

ÍNDICE

1. INTRODUCCIÓN.....	4
2. AGENTES Y SISTEMAS MULTIAGENTE (SMA).....	7
2.1. Propiedades de los agentes.....	7
2.2. Arquitecturas de agentes.....	7
2.3. Tipos de agentes.....	8
2.4. Sistemas Multiagente (SMA).....	8
2.4.1. Ventajas de un SMA.....	8
2.4.2. Gestión de un SMA.....	9
2.4.3. Lenguaje de comunicación entre agentes (ACL).....	10
2.4.3.1. Elementos de un mensaje.....	10
2.4.3.2. Protocolos de comunicación.....	12
3. JADE.....	13
3.1 Paquetes de JADE.....	13
3.2 Principales componentes de JADE.....	14
3.3 Agentes y sus diferentes Comportamientos.....	14
3.4 Mensajes entre Agentes.....	15
3.5 Ontología.....	16
3.6 Protocolos de Comunicación.....	16
3.7 Herramientas de JADE.....	19
4. ARQUITECTURA DEL SISTEMA.....	23
4.1 Ventajas del uso de sistemas multiagente.....	26
4.2 Protocolo de Comunicación.....	26
4.3 Mensajes.....	27
4.4 Ontología.....	28
5. TOMA DE DECISIONES MULTICRITERIO CON ATRIBUTOS LINGÜÍSTICOS.....	31
5.1. Funciones de Negación.....	31
5.2. 2-tupla.....	32
5.3. Conjuntos Difusos (Fuzzy Sets).....	33
6. AGENTES DEL SISTEMA.....	34
6.1. Coordinador de transplantes.....	34
6.2. Especialista en Transplantes.....	36
6.3 Agente ClusDM.....	36
6.4 Agente TwoTuple.....	36
6.5 Agente FuzzyDM.....	39
7. DISTRIBUCIÓN MODULAR DEL SISTEMA.....	43
7.1. CoordinadorAgent.java.....	43
7.2. DadesEspecialista.java.....	45
7.3. Especialista.java.....	46
7.4. EspecialistaAgent.java.....	47
7.5. DatosTwoTuple.java.....	49
7.6. TwoTuple.java.....	49
7.7. TwoTupleAgent.java.....	50

7.8. DatosFuzzySet.java.....	51
7.9. FuzzySet.java.....	51
7.10. FuzzySetAgent.java.....	52
7.11. ResultatMCDM.java.....	53
7.12. Donant.java.....	53
7.13. MantDonant.java.....	54
7.14. Pacient.java.....	55
7.15. MantPacient.java.....	56
7.16. agentePuntuacion.java.....	57
7.17. MantResultados.java.....	57
7.18. Paquete Gráfico.....	58
7.19. Paquete Ontology.....	58
8. Manual de Usuario.....	61
8.1. Menú.....	61
8.1.1. Archivo.....	61
8.1.2. Pacientes.....	62
8.1.3. Donantes.....	63
8.1.4. Agentes.....	64
8.2. Pestañas.....	64
9. Juego de pruebas y comparativa.....	65
10. Código Fuente.....	77
10.1. agentePuntuacion.java.....	77
10.2. ClusDM.java.....	77
10.3. ClusDMResponderAgent.java.....	101
10.4. CoordinadorAgent.java.....	109
10.5. DadesClusDM.java.....	122
10.6. DadesEspecialista.java.....	124
10.7. DadesFuzzySet.java.....	133
10.8. DadesTwoTuple.java.....	134
10.9. Donant.java.....	135
10.10. Especialista.java.....	137
10.11. EspecialistaAgent.java.....	148
10.12. FuzzyAgent.java.....	166
10.13. FuzzySets.java.....	174
10.14. MantDonant.java.....	184
10.15. MantPacient.java.....	190
10.16. MantResultados.java.....	197
10.17. ObjetoGrado.java.....	202
10.18. Pacient.java.....	202
10.19. ResultatMCDM.java.....	204
10.20. Triangulo.java.....	204
10.21. TwoTuple.java.....	204
10.22. TwoTupleAgent.java.....	209
10.23. VocabularioBase.java.....	217
10.24. Paquete Ontology.....	218
10.24.1. AlgoritmeNoImplementat.java.....	218
10.24.2. Atrib_objectes.java.....	218
10.24.3. Atribut.java.....	219

10.24.4. ClassesInsuficients.java.....	221
10.24.5. DatosNoValidos.java.....	221
10.24.6. Domain_OntologyObj.java.....	222
10.24.7. EspecialistaRequest.java.....	222
10.24.8. Grau.java.....	224
10.24.9. MassaAtributs.java.....	225
10.24.10. MassaObjectes.java.....	225
10.24.11. MCDMOntology.java.....	226
10.24.12. MCDMRequest.java.....	233
10.24.13. Negacio.java.....	235
10.24.14. Peso.java.....	235
10.24.15. PesosNoValids.java.....	236
10.24.16. Resultat.java.....	237
10.24.17. Triangle_difus.java.....	238
10.24.18. Valor_atribut.java.....	239
10.25. Paquete Gráfico.....	240
10.25.1. gui.java.....	240
10.25.2. guiAlta.java.....	259
10.25.3. guiAviso.java.....	268
10.25.4. guiBuscar.java.....	270
10.25.5. guiBuscarLog.java.....	277
10.25.6. guiFile.java.....	278
10.25.7. guiGrafico.java.....	280
10.25.8. guiResultado.java.....	284
10.25.9. guiTexto.java	288
11. Conclusiones y Valoración Personal.....	291
12. BIBLIOGRAFIA.....	292

1. INTRODUCCIÓN

Los trasplantes de órganos es uno de los campos de la medicina moderna que conlleva una gran complejidad y dificultad, las cuales nos encontramos desde el momento en que el órgano es extraