alojan los archivos del sistema experto de O-DEVICE. Su nombre: [`?*odevice-folder*`].

- Dicha asignación se realiza en el nivel de ejecución [`n-1`] del agente, por medio del archivo de carga [`bhv.acpb.bsh`], el cual emite sentencias de carga al motor CLIPS para que construya, mediante MAKE_BUILD, esa variable global en es motor de inferencia.

- Se ha comentado la línea número 15 del archivo /`config.bat`/ para que no sea sobrescrito el valor de [`?*odevice-folder*`], durante la inicialización de O-DEVICE.

- El procedimiento de transformación y traducción de la ontología se ejecuta secuencialmente en procesos lanzados desde el archivo /`prepare.bat`/, hasta la función de transformación [`deffunction create-facts`] en la línea 942 de [`functions.clp`], modificada para invocar el método `main()` de J2FC de manera cualificada.

- El modelo de ejecución de la línea 949 de [`functions.clp`] no es viable en un agente porque, no debe lanzarse una nueva máquina *Java* desde un agente de la plataforma.

A partir de esas limitaciones y decisiones de diseño, se posee una visión de conjunto de los problemas pendientes para integrar O-DEVICE en agentes. Esos problemas son:

- Es necesario capturar el nombre de la ontología a transformar desde un mensaje remitido al agente y emplear dicho valor, dentro del proceso [`prepare.bat`] de manera dinámica por el agente pues la función [`load-ontology`] espera una *Lista* `String`[] de parámetros.

- Debe transformarse la ontología/s a formato de Tripletas de *hechos*, con los recursos y de manera acorde con el modelo de ejecución propio de los agentes en JADE.

- Deben recargarse la *Knowledge-Base* y la *Working-Memory* del agente cuando haya finalizado una transformación de una ontología, para que pueda asistir a las consultas realizadas a su base de conocimientos tan pronto como sea posible.

- Deben cargarse los archivos presentes en /`bundle`/ y /`triple-facts`/ en el proceso de inicialización del agente, de forma que se encuentre operativo con la última transformación solicitada, después de haber permanecido desconectado de la plataforma multiagente.

Se resume a continuación cómo se han solventado estos problemas con las herramientas aportadas y cómo integrar O-DEVICE en agentes y permitir a los mismos: autonomía sobre

las actuaciones realizadas, estabilidad operativa con los analizadores sintácticos de JENA y, los mecanismos de construcción de clases para ampliar los comportamientos del agente.

6.4.5 Gestión de *WM + KB* mediante comportamientos

Indicar antes de esta relación de posibles soluciones, que existe una versión de O-DEVICE para *Java* llamada JO-DEVICE[41] que fue estudiada como alternativa para la transformación de las ontologías en OWL sobre el modelo de ejecución del agente [ODEVnnn]. Se trata de una API *Java* para O-DEVICE destinada a integrarse en aplicaciones *Java*. La declaración de su constructor de clase [`public JODEVICE(String config_path)`], crea una primera instancia del motor CLIPS con [`this.clipsPrepare= new Environment()`] y una segunda [`this.clipsExec = new Environment()`] como instancia de motor CLIPS destinada a realizar transformaciones y posteriores incorporaciones a la memoria de producción y la base de conocimientos. Esa API no posee un método para establecer referencias externas de un motor CLIPS, como por ejemplo el método: [`setEngine(Environment)`] y, esa circunstancia fue la causa principal para no reutilizar JO-DEVICE como solución de integración de O-DEVICE en agentes JADE.

Los agentes creados a partir de la propuesta *PsNodeAgent* de la presente investigación, emplean una única instancia de motor de reglas denominada RBEngineCLIPS o RBEngineJESS que implementan la interface RBEngine. Son inseparables por diseño, en el sentido que si una de las dos instancias de objeto muere (*agente* o *motor*), finaliza inmediatamente la otra; característica que puede comprobarse si se teclea [`(exit)`] en alguna de las *Shell Asíncrona* o *Síncrona*. Eso obligó a crear una solución para integrar el sistema O-DEVICE y modelar la secuencia seguida por el flujo de datos transformados, desde las ontologías en OWL hasta el formato de las memorias del motor de reglas del agente, de manera distinta:

[Elaboración propia]

1. Sobre el Editor de comportamientos:
Editar: bhv.acpb.bsh Incorporar al principo del archivo:

```
abbr = "TRUE";
abbrGlobal  = "(defglobal ?*abbr* = " +  abbr + ")";

imports = "FALSE";
importsGlobal  = "(defglobal ?*imports* = " +  imports + ")";

oDeviceFolder  =  nodeEnginePath .replace("\\","/");
oDeviceFolderGlobal   = "(defglobal ?*odevice-folder* = " + "\"" + oDeviceFolder + "\" )";

oTripleFolder  = oDeviceFolder  + "triple-facts/";
oTripleFacts   = oTripleFolder + "$triples.clp";

initLoad = new String[][] {};
switch (myNode.engineSelected) {
      case "CLIPS":
          initLoad = new String[][] {
                 { "MAKE_CLEAR", "null" }
                ,{ "MAKE_BUILD", abbrGlobal}
                ,{ "MAKE_BUILD", importsGlobal}
                ,{ "MAKE_BUILD", oDeviceFolderGlobal}
                ,{ "RUN_ONCE_THEN_BATCH", "(batch* " + relativeEnginePath + "config.bat)"}
                ,{ "RUN_ONCE_THEN_BATCH", "(batch* " + relativeEnginePath + "exec.bat)" }
```

[41] ***JO-DEVICE***: https://ayera.dl.sourceforge.net/project/o-device/JO-DEVICE/jodevice_v2.rar

```
          };
         break;
...
IMPORTANTE: Presionar botón [Save]
```

- Cuando el agente se encuentra en el *nivel-de-ejecución* [n-1], antes de que exista la posibilidad de recibir solicitudes externas, posee el comportamiento de disparo retardado [ACPBEngineLoader] que se activa mediante el proceso [bhv.acpb.bsh]; como se ha visto en Apdo. 6.4.3 anterior. A dicho proceso de activación, se incorporan entonces dos procesos que son propios de O-DEVICE: *config.bat* y *exec.bat*.

- De esa forma, como muestra en el código precedente, siempre que se reinicie en frió o se reinicie con el botón [n-6!], el *agente* queda operativo y su *memoria de trabajo* y su *base de conocimientos*, se cargan con los *hechos* y *reglas* de la última transformación de ontología OWL solicitada al agente.

- El sistema O-DEVICE situó los datos de dichas transformaciones en (/bundle/ y /triple-facts/) por lo que, un *agente* recién creado con su *nodo* aún vacío provocaría un error durante de carga de archivos inexistentes que tiene que ser asumido por el agente.

- Indicar, en relación al *mecanismo de acceso directo al motor de reglas* que se está empleando sobre el motor, que está absolutamente[42] prohibido en *niveles de ejecución* superiores a [n-1], y que quedará reservado sólo a los instantes de inicialización del *agente* o durante labores de desarrollo controlados por el *Actor-Humano* sobre el terminal del agente durante su construcción. El motivo de esa prohibición, se encuentra en que las llamadas a ese método [myEngine .innerCommand (función, cadena)], podría acceder de manera concurrente en el mismo instante que el motor de reglas está ocupando los mismos recursos solicitados. Eso provocaría una violación de memoria y una inmediata ruptura de la máquina *Java* del contendor donde se encuentra el agente.

[Elaboración propia]

```
class AgentPrepareODEVICE2 extends OneShotBehaviour {

 PsNodeAgent  theAgent;
  String[]    ontoURLs;
 SequentialBehaviour multiPhase;

 public AgentPrepareODEVICE2(Agent a, String[] ontos) {
  super(a);
  theAgent = (PsNodeAgent) a;
  ontoURLs = ontos;
 }

 public AgentPrepareODEVICE2(Agent a) {
  this( a, new String[] {"file:///c:/pruebas/reasoners/pizza.owl",
```

[42] ***Modo de acceso directo al motor de reglas***: es aquel utilizado mediante una llamada directa al motor tipo [myEngine .innerCommand(FuncionMotor, CadenaAEvaluar)], que sólo y exclusivamente deben ser ejecutados por el Actor-Humano y en *nivel de ejecución* del agente igual a "1" cuando no están abiertas las comunicaciones para recibir solicitudes externas destinadas al motor del agente. Esta prohibición se basa en que, si el motor de reglas recibe una petición de proceso cuando aún se encuentra en la ejecución de otro, se produce una violación de la memoria de la máquina virtual de Java y una finalización abrupta del agente.

```
    "file:///c:/pruebas/reasoners/food.owl"});
 }

  // ////////////////////////////////////
  // Technique: All ready !
  multiPhase  = new SequentialBehaviour( theAgent );
  multiPhase .setBehaviourName("pphase");
  ACLMessage queryMessage = null;

  // ////////////////////////////////////
  // Phase-1
  queryMessage = theAgent .makeProcessMessage( RBEngine.EVAL_COMMAND , runnintPhase1 );
  multiPhase.addSubBehaviour(new N2NREInitiator(theAgent, queryMessage ));

  // ////////////////////////////////////
  // Phase-2
  for (String arg : ontoURLs){
     String[] j2cfParams = new String[] {
       "-onto" ,   arg
       ,"-out" ,   oTripleFacts
       ,"-abbr",   abbr
       ,"-imports",  imports
   };
    multiPhase.addSubBehaviour(new OneShotWithParams( theAgent, j2cfParams ));
  }

   // ////////////////////////////////////
   // Phase-3
   queryMessage = theAgent .makeProcessMessage( RBEngine.EVAL_COMMAND , runnintPhase3 );
   multiPhase.addSubBehaviour(new N2NREInitiator(theAgent, queryMessage ));


  // ////////////////////////////////////
  // Insert on Agent Agenda:
  theAgent .addBehaviour(multiPhase);
 }

 class OneShotWithParams extends OneShotBehaviour {
  String[] jenaParams;
  public OneShotWithParams(Agent a, String[] params){
   super(a);
   jenaParams = params;
  }
  public void action() {
     J2CF.main(jenaParams);
  }
 }
```

- La solución propuesta es [`AgentPrepareODEVICE2`], (véase código superior), que incorpora el mecanismo de *intervención diferida* sobre el motor del agente, creando una secuencia controlada de acontecimientos y empleando además, el protocolo de interacción de doble secuencia para permitir al motor de reglas, gestionar sus espacios de disponibilidad para atender peticiones y no incurrir en acceso concurrente a los recursos.

- Ese archivo de clase, se localiza en su subdirectorio /config/ para ser cargado e interpretado por el agente en tiempo de ejecución con la función [`source ("nombre-archivo");`] . *véase* [`class.AgentRestartODEVICE2.bsh`] con el *Editor de comportamientos*.

- El mecanismo de *intervención diferida* emplea la implementación del *Protocolo de Interacción* propuesto en la Figura 5.10: Diagrama protocolo de interacción PII2S en [`examples.psnode.core.behaviours.N2NREInitiator`] para crear un comportamiento Iniciador de una conversación de la entidad de software *agente*, consigo misma, de manera que se remite un mensaje de solicitud a su motor de reglas, que es gestionada por el pro-

pio motor de reglas cuyas respuestas son transmitidas al agente solicitante; sin que el motor de reglas advierta ninguna diferencia con cualquiera otra solicitud remitida desde agentes del mismo *Directorio de aplicación.* Por ello, se tiene la plena certeza de que la solicitud emitida al motor de reglas no producirá su bloqueo o una interrupción en la memoria de *Java*, pues las peticiones son gestionadas, encoladas y tramitadas; de una en una.

[Elaboración propia]

Comportamientos utilizados para integrar el Sistema O-DEVICE

Objetivo del proceso archivo en O-DEVICE **Comportamiento**	***Transformar*** prepare.bat	***Incorporar*** exec.bat	***Inicializar*** config.bat
ACPBEngineLoader tipo WakerBehaviour	○	●	●
AgentRestartODEVICE2 tipo OneShotBehaviour	○	●	○
AgentREInitiatorOWL tipo SimpleAchieveREInitiator	○	○	○
AgentREResponderOWL tipo SimpleAchieveREResponder	●	○	○
AgentPrepareODEVICE2 tipo OneShotBehaviour	●	○	●

Tabla 6.3: Comportamientos adicionales en agentes con O-DEVICE integrado

6.4.6 Recarga de las ontologías OWL/RDF en el agente

La transformación en O-DEVICE se lleva a cabo en el proceso [`prepare.bat`] y la solución para integrar dicho proceso en los agentes ya ha sido abordada el apartado anterior. Una vez que el agente concluye la transformación de la ontología solicitada, los directorios `/bundle/` `/triple-facts/` contienen *hechos* que deben ser incorporados al motor de reglas, para inferir nuevo conocimiento sobre los mismos y crear entonces las deducciones y las nuevas *reglas* necesarias para completar el modelo OWL en representación en objetos COOL más la propia representación en *reglas*. Ese proceso lo lleva a cabo O-DEVICE mediante [`exec.bat`].

El agente pasa, en cierto sentido, se actuar como elemento de análisis, transformación y almacenamiento de resultados intermedios; a entidad de software desplegada en la plataforma que posee un motor CLIPS con una imagen de la ontología preparada para recibir consultas. Esa adaptación del agente requiere de al menos tres comportamientos adicionales:

- Dos forman el modelo de conversaciones pautadas tipo *FIPA-Request* para intercambiar las solicitudes de transformación dirigidas al agente o para consultar su base de conocimientos. Esos comportamientos son triviales y se construyen en base a las clases de tipo *Initiator* en los agentes solicitantes: [`Helper04, Helper05, etc.`] y un comportamiento *Respondedor* en cada agente [`ODEVnnn`], a la escucha de peticiones (*véase*, Tabla 6.3).

- El tercer comportamiento es el que permite al agente cambiar su modo de operación, cargar el *modelo* y analizar el mismo para obtener e incorporar nuevo conocimiento y, una vez realizado, quedar operativo a la espera de consultas a su base de conocimientos. Ese comportamiento se crea a partir de la clase [`class.AgentRestartODEVICE2.bsh`]:.

[Elaboración propia]

```
class AgentRestartODEVICE2 extends OneShotBehaviour {
 PsNodeAgent   theAgent;
```

```
 public AgentRestartODEVICE2(Agent a){
  super(a);
  theAgent = (PsNodeAgent) a;
 }

 public void action() {
   String oDeviceFolder   = theAgent .myNode().nodeEnginePath .replace("\\","/");
   String oDeviceFolderGlobal  = "(defglobal ?*odevice-folder* = " + "\"" + oDeviceFolder + "\" )";

   String fileName   = "restartODEVICE.bat";
   String commandToRunning  = "(batch* " + theAgent .myNode().relativeEnginePath + fileName +")";

   String restartProcess   =
    "(clear)"     + System.lineSeparator() +
    oDeviceFolderGlobal  + System.lineSeparator() +
    "(batch* " + theAgent .myNode().relativeEnginePath + "config.bat)" + System.lineSeparator() +
    "(batch* " + theAgent .myNode().relativeEnginePath + "exec.bat)" + System.lineSeparator();

   File oDeviceRestartFile  = new File( theAgent .myNode().nodeEnginePath + fileName );
   BufferedWriter bufferW   = null;
   FileWriter fileW   = null;

... (Proceso de escritura..)

  ACLMessage queryMessage = theAgent.makeProcessMessage(RBEngine.EVAL_COMMAND,commandToRunning );
  N2NREInitiator b   = new N2NREInitiator(theAgent, queryMessage );
  b.setBehaviourName( "oRest");
  theAgent.addBehaviour(b);

}
```

- La solución propuesta es crear un archivo [restartODEVICE.bat], como muestra el código superior de la clase [AgentRestartODEVICE2], que ejecuta el proceso de carga de los *hechos* en formato N-Triple e infiere sobre el mismo nuevo conocimiento. Ese archivo es entonces incorporado al mecanismo de *intervención diferida* sobre el motor del agente siguiendo pasos homólogos a los del proceso de inicialización del agente.

6.4.7 Resumen de la integración de O-DEVICE

Como resumen de la prueba de concepto se incorporan estas conclusiones preliminares:

1. Los *agentes* [ODEVnnn] pueden transformar ontologías de forma autónoma.
2. Los *agentes* [ODEVnnn] pueden atender consultas en formato COOL de CLIPS.
3. Los *agentes* [ODEVnnn] se inicializan tras periodos fuera de servicio.
4. Los *agentes* [ODEVnnn] gestionan el flujo sufrido por los datos del archivo .OWL.
5. Los agentes [HelperNN], facilitan a los *agentes* [ODEVnnn] migrar hacia contenedores.
6. Cada *agente* [ODEVnnn] gestiona una instancia de O-DEVICE y modelo de datos.

[Elaboración propia]

Estado de comportamientos por agente en la aplicación O-DEVICE

Nombre de instancia de objeto comportamiento: **Agente:**	*ODEV101*	*ODEV201*	*Helper04*
jamo JADE-Management-ontology Actions	○	○	○
acpb Agent Capabilities Loader Behaviour	●	○	○
n2nr Node To Node Responder Behaviour	●	●	○
e2nr Engine To Node Responder Behaviour	●	●	○
e2as Engine To Agent Shell Behaviour	○	○	○
nmig Node Migration Behaviour	●	●	●
	● *activado*		

Estado de comportamientos por agente en la aplicación O-DEVICE			
Nombre de instancia de objeto comportamiento: **Agente:**	***ODEV101***	***ODEV201***	***Helper04***
	○ *desactivado*		

Tabla 6.4: Comportamientos activados en agentes con O-DEVICE integrado

6.5 Conclusiones preliminares

- El proceso de transformación de las aplicaciones basadas en *Sistemas de Producción* al paradigma multiagente, debería iniciarse con una comprobación de funcionamiento de la aplicación desde las *Shell-Clips* o *Shell-Jess* dentro del *Directorio de aplicación*. De esa forma, se obtendrá una valiosa información para planificar y diseñar la mejor estrategia de integración en los agentes [`PsNodeAgent`] y [`PsAgent`].

- Cada instancia de agente que contiene una parte o la totalidad de la aplicación original, es una pieza de software singular, en el sentido que poseen comportamientos propios que complementan su funciones básicas aportadas por la clase [`PsNodeAgent`] y por los conocimientos que se van acumulando progresivamente en su *Memoria de Reglas* y los acontecimientos recopilados (*hechos*) en su *Memoria de Trabajo*. Destacando con ello que los comportamientos de respaldo y recuperación del estado de las memorias del motor de reglas, permitirían además, potenciar capacidades de auto-aprendizaje en estos agentes.

- Los agentes de una aplicación construidos a partir de la clase [`PsAgent`] no requieren conexión a ningún motor de reglas, e incluso; no requieren la presencia de dichas librerías. Son agentes destinados a ofrecer soporte a los agentes construidos con [`PsNodeAgent`] para facilitar la migración entre contenedores, para emitir mensajes de solicitudes y, para permitir a los *ingenieros de software* modelar funcionalidades en la aplicación que no requieren de motor sino, que están destinadas a gobernar a los agentes que las componen.

7 Conclusiones

'Las ciudades de León y Ponferrada se encuentran situadas en el centro del cuadrante Noroeste de la península ibérica, que agrupa el occidente de la región de Castilla y León junto a las regiones de Cantabria, Asturias, Galicia y Norte de Portugal, ribereñas del océano Atlántico.'

Situación geográfica

Universidad de León
(http://www.unileon.es/universidad/localizacion-e-historia/como-llegar)

7.1 Conclusiones finales

La investigación ha concluido con un conjunto de resultados que permiten cumplir el objetivo principal de la misma: desplegar aplicaciones basadas en *Sistemas de Producción* sobre una plataforma multiagente manteniendo intactas las propiedades intrínsecas de los agentes que la componen e incorporando a estas aplicaciones las funcionalidades propias de los sistemas distribuidos.

Los resultados y conclusiones finales se recogen en los siguientes epígrafes:

(**1º**) Se ha diseñado un modelo de integración de un motor de inferencia con un agente software utilizando una estructura desacoplada implementada como interface. Esta característica ha permitido que el modelo de integración sea independiente de la tecnología del motor de inferencia seleccionado.

(**2º**) Se ha diseñado un modelo de coordinación entre agente y motor de inferencia mediante un protocolo de interacción conforme a las especificaciones FIPA. Este modelo permite a un agente comunicarse con su propio motor ó con un motor remoto asociado a otro agente de la misma aplicación.

(**3º**) Se ha diseñado un agente con funcionalidad de entorno de desarrollo que tiene la capacidad de extender a cualquier otro agente de la aplicación con el fin de facilitar el diseño, análisis y depuración de sus comportamientos, así como modelar el conocimiento mediante diferentes shells que permiten interactuar con el código del agente y con el motor de inferencia tanto en tiempo de diseño como en tiempo de ejecución.

(**4º**) Se ha diseñado una ampliación de las capacidades del agente para interpretar, en tiempo de ejecución, comportamientos externalizados en archivos.

(**5º**) Se ha diseñado un mecanismo para conseguir la migración, de estos agentes complejos con motor de inferencia integrado, entre contenedores de la plataforma multiagente; incluidos los conocimientos acumulados y la configuración íntegra del agente.

(**6º**) Se han realizado una serie de pruebas de concepto mediante las que se ha comprobado el adecuado funcionamiento de los elementos diseñados, por lo que se puede afirmar que el conjunto de aportaciones realizadas en esta investigación ha logrado conseguir el objetivo principal de la misma.

7.2 Relación de futuros trabajos de investigación

Los futuros proyectos y trabajos de investigación se clasifican atendiendo a (**1**) proyectos destinados a realizar *trabajos de mejora*, para optimizar las técnicas propuestas, y (**2**) proyectos destinados realizar la *explotación* de las técnicas implementadas.

***Proyectos de mejora** de las técnicas e instrumentos aportados:*

a. Finalización del agente [`PsBoardAgent`] de repositorio de *Tickets*, que permitirá implementar la resolución de problemas colaborativa utilizando este repositorio común de soluciones parciales del problema.
b. Creación de [`RBEnginePROLOG`] con `SWI-Prolog` e integrar en agentes.
c. Creación de [`RBEngineDROOLS`] e integrar en agentes [`PsNodeAgent`].
d. Desarrollo de *agente de soporte* para supervisar aplicaciones compuestas por un número elevado de *agentes*.

***Proyectos de explotación** de las técnicas e instrumentos aportados:*

e. Modelos de sincronización entre agentes: basados en la investigación de Lamport [Lamport, 1978] y los trabajos de [Ben-Zvi & Moses, 2010].
f. Análisis de la degradación del rendimiento en *agentes* en situaciones de sobrecarga de solicitudes: solución mediante comportamiento de *auto-gobierno*.
g. Aplicaciones educativas del sistema construido.
h. Explotación de aplicaciones basadas en sistemas de producción distribuidos para dominios concretos como el ámbito de la ciberseguridad ó el análisis de datos en redes sociales de Internet.

BIBLIOGRAFÍA

[Arnold et al., 1996] Arnold, K., Gosling, J., & Holmes, D. (1996). *The Java programming language*, volume 2. Addison-Wesley Reading.

[Badica et al., 2011] Badica, C., Braubach, L., & Paschke, A. (2011). Rule-based distributed and agent systems. In *Rule-Based Reasoning, Programming, and Applications* (pp. 3–28).: Springer.

[Barker et al., 1989] Barker, V. E., O'Connor, D. E., Bachant, J., & Soloway, E. (1989). Expert systems for configuration at digital: Xcon and beyond. *Communications of the ACM*, 32(3), 298–318.

[Bellifemine et al., 2010] Bellifemine, F., Caire, G., Trucco, T., & Rimassa, G. (2010). *JADE Programmer's Guide*, volume 1. CSELT S.p.a. and Tilab S.p.a.

[Bellifemine et al., 2001] Bellifemine, F., Poggi, A., & Rimassa, G. (2001). Developing multi-agent systems with a fipa-compliant agent framework. *SOFTWARE-PRACTICE AND EXPERIENCE*, 1(31), 103–128.

[Bellifemine et al., 2007] Bellifemine, F. L., Caire, G., & Greenwood, D. (2007). *Developing multi-agent systems with JADE*, volume 7. John Wiley & Sons.

[Ben-Zvi & Moses, 2010]Ben-Zvi, I. & Moses, Y. (2010). Beyond lamport's happened-before: On the role of time bounds in synchronous systems. In *Distributed Computing* (pp. 421–436). Springer.

[Beneventi et al., 2004] Beneventi, A., Poggi, A., Tomaiuolo, M., & Turci, P. (2004). Integrating rule and agent-based programming to realize complex systems. *WSEAS Trans. on Information Science and Applications*, 1(1), 422–427.

[Benito, 2005] Benito, M. (2005). Sistemas de clasificación. manual de aprendizaje de la clasificación decimal universal y breve introducción a la clasificación decimal de dewey. Available in: http://www.taranco.eu/cdu/cdu-breve.htm.

[Bigus & Bigus, 2001] Bigus, J. P. & Bigus, J. (2001). *Constructing intelligent agents using Java*. Wiley New York.

[Buchanan & Feigenbaum, 1978] Buchanan, B. G. & Feigenbaum, E. A. (1978). Dendral and meta-dendral: Their applications dimension. *Artificial intelligence*, 11(1), 5–24.

[Caire & Cabanillas, 2010] Caire, G. & Cabanillas, D. (2010). *JADE Tutorial Application-Defined Content Languages and Ontologies*, volume 5-April-2010 - JADE 4.0. Telecom Italia S.p.A.

[Cancedda & Caire, 2010] Cancedda, P. & Caire, G. (2010). *JADE Tutorial Creating Ontologies by means of the Bean-Ontology Class*, volume 15-April-2010 - JADE 4.0. Telecom Italia S.p.A.

[Cardoso, 2007] Cardoso, H. L. (2007). Integrating jade and jess. available in: http://jade.tilab.com/ documentation/tutorials-guides/integrating-jade-and-jess/.

[Cohen & Feigenbaum, 2014] Cohen, P. R. & Feigenbaum, E. A. (2014). *The handbook of artificial intelligence*, volume 3. Butterworth-Heinemann.

[Corchado & Molina, 2002] Corchado, J. M. & Molina, J. M. (2002). *Introducción a la teoría de agentes y sistemas multiagente*. Catedral Publicaciones. Salamanca.

[Corkill, 1991] Corkill, D. D. (1991). Blackboard systems. *AI expert*, 6(9), 40–47.

[Corkill, 2003] Corkill, D. D. (2003). Collaborating software: Blackboard and multi-agent systems & the future. In *Proceedings of the International Lisp Conference* New York, NY.

[CSELT & TILab, 2014] CSELT, S. & TILab, S. (2014). Jade - java agent development framework. is a framework to develop multi-agent systems in compliance with the fipa specifications. jade successfully passed the 1st fipa interoperability test in seoul (jan. 99) and the 2nd fipa interoperability test in london (apr. 01). Available in: http://jade.tilab.com/ downloaded in Open Source, under LGPL restrictions.

[CSELT & TILab, 2017] CSELT, S. & TILab, S. (2017). Jade - java agent development framework. is a framework to develop multi-agent systems in compliance with the fipa specifications. jade 4.5.0 - revision 6825 of 23-05-2017 10:06:04. Available in: http://jade.tilab.com/ downloaded in Open Source, under LGPL restrictions.

[Durfee, 2001] Durfee, E. H. (2001). Distributed problem solving and planning. In *Multi-agent systems and applications* (pp. 118–149). Springer.

[Durfee et al., 1995] Durfee, E. H., Lesser, V. R., & Corkill, D. D. (1995). Trends in cooperative distributed problem solving. *Knowledge and Data Engineering, IEEE Transactions on*, 1(1), 63–83.

[Durfee & Rosenschein, 1994] Durfee, E. H. & Rosenschein, J. S. (1994). Distributed problem solving and multi-agent systems: Comparisons and examples. *Ann Arbor*, 1001(48109), 29.

[Eriksson, 2003] Eriksson, H. (2003). Using jesstab to integrate protégé and jess. *IEEE Intelligent Systems*, 18(2), 43–50.

[Erman et al., 1980] Erman, L. D., Hayes-Roth, F., Lesser, V. R., & Reddy, D. R. (1980). The hearsay-ii speech-understanding system: Integrating knowledge to resolve uncertainty. *ACM Computing Surveys (CSUR)*, 12(2), 213–253.

[Escoí et al., 2013] Escoí, M., Villaplana, F. D. A., Minguet, E. E., Saiz, A. G., Fornes, P. G., Marín, A. D. J., Roig, R. S., & Salvador, J. (2013). *Concurrencia y sistemas distribuidos*. Editorial de la Universidad Politécnica de Valencia.

[Felleisen & Friedman, 2000] Felleisen, M. & Friedman, D. P. (2000). *A Little Java, A Few Patterns*. The MIT Press Cambridge, Massachusetts London, England.

[Forgy, 1981] Forgy, C. L. (1981). Ops5 user's manual. In *Computer Science*, volume 81 (pp. 135).: Carnegie-Mellon University.

[Forgy, 1982] Forgy, C. L. (1982). Rete: A fast algorithm for the many pattern/many object pattern match problem. In *Artificial Intelligence*, volume 19(1) (pp. 117–37).

[Friedman-Hill, 2003] Friedman-Hill, E. (2003). *JESS in Action*. Manning Greenwich, CT.

[Friedman-Hill, 2016] Friedman-Hill, E. (2016). Jess java expert system software (8.0a1 (alfa)). jess the rule engine for the javatm platform web site. Available in: http://herzberg.ca.sandia.gov/.

[Garg, 2005] Garg, V. K. (2005). *Concurrent and distributed computing in Java*. John Wiley & Sons.

[Garijo, 2005] Garijo, F. (2005). Tecnologá de agentes: orgenes, conceptos y caractersticas. Ín *Agentes software y sistemas multiagente: conceptos, arquitecturas y aplicaciones* (pp. 1–28).: Prentice Hall.

[Gasser & Ishida, 1991]Gasser, L. & Ishida, T. (1991). A dynamic organizational architecture for adaptive problem solving. In *AAAI*, volume 91 (pp. 185–190).

[Giarratano, 1997] Giarratano, J. C. (1997). Clips user's guide, software technology branch, nasa/lyndon b. *Johnson Space Center, USA*, (pp. 159).

[Giarratano, 2014] Giarratano, J. C. P. (2014). *CLIPS User's Guide. Version 6.30*. CLIPS.

[Giovanni et al., 2003] Giovanni, C., Wim, C., Francisco, G., Jorge, G., Juan, P., Francisco, L., Paulo, C., Paul, K., Jamie, S., Richard, E., & Philippe, M. (2003). An agent oriented analysis using message uml. In G. Goos, J. Hartmanis, & J. van Leeuwen (Eds.), *Proceedings of the Second International Workshop (AOSE-2002). Lecture Notes in Computer Science. Springer-Verlag 2222*.

[Greenwood, 2004] Greenwood, D. (2004). Fipa the foundation for intelligent, physical agents.

[Griss et al., 2003] Griss, M. L., Fonseca, S., Cowan, D., & Kessler, R. (2003). Using uml state machine models for more precise and flexible jade agent behaviors. In *Agent-Oriented Software Engineering III* (pp. 113–125). Springer.

[Gruber, 1995] Gruber, T. R. (1995). Toward principles for the design of ontologies used for knowledge sharing? *International journal of human-computer studies*, 43(5-6), 907–928.

[Hayes-Roth, 1985] Hayes-Roth, B. (1985). A blackboard architecture for control. *Artificial intelligence*, 26(3), 251–321.

[Helin, 2003] Helin, H. (2003). *Supporting nomadic agent-based applications in the FIPA agent architecture*. University of Helsinki.

[Hermann et al., 2008] Hermann, C., Jung, B., & Petta, P. (2008). Designing criteria-driven scheduling as integrated service for ieee-fipa compliant multi-agent infrastructures. In *Multikonferenz Wirtschaftsinformatik*: Citeseer.

[Hoffman et al., 2001] Hoffman, O., Bellifemine, F., & Friedman-Hill, E. (2001). Software: Jadejessprotege, package example for closer integration of jade with jess, optionally also with protege. Available in: http://jade.tilab.com/documentation/examples/jadejessprotege.

[Ishida, 1994] Ishida, T. (1994). *Parallel, distributed and multiagent production systems*. Springer-Verlag Berlin.

[Ishida, 1995] Ishida, T. (1995). Parallel, distributed and multi-agent production systems-a research foundation for distributed artificial intelligence. In *ICMAS* (pp. 416–422).

[Kamyab & Bellifemine, 2004] Kamyab, K. & Bellifemine, F. (2004). Basic jess behaviour class. examples.jess (jade). imperial college of london (upgrade to jess 5.1).

[Kersten & Kak, 1993] Kersten, P. & Kak, A. (1993). A tutorial on lisp object-oriented programming for blackboard computation (solving the radar tracking problem). *International journal of intelligent systems*, 8(5), 617–669.

[Lamport, 1978] Lamport, L. (1978). Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7), 558–565.

[Larman, 2002] Larman, C. (2002). *UML y Patrones. Una introducción al análisis y dise'o orientado a objetos y al proceso unificado. 2 Ed.* Prentice Hall - Pearson Educacion S.A.

[Lea, 1999] Lea, D. (1999). *Concurrent Programming in Java: Design Principles and Patterns, Second Edition*. Addison-Wesley.

[Lenat & Feigenbaum, 1991] Lenat, D. B. & Feigenbaum, E. A. (1991). On the thresholds of knowledge. *Artificial intelligence*, 47(1), 185–250.

[Lesser, 1989] Lesser, V. R. (1989). Coordination in distributed problem-solving networks. *ACM Computing Surveys (CSUR)*, N00014-89J(1877), 1–17.

[Luger & Chakrabarti, 2011] Luger, G. & Chakrabarti, C. (2011). Knowledge-based probabilistic reasoning from expert systems to graphical models: Report.

[Mas, 2005] Mas, A. (2005). *Agentes Software y Sistemas Multiagente. Conceptos, Arquitecturas y Aplicaciones*. Prentice Hall.

[Meditskos & Bassiliades, 2006]Meditskos, G. & Bassiliades, N. (2006). O-device: An object-oriented knowledge base system for owl ontologies. In *Hellenic Conference on Artificial Intelligence* (pp. 256–266).: Springer.

[Meditskos & Bassiliades, 2008]Meditskos, G. & Bassiliades, N. (2008). A rule-based object-oriented owl reasoner. *Knowledge and Data Engineering, IEEE Transactions on*, 20(3), 397–410.

[Meditskos & Bassiliades, 2011]Meditskos, G. & Bassiliades, N. (2011). Clips–owl: A framework for providing object-oriented extensional ontology queries in a production rule engine. *Data & Knowledge Engineering*, 70(7), 661–681.

[Merritt, 2012] Merritt, D. (2012). *Building expert systems in Prolog*. Springer Science & Business Media.

[Niemeyer, 2008] Niemeyer, P. (2008). Beanshell: Lightweight scripting for java. 2008. *Homepage: http://www. beanshell. org*.

[Niemeyer, 2014] Niemeyer, P. (2014). Beanshell website.

[Peers, 1998] Peers, S. (1998). A blackboard system approach to electromagnetic sensor data interpretation. *Expert Systems*, 15(3), 197–215.

[Perraju & Prasad, 2000] Perraju, T. S. & Prasad, B. E. (2000). An algorithm for maintaining working memory consistency in multiple rule firing systems. *Data & Knowledge Engineering*, 32(2), 181 – 198.

[Pilone & Pitman, 2005] Pilone, D. & Pitman, N. (2005). *UML 2.0 in a nutshell. A Desktop Quick Reference*. O'Reilly.

[Poggi & Tomaiuolo, 2011] Poggi, A. & Tomaiuolo, M. (2011). Rule engines and agent-based systems. *Machine Learning: Concepts, Methodologies, Tools and Applications*, (pp. 211).

[Poggi et al., 2004] Poggi, A., Tomaiuolo, M., & Turci, P. (2004). Extending jade for agent grid applications. In *Enabling Technologies: Infrastructure for Collaborative Enterprises, 2004. WET ICE 2004. 13th IEEE International Workshops on* (pp. 352–357).: IEEE.

[Poggi et al., 2007] Poggi, A., Tomaiuolo, M., & Turci, P. (2007). An agent-based service oriented architecture. In *WOA* (pp. 157–165).

[Protégé Community & Stanford University, 2014] Protégé Community, D. T. & Stanford University, S. o. M. (2014). *Protégé,A free, open-source ontology editor and framework for building intelligent systems*. Stanford Center for Biomedical Informatics Research (BMIR), Stanford University. Stanford, California 94305.

[Rahimi et al., 2007] Rahimi, S., Nagaraja, G., Gandy, L., & Gupta, B. (2007). An expert system for analysis of consistency criteria in checkpointing algorithms. *Scalable Computing: Practice and Experience*, 8(2), 197–208.

[Riley, 2008] Riley, G. (2008). *CLIPS Reference Manual Volume II Advanced Programming Guide*. Beta, Quicksilver.

[Riley, 2016a] Riley, G. (2016a). Clips feature test 630 files. Available in: https://sourceforge.net/projects/clipsrules/files/CLIPS/6.30/feature_tests_630.zip/download.

[Riley, 2016b] Riley, G. (2016b). Clips rule based programming language expert system tool clips (6.31) and clipsjni (0.5), clips rule based programming language web site. Available in: https://sourceforge.net/projects/clipsrules/.

[Rimassa, 2003] Rimassa, G. (2003). *Runtime Support for Distributed Multi-Agent Systems*. PhD thesis, Universita degli Studi di Parma, Departimento di Ingegneria Dell'Informazione.

[Rubin et al., 1988] Rubin, K. S., Jones, P. M., & Mitchell, C. M. (1988). Ofmspert: Inference of operator intentions in supervisory control using a blackboard architecture. *Systems, Man and Cybernetics, IEEE Transactions on*, 18(4), 618–637.

[Rumbaugh et al., 2007] Rumbaugh, J., Booch, G., & Jacobson, I. (2007). *El Lenguaje unificado de modelado: manual de referencia*. Addison-Wesley object technology series. Pearson Educación.

[Russell & Norvig, 2010] Russell, S. & Norvig, P. (2010). *Artificial intelligence: a modern approach, 3rd Edition*. Prentice-Hall, Inc., 3rd edition.

[Shoham, 1993] Shoham, Y. (1993). Agent-oriented programming. *Artificial intelligence*, 60(1), 51–92.

[Wooldridge, 2002] Wooldridge, M. (2002). *An Introduction to Multi-Agent Systems*. John Wiley & Sons Ltd.

Especificaciones FIPA:

[FIPA00001] FIPA Abstract Architecture Specification. Foundation for Intelligent Physical Agents, 2002. http://www.fipa.org/specs/fipa00001/

[FIPA00008] FIPA Agent Communication Language Specification. Foundation for Intelligent Physical Agents, 2000. http://www.fipa.org/specs/fipa00008/

[FIPA00018] FIPA SL Content Language Specification. Foundation for Intelligent Physical Agents, 2000. http://www.fipa.org/specs/fipa00008/

[FIPA00023] FIPA Agent Management Specification. Foundation for Intelligent Physical Agents, 2000 http://www.fipa.org/specs/fipa00023/

[FIPA00025] FIPA Interaction Protocol Library Specification. Foundation for Intelligent Physical Agents, 2000. http://www.fipa.org/specs/fipa00025/

[FIPA00035] FIPA Subscribe Interaction Protocol Specification. Foundation for Intelligent Physical Agents, 2000. http://www.fipa.org/specs/fipa00035/

[FIPA00037] FIPA Communicative Act Library Specification. Foundation for Intelligent Physical Agents, 2000. http://www.fipa.org/specs/fipa00037/

[FIPA00061] FIPA ACL Message Structure Specification. Foundation for Intelligent Physical Agents, 2000. http://www.fipa.org/specs/fipa00061/

[FIPA00067] FIPA Agent Message Transport Service Specification. Foundation for Intelligent Physical Agents, 2000. http://www.fipa.org/specs/fipa00067/

[FIPA00070] FIPA ACL Message Representation in String. Foundation for Intelligent Physical Agents, 2000. http://www.fipa.org/specs/fipa00070/

[FIPA00075] FIPA Agent Message Transport Protocol for IIOP Specification. Foundation for Intelligent Physical Agents, 2000. http://www.fipa.org/specs/fipa00075/

ÍNDICE ALFABÉTICO

A

ACC, 42, 43, 48, *Véase* Agent Communication Chanel
acción *Advertise*, 79
acción *Process*, 80
acciones, 77
acciones Process, 91
acciones *Repl*, 79, 91
Acklin B. V., 84
ACL, 32, 35, 37, 38, 42, *Véase* Agent Communication Language
ACLGui, 84
`acl-representation`, 43
ACLTrace, 84
activaciones, 54
Actores, 21
Actor-humano, 78, 80, 87
Actos comunicativos, 42, 80
Adquisición del conocimiento, 18
Advertise, 79
Agenda, 17, 20
Agent Communication Chanel, 42
Agent Communication Language, 35, 37
Agent Unified Modeling Language, 43
Agent::Problem-Solvers Node, 67
Agent::Problem-Solvers-Node, 68
Agente, 23, 33, 36
  Definición de Bellifemine F, 33
  Definición de FIPA, *Véase Visión de Greenwood D*, 37
Agente AMS, 46
Agente-AMS, 51
Agente-DF, 51
Agente-nodo, 103
Agentes fundamentales en JADE, 44
Agentes-DF, 45
Agentes-nodo, 100
Agentes-software inteligentes, 32
Agent-wise, 55
Agent-wise Blocking with timeout, 55
AGREE, 89
Aláiz, Héctor, i
Algoritmos
  Búsqueda del camino óptimo, 15
  Cómputo de coste o máximo beneficio, 16
  De búsqueda, 16
  Espacio de estados finitos, 15
  Rendimiento, 16
Almacenamiento intermedio, Véase Polling
Amherst, 1
AMS, *Véase* Sistema de Gestión de Agentes
AMSSubscriber, 58
AP, *Véase* Plataforma de Agentes
API de BeanShell, 75
API de JADE, 45, 83
API de Jess, 75
Aplicación, 91
aplicación Sudokus, 121
Apropiativo, *Véase* Preemptivo
Archivo de funciones de utilidad, 97
archivos de sintaxis, 83
Arquitectura
  de Pizarra, 26
Arquitectura abstracta, 32
Aspecto de la documentación, 93
Atributo-valor, *Véase* Representación de datos
AUML, 43, *Véase* Agent Unified Modeling Language
ayuda *en-línea*, *Véase* help()

B

Badica, Costin, 29
Bancos de conocimientos, 27
barrera de protección natural, 80
BeanShell, 70, 83
Behaviours, 33
Bellifemine, Fabio, 31, 32, 46, 48, 58, 60
Beneventi, Alessandro, 33
Bigus, Joseph P y Bigus, Jennifer, 15
bitácora, 104
Blackboard Systems, *Véase* Sistemas de Pizarra

Bloqueo del agente con límite de tiempo, Véase Agent-wise Blocking with timeout
Bloqueo en cada ciclo de un SE, 17
board, 73
Braubach, Lars, 29

C

Cabanillas, David, 76
Caire, Giovanni, 46, 76
Cancedda, Paolo, 76
capa de propiedades 'N, 104
capacidad de migración, 67
Cardoso, Herique L, 66, 68
Caso de uso particular, 89
catálogo cerrado, 81
catálogo de *Tickets*, 92
Ciclo de ejecución, 18
Ciclo de vida, 34
cierre de cada intérprete, 74
CL, *Véase* Content Languages
clasificación DDC, 77
CLASSPATH, 69, 70
CLIPS, 16, 18, 64, 70, 86
CLIPSJNI, 75
Codificación de datos estructurados, 39
Códigos de procedimiento, 81
Coherencia, Véase Etapa de diseño
cola de mensajes, 84
Cola de mensajes, 55
combinación de teclas, 85
`comments`, 43
Competencia, Véase Etapa de diseño
comportamiento básico, 58
comportamientos compuestos, 59
comportamientos en *paralelo*, 59
comprobaciones semánticas, 76
Computación en GRID, 30
comunicación entre *agentes*, 76
Comunicación entre el *agente* y..., 66
concepto *Ticket*, 78
conceptos, 77
Conceptos del Dominio, 15
Configuración centralizada de librerías, 118
Confinar procesos, 51
Constructor de Agentes-nodo, 71
Constructor de Directorio de aplicación, 67
contenedores, 46
contenido de un mensaje, 76
Contenido del mensaje, 42
content, 39, 40, 79
Content Languages, 38
Contract-Net (CN), 22
Control Component, Véase Mecanismo de control
Control de la conversación, 42
Convenios, 102
conversación, 57, 88
conversation-id, 41
Conversation-wise, 55
Conversation-wise Blocking with timeout, 55
coordinación de *hechos*, 81
coordinar memorias, 80
CORBA, 48
CORBA IIOP, 48
Corchado, Juan Manuel, 2
Corkill, Daniel, 27
CSELT & TILab, 31
Cuerpo de estándares, *Véase* FIPA
CyclicBehaviour, 58

D

DARPA, *Véase* Hearsay-II
Data-Driven Reasoning, *Véase* Sistemas Expertos de razonamiento dirigido por datos
`date`, 43
desarrollador de aplicaciones, 69
Descomponer, *Véase* fases
Descripción del contenido, 42
Desempeño eficiente, 23
Despachar nuevas consultas, 91
`dewey`, 78
Dewey, 77
Dewey, Melvil, 77
DF, *Véase* Servicio de Directorio
Directorio de aplicación, 67, 68, 100
  config/, 83, 86
  Constructor de, 70
  lib/, 69
  nodes/, 96, 102
  nodes/Board/, 106
  nodes/node-Name/engine, 86
  raíz del directorio /, 71
  var/, 74, 93, 98
  var/data, 94
  var/doc/dps/, 94
  var/doc/index.html, 75
directorio de librerías nativas, 101
disparo de reglas, 91
Distribución de componentes, 103
DNS, 72
Documentación, 93
Dominio de conocimiento, 12, 78
dominio del discurso, 76
Dominio del problema, 12
DPS-Framework, 67, 68

DPS-Framework-Builder, 67, 68
`dps-frameworkBuilder-0.9.1.jar`, 70, 100
dummy, 73
Durfee, Edmund H., 2, 11, 21, 22, 24

E

E2NR, 90
Editor de archivo, 86
Editor de comportamientos, 84
Editor de mensajes, 84
Efecto Racional, *Véase* Respuesta Racional
Ejecución a impulsos, 91
Ejecución *basadas en tareas*, 54
Ejecución de reglas, 18
emular una *Shell*, 87
Encadenamiento hacia delante, *Véase* Sistemas de Producción
encoding, 39, 40
Engine to Node Responder, *Véase* E2NR
entidad AMS, 34
Entidad AMS, 34
Entidades abstractas de un mensajes ACL, 39
Entorno acotado, 24
Entorno de ejecución, 101
Entorno de pruebas, 80, 88
Entorno Tecnológico, 70
Envelope, 39
Erman, Lee D, 1
Escenario de colaboración, *Véase* fases
Escenario para replicar ensayos, 101
Escoí, Muñoz, 56
Espacio de estados, *Véase* Algoritmos'
estados de bloqueo, 69
estrategia, *Véase* Algoritmo
Estructura de mensajes ACL, 32
Etapa de diseño, 24
eventos, 54
Execute, 86
Experto del dominio, 13

F

Factor crítico de éxito, 107
FAILURE, 91
Fase de disparo, 17
fases, 78
Feigenbaum, Edward A, 11, 12
fiabilidad, 80
Finite State Machine, 59
FIPA, 2, 32, 34, 79
FIPA Misión, 63
FIPA00001, *Véase* Arquitectura abstracta
FIPA000023, 35
FIPA00008, 37, 38, 45, *Véase* Lenguaje de comunicación ACL
FIPA00018, *Véase* Lenguaje de contenido
FIPA00023, 33, 45, *Véase* Gestión de agentes en la plataforma
FIPA00025, 38, 43, 88, *Véase* Librería de protocolos de interacción
FIPA00035, 52, *Véase* Protocolo de interacción de suscripciones
FIPA00037, 38, *Véase* Librería de actos de comunicación
FIPA00061, 38, *Véase* Estructura de mensajes ACL
FIPA00067, 36, 41
FIPA00070, *Véase* Representación de mensajes mediante cadenas de texto
FIPA-Request, 88
formato de *cadenas FIPA*, 76
Forward chaining, *Véase* Encadenamiento hacia delante
Fotogramas, *Véase* Algoritmos, Espacio de estados finitos
Frame y Ontology, Véase FIPA00061
Frames, *Véase* Representación de datos
Framework, *Véase* DPS-Framework
`framework.properties`, 91
Friedman-Hill, Ernest, 11, 18
`from`, 43
`from 0`, 78
FSMBehaviour, 59
Fuentes de conocimiento, 26, *Véase* Bancos de conocimientos

G

GADT, *Véase* Global-Agent Descriptor, *Véase* Global-Agent Descriptor
García, Isaías, i, 15
Garijo, Francisco, 1, 25
Gasser, Les, 11
Gestión de agentes en la plataforma, 32
gestor de eventos, 87
`getThread`, 61
Giarratano, Joseph C, 11, 12
Global-Agent Descriptor Table, 49
Goal-Driven Reasoning, *Véase*
Griss, Martin, 69
Grupo de investigación Amherst, Univ. de Massachusets, 1
grupo de *Threads*, 54

H

HashMap, 66

Hayes-Roth, Frederick, 1
Hearsay-II, 1
Helin, Heikki, 68, 99
`help()`, 83
Hermann, Christoph, 2
herramienta de construcción, 46
Hilo de ejecución, 33, 53, 56
histórico de comandos tecleados, 83
Hoffman, Oliver, 66
HSQLDB, 70
HyperSQL, 94

I

Índices de prioridad, *Véase* Resolución de conflictos
Inducción, proceso de, 18
INFORM, 91
Ingeniería de software, 33
Ingeniero de conocimiento, 8, 13
In-Process, 52
in-reply, 41
instanciar nuevos objetos, 83
Instancias de Agentes-nodo, 67
Instrumentos, 75
Integración de un intérprete de Java, 94
Integración de un motor de inferencia, 66
Inteligencia
  Inteligencia Artificial (AI), 1
  Inteligencia Artificial Distribuida (DAI), 2, 22, 23
Inteligencia Artificial Distribuida, 11
`intended-receiver`, 43
interconexión extremo-a-extremo, 37
Interface DBEngine.java, 98
interface de programación avanzada, *Véase* API
Interface `RBEngine`, 81
interfaz gráfica, 82
Interfaz Grafica de Usuario, 45
Interoperabilidad, 32
Interoperable, 39
Interpretación de Envlope, 43
interrogar al repositorio, 92
IPL, 43, *Véase* Librería de Protocolos de Interacción
Ishida, Toru, 11, 16, 19

J

JADE, 57, 64, 91
JADE User Agents, 54
Java Agent Development Environment, *Véase* JADE
Java RMI, 46
`javax.swing.JTextArea`, 87
Jess, 65, 70, 86

K

KBS, *Véase* Sistemas basados en conocimiento
Knowledge sources, Véase • Fuentes de conocimiento, Véase • Fuentes de conocimiento

L

LADT, *Véase* Local-Agent Descriptor Table
language, 40
Larman, Craig, 67
Launcher, 73
Lea, Doug, 53
LEAP, 61
Left Hand Side, 17
Lenguaje de comunicación de agentes ACL, 32
Lenguaje de Contenido, 38
Lenguaje del contenido, 32
Lesser, Victor R, 1
`level.00.bsh`, 83, 97
Librería de actos de comunicación, 32
Librería de protocolos de interacción, 32
Librería de *Protocolos de Interacción*, 43, 88
librerías centralizadas, 69
*Línea de trabajo de **propuestas***, 66
*Línea de trabajo de **técnicas***, 66
*Línea de trabajo de **utilidades***, 65
*Línea de trabajo **instrumental***, 65
Líneas de trabajo, 64
lista de confianza, 91, 101
LoaderBehaviour, 60
Local-Agent Descriptor Table, 49

M

Main-container, 48
Marcos de Actores, *Véase* Sistemas de Pizarra
MAS Platform, 67
Mas, Ana, 2, 16
MAS-Platform, 69
Mayúsculas+*Ctrl+Enter*, 85
Mecanismos de control, 26
Mensaje ACL, 38
Merrit, Dennis, 11, 12
message, 39
Messages Transfer System, 34
metadatos, 101
metáfora de los expertos, 21

`method`, 81
método yield, Véase Hilo de ejecución
Middleware, 31, 38
Migración de agentes-nodo, 101
Modelo de ejecución, 33
Modelo de Gestión de Eventos, 48
Modelo de referencia, 35
Modelo de *solicitud y respuesta*, 48
Modelo del dominio, 66
Modelo social, 23
Módulo de Notificaciones, 79
motor local, 85
Motores del sistema experto a integrar, 67
MsgReceiver, 58
MTP, 42, 44
MTS, 34, 42, *Véase* Servicio de Transporte de Mensajes
Multitarea *cooperativa*, 33, 57
`myAgent`, 83
`myBOB`, 83
`myNode`, 83

N

N2NI, 89
N2NR, 89
N2TbI, 92
NASA, 17
Niemeyer, Pat, 70
Nivel-0, 97
Nivel-1, 97
Nivel-3, 97
Nivel-5, 97
Nivel-6, 97
Node to Node Initiator, *Véase* N2NI
Node to Node Responder, *Véase* N2NR
Node to Ticket-Board Initiator, *Véase* N2TBI
`node.methods.bsh`, *Véase* Archivo de funciones de utilidad
Node-Layer, 104
nodes, 73
Nodos de resolución, 22
nombre de la aplicación, 101
notificaciones, 54
Nueva metáfora, 22

O

objeto *Composite*, 59
Objetos activos de *Java*, 33, 54
Objetos *behaviours*, *Véase* Behaviours
OCLC, 77
OneShotBehaviour, 58
Ontología, 37
ontología *fipa-acl*, 39, 76
Ontología *Problem-Solvers*, 75
ontology, 41
OPS5, 17, 19, 20
óptimo, *Véase* Algoritmo
órdenes, 54
Orientada a la tarea, 54
Orientada al Actor, 54

P

P2P, 48
Pantalla
  Editor de archivos, 82
  Editor de comportamientos, 82
  Editor de mensajes, 82
  Shell Asíncrona, 82
  Shell JADE, 82
  Shell Síncrona, 82
Parámetros adicionales, 42
Parámetros del *envelope*, 42
Parámetros obligatorios, 42
Participantes, 42
pasarela CLIPSJNI, 70
Paschke, Adrian, 29
payload, 39
`payload-encoding`, 43
`payload-length`, 43
Pérdida de la capacidad de colaboración, 28
performativa, 40
performative, 41
Pestaña
  Gestión del agente, 82
  Gestión del motor de inferencia, 82
`phase`, 78
Phase, 78
PII2S, 85
Pizarra, 26
Pizarra, Arquitectura de.., 1, 11
planificación progresiva, *Véase* Multitarea cooperativa
Plataforma de Agentes, 37
Plataforma distribuida, 47
Plataforma multiagente, 67
platform, 73
pocket, 73
Poggi, Agostino, 46, 66
Poggi, Giovanni, 68
política de recepción, 55
políticas de bloqueo, 55
Polling, 55
portabilidad, 81
posts, 54
predicado *ProblemFor*, 78
predicados, 77

Preemptivo, 33
Primer boceto, 88
Prioridad de ejecución, 20
ProblemFor, 78
Problem-Solvers, 76
Procesamiento simbólico, *Véase* Prolog
proceso de catálogo, 79
Proceso de migración, 66
Procesos de inferencia prolongados, 66
`process`, 78
Process, 80
Productos aportados, 65
programador de tareas, 57, 58
programar en Lenguaje *Java*, 83
Programming in Large, *Véase*
Prolog, 1, 17
propagar hechos, 80
Propiedades de Agente, 23
Propiedades de Entorno, 23
Propiedades de sistema, 23
Propiedades individuales, 24
Propuestas, 100
Protègè, 66
protocol, 41
Protocolo de interacción de doble secuencia, 88
Protocolo de interacción de suscripciones, 32
Protocolo de interacción interna de doble secuencia, 66, 85
protocolo desdoblado, 89
protocolo IIOP, 45
Protocolos de interacción, 44, 45, 88
Prototipos, 11
Pruebas unitarias, 98
ps.board.agent.debugging.mode, 105
`ps.framework.app.nativelib`, 120
`ps.framework.node.serie`, 91
`PsNodeSettings.java`, *Véase* Node-Layer
puntos de montaje, 69

Q

QUERY-IF, 79

R

RE, *Véase* Respuesta Racional
Read-Evaluation-Print-Loop, 79
`received`, 43
receiver, 40
ReciverBehaviour, 60
Reconocimiento de patrones, 18
Reconocimiento del Habla, *Véase* Hearsay-II
Reddy, D Raj, 1
Refracción, 20
`registerFirstState`, 59
`registerState`, 59
`registerTransition`, 59
Registros, *Véase* Representación de datos
Reinicio en caliente, 97
relays, 55
`removeBehaviour`, 61
Repl, 79
reply-by, 41
reply-to, 40
reply-with, 41
Representación
  de datos, 14
  del Conocimiento, 2, 16
Representación de mensajes mediante cadenas de texto, 32
REQUEST, 79
REQUEST-WHEN, 89
Resiliencia, 23
Resolución de conflictos, 20
Resolución de problemas, 15
Respuesta Racional, 90
Respuestas racionales, 92
RHS, 20
Right Hand Side, 17
Riley, Gary, 87
Rimassa, Giovanni, 32, 46, 52, 55, 57
RMI, *Véase* Java RMI
Rosenschein, Jeffrey S, 11, 21, 23
Round-Robin, 57
Router de salida, 87
RSyntaxtextArea, 70
RSyntaxTextArea, 84
Rule-Based Engines, 67, 69
Rumbaugh, J, 32
Russell & Norvig, 16
Ruta absoluta, 120
Ruta relativa, 120

S

Salience, *Véase* Prioridad de ejecución
`scheduleFirst`, 59
`scheduleNext`, 59
Secuencia planificada, *Véase* Protocolos de interacción
seguridad, 80, 91
Seguridad, 37
selección *FIFO*, *Véase* políticas de bloqueo
selección por Pattern Matching, Véase políticas de bloqueo

sender, 40
sentencias de consulta, 78
sentencias redactadas, 80
sentencias válidas, 76
SequentialBehaviour, 59
SerialBehaviour, 59
Servicio de Directorio, 36
Servicio de páginas amarillas, 33
Servicio de páginas blancas, 36
Servicio de repositorio de tickets, 105
Servicio de Tickets
  Distribucion de componentes, 105
  Interfaz gráfica, 106
Servicio de Transporte de Mensajes, 36
Shell, 13, 87
Shell Asíncrona, 85, 90
Shell de entornos, 74
Shell JADE, 83
Shell Síncrona, 88
Shoham, Yoav, 98
SimpleAchieveREInitiator, 58
SimpleAchieveREResponder, 58
SimpleBehaviour, 58
sistema de ejecución, 46
Sistema de Gestión de Agentes, 36
Sistema Decimal de Dewey, 77
Sistema Experto
  Base de conocimientos, 13
  Componentes esenciales, 13
  Interfaz de usuario, 13
  Memoria de trabajo, 13
  Motor de inferencia, 13
  Shell, 13
Sistemas
  Sistemas basados en conocimiento, 12
  Sistemas de Producción, 16
  Sistemas DPS, 1
  Sistemas Expertos
    definición, 12
    definición Feigenbaum como, 12
Sistemas de disparo paralelo, 20
Sistemas de Pizarra, 21, 24
Sistemas de Producción, 14
  Multiagente, 20
  Paralelos asíncronos, *Véase* sistemas de producción distribuidos
  Paralelos Síncronos, *Véase* sistemas de disparo paralelo
Sistemas de Producción Distribuidos, 20
Sistemas de resolución de problemas
  Basados-en-casos, 21
  Cooperativo y distribuido, 21
Sistemas DPS, *Véase* Sistemas
Sistemas Expertos, 12
  de encadenamiento borroso, 14
  de encadenamiento inverso, 14
  de razonamiento dirigido por datos, 14
Sistemas Multiagente, 21
  Definición Corkill, 27
  Definición de Durfee/Rosenschein, 23
`sniffer.properties`, 102
Sociedades de agentes, 23
Software-colaborativo, 24, 29
SQLtool, 70
`store`, 66
Subsistema de adquisición de conocimiento, 12
SWI-Prolog, 94
System.exit(0);, 75

T

tablón público, 105
Tb2NR, 92
tecla *Tabulador*, 83
Técnicas, 94
  elaboración de *pruebas*, 94
  externalización de los comportamientos, 94
  integración del motor de inferencia, 94
  integración intérprete Java, 94
Técnicas de *migración*, 99
Thread subordinado, 66
ThreadedBehaviour, 60, 61, 91
Threads, 53
TickerBehaviour, 58, 59
Ticket, 78, 79
Ticket-Board to Node Responder, *Véase* TB2NR
Tickets, 93
`to`, 43
Tomaiuolo, Michele, 66, 68
Tools and Utilities, 67
Transiciones de estado, 53
`transport-behaviour`, 43
Tripletes, *Véase* Representación de datos

U

UML, 32
Uncertainty Reasoning, *Véase* Sistemas Expertos de razonamiento borroso
único *Thread*, 57
Universidad de Massachusetts, Véase Amherst
`until -1`, 78
USB, 69
Usuario final, 13
utilidad launcher, 73
Utilidades y herramientas auxiliares, 67

V

Volumetría, 93

W

waiting, Véase Hilo de ejecución
WakerBehaviour, 58, 59
Wooldridge, Michael, 2
WrapperBehaviour, 60

X

XCON, 18
XML, 39

ANEXO A: PRONTUARIOS

A.1 Nomenclatura

Relación de términos empleados para nombrar: paquetes, directorios, archivos, clases, métodos, campos, archivos de soporte y archivos de comportamientos externalizados.

Estructura de paquetes, nombres de clases y contenidos del *Directorio de aplicación*:

Paquetes:
Librería principal:
[*psnode-agents-0.9.1.jar*]
(·)
- examples
  - psnode
    - core
      - behaviours
      - engines
    - domain
    - util
      - images

Contenidos
(·) = utilidades aux. *psnode*, *launcher* y *shell*
examples = paquete de ejemplos en JADE
.psnode = agentes con motor integrado
.domain = ontología Problem-Solvers
.core.behaviours = comportamientos
.core.engines = prototipos de motor de inferencia
.util = i18n, agentes especiales y servicios
.util.images = iconos y logos

Programa de instalación:
[*dps-frameworkBuilder-0.9.1.jar*]
(·)
- examples.psnode.util
- wsframe
- wsnative

Contenidos:
(·) = utilidad auxiliar *framework*
.wsframe = estructura y contenidos de un *nodo*
.wsnative = *directorio de aplicación* y librerías
.util = servicio de desempaquetado de archivos

Directorio de aplicación:
(nombre-de-aplicación)
- config
- lib
- logs
- nodes

Subdirectorio /config***:***
commandsCLIPS.language (completado sintaxis)
commandsCLIPS.patterns (declaraciones)
commandsJADE.language (completado sintaxis)
commandsJESS.language (completado sintaxis)
commandsJESS.patterns (declaraciones)
framework.methods.bsh (técnica de *runlevels*)
framework.properties (propiedades *Aplicación*)
help.board.txt (ayuda on-line PsBoardAgent)
help.clips.txt (API-CLIPSJNI en Entorno Desa.)
help.jade.txt (API JADE en Entorno Desa.)
help.java.txt (Lenguaje Java en Entorno Desa.)
help.jess.txt (API Jess en Entorno de Desarrollo)
help.menu.txt (códigos acceso a ayudas on-line)
help.node.txt (variables de PsNodeAgent)

help.shell.txt (Java Scripting con BeanShell)
help.ss64.txt (OS-X commands)
java.shell.bsh (configuración utilidad *shell java*)

Subdirectorio /lib/**:**
./lib/ rsyntaxtexarea-2.5.7.jar
./lib/ bsh-2.0b4.jar
./lib/ commons-codec-1.3.jar
./lib/ CLIPSJNI.jar
./lib/ jess.jar
./lib/ jsr94.jar
./lib/ hsqldb.jar
./lib/ sqltool.jar
./lib/ jade.jar
./lib/ psnode-agents-0.9.1.jar

Pasarela CLIPS Apple:
./lib/mac/ libCLIPSJNI.jnilib
Pasarela CLIPS Win32*:*
./lib/w32/ CLIPSJNI.dll
Pasarela CLIPS Win64*:*
./lib/ w64/CLIPSJNI.dll
Pasarela CLIPS Linux32*:*
./lib/ux32/ libCLIPSJNI.so
Pasarela CLIPS Linux64*:*
./lib/ux64/ libCLIPSJNI.so

Subdirectorio /lib/jena/**:**
./lib/jena/ commons-logging-1.1.1.jar
./lib/jena/ icu4j_3_4.jar
./lib/jena/ iri.jar
./lib/jena/ j2cf.jar
./lib/jena/ jena.jar
./lib/jena/ json.jar
./lib/jena/ log4j-1.2.12.jar
./lib/jena/ lucene-core-2.3.1.jar
./lib/jena/ slf4j-api-1.7.10.jar
./lib/jena/ slf4j-log4j12-1.7.10.jar
./lib/jena/ stax-api-1.0.jar
./lib/jena/ wstx-asl-3.0.0.jar
./lib/jena/ xercesImpl.jar

Subdirectorios**:** /nodes/nombre-de-agente**/:**
Agente con motor integrado /nodes/PsNode/config**/:**
/config/bhv.acpb.bsh
/config/bhv.bkwm.bsh
/config/bhv.e2aa.bsh
/config/bhv.e2as.bsh
/config/bhv.e2nr.bsh
/config/bhv.jamo.bsh
/config/bhv.n2hr.bsh
/config/bhv.n2nr.bsh
/config/bhv.nmig.bsh
/config/bhv.rswm.bsh
/config/bhv.v2ad.bsh
/config/bhv.v2ai.bsh
/config/level.00.bsh
/config/level.01.bsh
/config/level.03.bsh
/config/level.05.bsh
/config/level.06.bsh
/config/node.methods.bs
/config/node.properties
/config/notebook.txt
/config/test.async.bsh

Agente con motor integrado /nodes/PsNode/engine/**:**
/engine/cam.clp
/engine/dilemma.clp
/engine/electrnc3.clp
/engine/fullmab.clp
/engine/grid2x2-p1.clp
/engine/grid3x3-p1.clp
/engine/grid3x3-p10.clp
/engine/grid3x3-p11.clp
/engine/grid3x3-p12.clp
/engine/grid3x3-p13.clp
/engine/grid3x3-p14.clp
/engine/grid3x3-p15.clp
/engine/grid3x3-p16.clp
/engine/grid3x3-p17.clp
/engine/grid3x3-p18.clp
/engine/grid3x3-p2.clp
/engine/grid3x3-p3.clp
/engine/grid3x3-p4.clp
/engine/grid3x3-p5.clp
/engine/grid3x3-p6.clp
/engine/grid3x3-p7.clp
/engine/grid3x3-p8.clp
/engine/grid3x3-p9.clp
/engine/grid4x4-p1.clp
/engine/grid4x4-p10.clp
/engine/grid4x4-p13.clp
/engine/grid4x4-p17.clp
/engine/grid4x4-p3.clp
/engine/grid4x4-p4.clp
/engine/grid4x4-p8.clp
/engine/grid4x4-p9.clp
/engine/grid5x5-p1.clp
/engine/grid5x5-p11.clp
/engine/grid5x5-p3.clp
/engine/grid5x5-p4.clp
/engine/grid5x5-p6.clp
/engine/grid5x5-p9.clp
/engine/hanoi.clp
/engine/hard.clp
/engine/hello.clp
/engine/output-frills.clp
/engine/output-none.clp

/engine/output-simple.clp
/engine/solve.clp
/engine/stack.clp
/engine/sudoku.clp

Agente con motor integrado /nodes/PsNode/var/**:**
/var/annotation_collaborativeNewspaper.rdf
/var/class.AgentPrepareODEVICE2.bsh
/var/class.AgentRestartODEVICE2.bsh
/var/collaborativePizza.owl
/var/food.owl
/var/MotorVehicle.rdf
/var/pizza.owl

Agente sin motor integrado /nodes/PsAgent/config/**:**
/config/bhv.jamo.bsh
/config/bhv.nmig.bsh
/config/class.JAMONodeManagement.bsh
/config/class.NMIGManagement.bsh
/config/class.SudokuREInititator.bsh
/config/level.00.bsh
/config/level.01.bsh
/config/level.03.bsh
/config/level.05.bsh
/config/level.06.bsh
/config/node.methods.bsh
/config/node.properties

Agente Servicio de Tickets /nodes/PsBoard/config/**:**
/config/bhv.bcpb.bsh
/config/bhv.grid.bsh
/config/bhv.jamo.bsh
/config/board.methods.bsh
/config/board.properties
/config/class.B2NR.bsh
/config/class.BoardCapabilitiesLoader2.bsh
/config/class.BoardGridLoader2.bsh
/config/class.JAMONodeManagement2.bsh
/config/level.00.bsh
/config/level.01.bsh
/config/level.03.bsh
/config/level.05.bsh
/config/level.06.bsh

Agente Servicio de Tickets /nodes/PsBoard/var/**:**
/var/ticketDB.data
/var/ticketDB.log
/var/ticketDB.properties
/var/ticketDB.script
/var/tikets.log

Agente DummyAgent /nodes/Dummy/config/**:**
/config/bhv.bkDm.bsh
/config/bhv.gdae.bsh
/config/bhv.shab.bsh
/config/class.BKDMBackupDummy.bsh
/config/class.GENERICDaemon.bsh
/config/level.00.bsh
/config/level.01.bsh
/config/level.03.bsh
/config/level.05.bsh
/config/level.06.bsh
/config/node.methods.bsh
/config/node.properties

Clases de Agente:
Agente con motor integrado
[*examples.psnode.PsNodeAgent*]
Agente de soporte sin motor integrado
[*examples.psnode.util.PsAgent*]
Agente repositorio de Tickets
[*examples.psnode.PsBoardAgent*]
Agente entorno de desarrollo
[*examples.psnode.util.PsPocket*]
Agente de ejemplo de extensión DummyAgent
[*examples.psnode.util.PsDummy*]

Clases de las utilidades auxiliares:
[*framework*]
[*launcher*]
[*psnode*]
[*shell*]

Clases de la ontología *Problem-Solvers*:
Paquete:
examples.psnode.domain
Definición:
[ProblemSolversOntology]
Conceptos:
[Dewey] = Código identificador
[Phase] = Fase de resolución
[Ticket] = Resultados de resolución
Interfaces y comportamientos:
[PsoNames] = Interface de constantes
[RMNGBehaviour] = Comportamiento
Verbos:
[Process] = Procesar en motor de inferencia
[Repl] = Procesar y responder al actor-humano
Gestión de Tickets:
[Advertise] = Remitir Ticket a PsBoardAgent
[ProblemFor] = Consultar Tickets pendientes
Acciones especiales:
[Migrate] = Consultar viabilidad de migración

Clases de comportamiento de los agentes:
Paquete:
examples.psnode.core.behaviours

Clases de utilidad:
[BhConstants] = Constantes utilizadas
[MTRepository] = MessageTemplates Repo.

Comportamientos de carácter Basal:
[ACPBEngineLoader] = Agent Capabilities
[JAMONodeManagement] = JADE Mngmt.
[BoardCapabilitiesLoader] = Board Capabilities
[BoardGridLoader] = Cargador Recorset tickets
[TBSITicketBoardREInitiator] = Registrador DF

Comportamientos de Respaldo y Restauración:
[BKWMDaemonState] = Respaldo WM ticker
[BKWMExecutorState] = Respaldo WM exec.
[BKWMFSMBehaviour] = FSM de Respaldo
[RSWMFSMBehaviour] = Restaurar WM
[BKDMBackupDummy] = Respaldo Dummy

Comportamientos de Interacción e Interpretación:
[E2AAEngineInterpreter] = Analizador
[E2ASAsynchronousShell] = Shell Asíncrona

Comportamientos del motor de inferencia integrado:
[E2NRFSMBehaviour] = FSM gestión de motor
[E2NRCaptorState] = Intérprete de solicitudes
[E2NRProcessState] = Procesador de solicitudes
[E2NRNoticerState] = Notificador de solicitudes

Comportamientos de los Protocolos de Interacción:
[N2BREInitiator] = Iniciador consultas a Board
[N2BREResponder] = Respondedor del Board
[N2NREInitiator] = Iniciador consultas a Nodos
[N2NREResponder] = Respondedor de Nodos

Comportamientos genéricos tipo Daemon-Executor:
[GENERICDaemonState] = Ticker genérico
[GENERICExecutorState] = Exec. genérico
[SHABFSMBehaviour] = FSM Shock-Absorber
[SHABDaemonState] = Generic Shock-Abs.
[SHABExecutorState] = Generic Executor
[GGCCGarbageCollector] = Limpiador de msg.

Comportamientos de soporte a la Migración:
[MIABMessageProposer] = Proponer migración
[NMIGManagement] = Gestión de migración
[H2NREInitiator] = Iniciador de migración
[N2HREResponder] = Respondedor de migrac.

Comportamientos de Gestión de Reglas Activadas:
[V2ADAgendaDaemonState] = Visitador de Agenda
[V2ADAgendaExecutorState] = Ejecutor de reglas
[V2ADFSMBehaviour] = Gestión de la Agenda

Clases de motores de inferencia:
Paquete:
examples.psnode.core.engines
Interface:
[RBEngine]

Métodos:
.getBuffer()
.getRBEngine()
.getState()
.getVersion()
.innerCommand(int, String)
.innerCommand(int, String, String)
.innerCommand(String)
.innerCommand(String, String)
.innerCommand(String, String, String)
.innerShell()
.outputBuffer()
.resetBuffer()
.retainedBuffer()
.setRouter(Object)
.setThGroup(ThreadGroup)
.setThName(String)
.startBuffer()
.stopBuffer()

Clases Prototipo que extienden a [RBEngine]**:**
[RBEngineCLIPS]
[RBEngineJESS]

Métodos:
.RBEngineCLIPS()
.RBEngineCLIPS(Environment)
.RBEngineCLIPS(int)
.RBEngineCLIPS(Object)
.RBEngineCLIPS(Object, ThreadGroup, String)
.RBEngineCLIPS(ThreadGroup, String)
~
.RBEngineJESS()
.RBEngineJESS(Object)
.RBEngineJESS(Object, ThreadGroup, String)
.RBEngineJESS(Rete)
.RBEngineJESS(ThreadGroup, String)

Clases de soporte a consolas y routers:
[ConsolePanelCLIPS]
[ConsolePanelJESS]
[ConsoleRouterCLIPS]
[ConsoleRouterJESS]
[RBEngineCLIPSRouter]
[RBEngineJESSRouter]

Clases de utilidad:
Paquete:
examples.psnode.util

Servicios:
[CommandLineReader] = Shell en terminal
[JadeConsole] = Shell - JADE en agentes
[PsConstants] = Interface de constantes
[psnode_messages.properties] = recursos i18n

Definición de Nodo:
[PsNodeSettings] = Objetos tipo Nodo

Clase de Servicios del Directorio de aplicación:
[PsService] = utilidades estáticas
Métodos:
.addNativeLibraryPath()
.bhvState(String)
.createFramework(String, boolean, boolean)
.DirList(String)
.getDpsAllEnvironment()
.getDpsJAVAClassesURLs()
.getDpsJAVAClassPaths()
.getDpsJVMProperties()
.getDpsNativleLibURLs()
.getDpsOSEnvironment()
.getJadeSettings(String)
.getJVMEnv()
.getNativeLocalPath()
.getNativePath()
.getNativeRelativePath()
.getOSarch()
.getOSname()
.i18n(String)
.isDPSCandidatePath(String)
.isDPSFramework()
.isDPSFramework(String)
.isPsAgentDir(String)
.isPsBoardAgentDir(String)
.isPsNodeAgentDir(String)
.main(String[])
.makeNewPsNode(String)
.makeNewPsNode(String, boolean)
.zipDir(String, String)

Agentes de soporte:
[PsAgent] = Agente de soporte a migraciones
[PsAgentGui] = Interfaz gráfica de PsAgent

Métodos:
.doDelete()
.interpreterBSh()
.jadeConsole()
.migrateNodeAction(Migrate, ACLMessage)
.myBOB()
.myGui()
.myNode()
.requestMigrate(String)
.requestShowGui(String)
.showGuiAction(ShowGui, AID)

Agentes de ejemplo de Entorno de Desarrollo:
[PsDummy] = Extensión IDE de DummyAgent
[PsDummyGui] = Interfaz gráfica de PsDummy

Agente Entorno de Desarrollo:
[PsPocket] = Extensión IDE de PsNodeAgent
Métodos:
.doDelete()
.getJspShEng()
.interpreterBSh()
.jadeConsole()
.makeReplMessage()
.makeReplMessage(String)
.makeReplMessage(String, String)
.makeReplMessage(String, StringBuilder)
.makeTrace(ACLMessage)
.makeTrace(String, ACLMessage)
.myBOB()
.myEngine()
.myGui2()
.myNode()
.showGuiAction(ShowGui, AID)
.threadBhv()

Agente Entorno de Desarrollo (GUI):
[PsPocketGUI] = Interfaz Gráfica de PsPocket
Métodos:
.PsPocketGUI(PsPocket, PsNodeSettings)
.addMessageNode(String, ACLMessage)
.appendToPane(JTextPane, String, Color, int)
.appendToPane(String)
.appendToPane(String, Color, int)
.disposeAsync()
.makeRequestECHO(ACLMessage, String)
.refreshStatusBar()
.setItsMsg(ACLMessage)
.setJspShEng(JScrollPane)
.setUI2Metal()
.setUI2Motif()
.setUI2Nimbus()
.setUI2Windows()

Nomenclatura de métodos y campos accesibles en objetos de la *Shell-JADE*:

Objetos:
[logger] = *jade.util.logger*
[myAgent] = El propio objeto agente
[myNode] = El objeto node de dicho agente
[myBOB] = Comportamientos externalizados
[myEngine] = Motor prototipo
[innEngine] = Objeto motor de inferencia
[myContainer] = Contenedor del agente
[myGui2] = Interfaz Gráfica de PsPocket
[myGui] = Interfaz Gráfica de PsNodeAgent

[myBSh] = Objeto Intérprete Java BeanShell
[myShell] = Router de salida de datos del motor

Variables:
frameworkName = nombre Directorio de Aplic.
frameworkRootPath = Path del Directorio de A.
frameworkConfigPath = Path de /config/
frameworkLogPath = Path de /logs/
frameworkVarPath = Path de /var/
FileSeparator = Separador de directorios
nodeName = Nombre del nodo
nodeRootPath = Path hacia el nodo
nodeConfigPath = Path de /config/ del nodo
nodeEnginePath = Path de /engine/ del nodo
nodeVarPath = Path de /var/ del nodo
deweyDecimalCode = Código Dewey
descriptionOfRole = Descripción del agente
solverFromPhase = Desde la fase de resolución
solverUntilPhase = Hasta la fase de resolución
nameOfTicketsBoard = Board por defecto

Propiedades:
sCyclesOfInferenceSerie = Serie de ciclos de ejecución de disparo de reglas en el motor
sMaxCyclesOfInference = Máximo número de ciclos de disparo por cada visita a la Agenda.
nodeProp2 = String[] de node.properties

Métodos:
init(integer)
myBOB() .perspective()
print(this.varables)
myGui2 .addMessageNode(String, ACLMessage)
myEngine .setRouter(myEngine)
myEngine .setRouter(myShell)
setFont(jconsole, 14)
setAccessibility(true)

Objeto: *myNode* (sólo métodos principales):
.PsNodeSettings(Agent)
.getProperties()
.getProperties(String)
.getProperty(String)
.perspective()
.printDebugging()
.printLaunchParameters()
.printLoggin()
.refresh()
.scenario()
.setProperty(String, String)
.toString()

Objeto: *myAgent* (sólo métodos principales):
.addBehaviour(Behaviour)
.getAgentState()
.getAID()
.getAID(String)
.getAMS()
.getArguments()
.getBootProperties()
.getContainerController()
.getContentManager()
.getCurQueueSize()
.getDefaultDF()
.getHap()
.getLocalName()
.getName()
.getQueueSize()
.getState()
.here()
.isRestarting()
.postMessage(ACLMessage)
.putBack(ACLMessage)
.receive()
.receive(MessageTemplate)
.removeBehaviour(Behaviour)
.send(ACLMessage)

A.2 Hoja de Referencia

Resumen en *una-página*, con ejemplos para mostrar en breves pasos cómo iniciar un proceso de transformación o creación de aplicaciones basadas en *sistemas de producción*, en aplicaciones distribuidas compuestas por *agentes*.

Suite de herramientas para desplegar Sistemas de Producción

Aplicaciones en **CLIPS** y **JESS** desplegadas mediante agente en la plataforma **JADE**

HOJA DE REFERENCIA

Departamento de Ingeniería Eléctrica y de Sistemas y Automática
Escuela de Ingenierías Industrial e Informática
Universidad de León http://www.unileon.es

Tesis doctoral: Técnicas para despliegue de arquitectura distribuida en sistemas expertos basados en reglas empleando el paradigma multiagente

Cómo crear Directorios de aplicación

Utilizar el programa de instalación [dps-frameworkBuilder.jar] para crear réplicas de la aplicación desplegadas en servidores gestionados por JADE. Cuando sólo se utiliza la opción [--new], se incorporan la documentación y las librerías Java y puede ser utilizado como un entorno de desarrollo de agentes:

```
$ java -jar dps-frameworkBuilder-0.9.1.jar
$ java -jar dps-frameworkBuilder-0.9.1.jar new app51
$ cd app51
```

Cómo configurar el acceso a las Librerías de Java

Existen dos modos: Local e Infraestructura. El modo-Local se establece por defecto cuando se crea el *Directorio de aplicación*. Requiere establecer el valor de la variable CLASSPATH como ruta relativa:

```
app51\$ export CLASSPATH=lib/*:                    **Nota
```

El modo-Infraestructura se utiliza cuando se han centralizado las librerías Java i.e. (/var/dps/lib). Requiere entonces localizar las librerías nativas de CLIPS en el archivo [./config/framework.properties]:

```
app51\$ export CLASSPATH=/var/dps/lib/*:
Editar archivo: [./config/framework.properties]
Variable: ps.framework.app.nativelib = /var/dps/lib
```

Cómo crear nuevos Nodos

Se generan con la utilidad auxiliar psnode o clonando el directorio de otro *nodo* ubicado en el directorio /nodes/ del Directorio de aplicación. Es una estructura de archivos en el sistema de ficheros empleada por los agentes de tipo [PsNodeAgent y PsAgent]:

```
app51\$ java psnode new Yarvis
app51\$ java psnode --new=Robot21,Robot41,Robot81
```

Cómo desplegar agentes en la plataforma

Emplear la utilidad *launcher* desde el Directorio de aplicación.

Lanzar la plataforma JADE

Los parámetros de *launcher* para iniciar la plataforma JADE son: `platform`, nombre de Host y puerto (Linux/OS X *vs.* MS-Windows):

```
[user.name\~\app51\$     java launcher platform localhost &
c:\pruebas\app51 > start java launcher platform localhost
```

Desplegar un agente [PsNodeAgent]

Los parámetros para desplegar un agente con motor integrado son: `node`, Host, nombre de agente, tipo de motor y puerto de escucha.

```
app51\$ java launcher node localhost Yarvis Jess
```

Desplegar una colección de agentes [PsNodeAgent]

Los parámetros para desplegar la colección de agentes son: `nodes`, Host, tipo de motor y puerto de escucha.

```
app51\$ java launcher nodes localhost jess            or...
app51\$ java launcher nodes localhost clips 4455  &
```

Desplegar un agente [PsNodeAgent] en su *Entorno de desarrollo*

Los parámetros son: `pocket`, Host, nombre de agente, tipo de motor de reglas y puerto de escucha:

```
app51\$ java launcher pocket localhost Robot21 clips
```

Desplegar un *Entorno de desarrollo* sin agente asociado

Desplegar un *entorno de desarrollo* en vacío mediante:

```
app51\$ java launcher pocket                              ***Nota
app51\$ java launcher pocket localhost Pocket jess 4455
```

Desplegar un agente [PsAgent] sin motor de inferencia integrado

Los parámetros son: `agent`, Host, nombre y puerto de escucha:

```
app51\$ java launcher agent localhost Robot41
```

Desplegar agente [DummyAgent] en su *Entorno de desarrollo*

Los parámetros son: `dummy`, Host, nombre y puerto de escucha:

```
app51\$ java launcher dummy localhost Robot81
```

Intérpretes: Shell {-Java, -Clips, -Jess, -hsql}

Accesibles con la utilidad auxiliar *shell* desde el *Directorio de aplicación*. Facilitan el proceso de transformación de aplicaciones.

Intérprete de comandos en Lenguaje Java

Apertura de un Intérprete de Lenguaje Java, con las clases principales de la API JADE, CLIPS y Jess incorporadas:

```
app51\$ java shell java
[framework://app51/]# > _
```

Intérprete de comandos en Lenguaje CLIPS

Apertura de la Shell del intérprete CLIPS (6.31) desde máquina Java:

```
app51\$ java shell clips
CLIPS> _
```

Intérprete de comandos en Lenguaje Jess

Apertura de la Shell del intérprete *Jess* (8.0a1) en modo depuración:

```
app51\$ java shell jess
Jess> _
Es un `alias` de: app51\$ java jess.Main -developer
```

Intérprete de comandos en Lenguaje SQL

Apertura de cliente SQL y conexión a BBDD de agentes TicketBoard.

```
app51\$ java shell hsql [-gui]
sql> \j SA password jdbc:hsqldb:file:nodes/Board/var/ticketDB
sql> SELECT * FROM tickets;
```

Entorno de desarrollo de agentes [PsNodeAgent]:

Un agente [PsNodeAgent] desplegado en su *Entorno de desarrollo*, permite acceder a: archivos de comportamientos, archivos del sistema experto, Shell API-JADE, Shell síncrona a motor de inferencia y Shell asíncrona a motor/es de inferencia; en tiempo de ejecución.

Pestaña del intérprete: Shell-JADE

Primera pestaña de agentes [PsNodeAgent] en entorno de desarrollo:

```
Robot21/config/$ > myAgent .getLocalName();
Robot21/config/$ > msg4 = new ACLMessage(ACLMessage.INFORM);
Robot21/config/$ > javap( msg4 );
Robot21/config/$ > myNode .perspective();
```

* **DPS-Framework**: Deploy Production Systems - Framework.
** **Nota**: en MsWindows, es necesario cambiar **`:`** por **`;`** y %**CLASSPATH%**
*** **Nota**: Alias de: **java launcher pocket localhost Pocket clips**
&: En MsWindows, ejecutar en background con: **start java platform...**

Autor: Estudiante de doctorado *Francisco J. Aguayo Canela*
Dirigido por: Dr. *Isaías García Rodríguez*

Barra de botones: Niveles de ejecución o (run-levels)
Se han creado 5 niveles {0, 1, 3, 5 y 6}. Corresponden a metaestados: apagado, inicializado, comunicaciones internas, comunicaciones externas y reiniciado del agente. Se activan desde la barra superior o también, de manera manual, desde el intérprete Shell-JADE con:

```
[ Agent ]://Robot21/config/$ > init();
```

Barra de botones: Migración del agente
El proceso de migración se activa desde el botón [Migrate] o, de manera manual, desde el intérprete Shell-JADE con:

```
//Robot21/config/$ > myAgent .requestMigrate( "Robot41" );
```

Donde 'Robot41' es el nombre de un agente de su *Directorio de aplicación*, al que se le solicita la colaboración en el proceso de migración de 'Robot21' hacia el *contenedor* de 'Robot41'.

Variables y métodos intrínsecos del intérprete: Shell-JADE
El manual de BeanShell es accesible desde [/var/doc/dps/index.html] En el Entorno de desarrollo se puede acceder a métodos intrínsecos:

```
[ Agent ]://Robot21/config/$ > …
 dir();                    ¦ Interface al Motor de inferencia.
 pathToFile();             ¦ El propio objeto Agente.
 this.variables            ¦ Colección variables en Namespace.
 this.methods              ¦ Colección de métodos en Namespace.
 source( "  " )             ¦ Cargar un archivo en Namespace.
 $_                        ¦ Último valor devuelto.
 print();                  ¦ Imprimir en consola BeanShell.
 exit();                   ¦ Finalizar entorno y agente.
 clear();                  ¦ Limpiar la consola Shell-JADE.
```

Y a los métodos y variables extrínsecas del agente:

```
[ Agent ]://Robot21/config/$ > …
 help();                   ¦ Ayuda on-line por área temática.
 myAgent                   ¦ El propio objeto Agente.
 myEngine                  ¦ Interface del motor del agente.
 innEngine                 ¦ Referencia directa al motor.
 myNode                    ¦ Propiedades del nodo del agente.
 myNode.perspective();     ¦ Volcado de propiedades del nodo.
 myBOB                     ¦ Colección de comportamientos.
 myContainer               ¦ Contenedor del agente.
 myBOBActivated            ¦ Comportamientos activados.
 frameworkVarPath          ¦ Path a directorio ./var
 frameworkLogPath          ¦ Path a directorio ./logs
 frameworkRootPath         ¦ Path al Directorio de aplicación.
 relativeEnginePath        ¦ Path relativo a ./engine
 relativeConfigPath        ¦ Path relativo a ./config
 relativeNodePath          ¦ Path relativo al nodo.
 nodeEnginePath            ¦ Path absoluto a ./engine del agente.
 nodeConfigPath            ¦ Path absoluto a ./config del agente.
 nodeRootPath              ¦ Path absoluto al nodo del agente.
 setFont(jconsoe, 12);     ¦ Tamaño de fuente en consola.
 hook("nombreAgente");     ¦ Hacer visible un agente, etc., etc.
```

Pestaña: Editores de guiones, configuración y archivos del SP:
Pueden editarse los archivos: [nodes.properties], niveles de ejecución, clases de comportamientos y los archivos [bhv.xxxx.bsh] destinados a crear nuevos objetos Behaviour. La colección [myBOB] contiene la relación de objetos Behaviour organizados por nivel de ejecución. Su declaración se localiza en [node.methods.bsh]

```
=Level= ===== ======= ============== ======================
Mak Act. Bhv   Area    File name      Function asigned
=== === ===== ======= ============== ======================
0 ¦ 1 ¦ jamo ¦ basal ¦ bhv.jamo.bsh ¦ JADE Mngmt. Ontology.
0 ¦ 0 ¦ acpb ¦ basal ¦ bhv.acpb.bsh ¦ Agent Capabilities.
1 ¦ 3 ¦ n2nr ¦ RE_re ¦ bhv.n2nr.bsh ¦ Node To Node Responder.
1 ¦ 3 ¦ e2nr ¦ RE_re ¦ bhv.e2nr.bsh ¦ Engine To Node Responder
3 ¦ 5 ¦ e2as ¦ Human ¦ bhv.e2as.bsh ¦ Engine To Agent Shell.
1 ¦ 3 ¦ e2aa ¦ Robot ¦ bhv.e2aa.bsh ¦ Engine To Agent Analyzer
3 ¦ 5 ¦ v2ad ¦ demon ¦ bhv.v2ad.bsh ¦ Visitor To Agenda Daemon
7 ¦ 7 ¦ v2bd ¦ demon ¦ bhv.v2bd.bsh ¦ Visitor To Board Daemon.
3 ¦ 5 ¦ bkwm ¦ Mngmt ¦ bhv.bkwm.bsh ¦ Backup Engine WM + KB.
3 ¦ 5 ¦ rswm ¦ Mngmt ¦ bhv.rswm.bsh ¦ Restore Engine WM + KB.
0 ¦ 1 ¦ nmig ¦ Mngmt ¦ bhv.nmig.bsh ¦ Node Migration Mngmt.
```

En la cuarta pestaña pueden editarse los archivos del *Sistema de Producción* que se hayan asignado específicamente a cada agente.

Pestañas de intérpretes del motor: Shell-Síncrona y Shell-Asíncrona
Desde la quinta y sexta pestañas se accede al motor de inferencia del agente. La Shell-Síncrona se encuentra activa en *nivel de ejecución* [n-1]. La Shell-Asíncrona se activa en [n-5] y permite acceder tanto, al motor de inferencia interno, como al motor de inferencia de agentes del mismo Directorio de aplicación (local o remoto).

Entorno de agentes [PsBoardAgent]:

Agentes que ofrecen el servicio de almacenamiento y entrega de ***Tickets*** a los agentes. Por convenio estos agentes registran sus capacidades en la plataforma JADE en el nivel [n-3]. Se despliegan con la utilidad *launcher* mediante: `board`, nombre de Host, nombre agente TicketBoard y puerto de escucha:

```
app51\$ java launcher platform localhost 4455 &
                    (Desde pantalla: Menu-->Tools-->Show the DF GUI )
app51\$ cp -a nodes\Board nodes\tb02
app51\$ java launcher board localhost tb02 4455 &
```

Los datos se actualizan dinámicamente en la tabla en el nivel [n-5]. La base de datos [ticketDB] se encuentra en el directorio privado /var/ de cada agente [PsBoardAgent] i.e. (app51/nodes/tb02/var/ticketDB).

Cómo crear mensajes de tipo: Acción Problem-Solvers REPL
Desde la Shell-Asíncrona se crean mensajes Problem-Solvers de tipo REPL al presionar [Ctrl+Mayús.+Enter] en la consola. Ese evento puede emularse desde la pestaña Shell-JADE con:

```
//Robot21/config/$> msgRepl = myAgent .makeReplMessage(String receiverLo-
calName, String commandToEval);
```

Cómo crear mensajes de tipo: Acción Problem-Solvers PROCESS
Pueden crearse manualmente desde la Shell-JADE con:

```
msgProc = myAgent .makeProcessMessage(String receiver, String
                                        sCode, String sValue);
```

Cómo crear objetos: PSO-Ticket y mensajes PSO-ADVERTISE

```
ticket22 = myAgent .makeTicket(dewey , phase, theProcess);
msgAdv = myAgent .makeAdvertiseMessage(ticket22);
```

Casos especiales de integración de aplicaciones

En situaciones donde sólo es necesario exponer *hechos*, *reglas* y *conocimientos* de una aplicación en un agente, puede pasarse la referencia interna de su motor de inferencia hacia un agente, e.g.:

```
c:\pruebas\app51\> start java launcher platform localhost
c:\pruebas\app51\> java shell java  ↓
[framework://app51/]# >
  clipsEngine     = new Environment();
  clipsEngine.eval( "(assert (person Frank))" );
  rt2    = jade.core.Runtime.instance();
  prof2  = new jade.core.ProfileImpl();
  cc2    = rt2.createAgentContainer( prof2 );
  args   = new Object[] { "CLIPS", clipsEngine };
  bot41  = cc2.createNewAgent( "Robot41",
                  "examples.psnode.util.PsPocket", args);
  bot41.start();
   ...
  bot41.kill();
```

JADE: Java Agent Development Environment: CSELT, S. & TILab, S. v 4.5.0
CLIPS: C++ Language Integrated Production System v. 6.31
BeanShell: Java Bean Interpreter Shell v. 2.04b

Jess: Java Expert System Software v. 8.0a1
RSyntaxTextArea: Java Rich Syntax Text Area Component v. 2.5.7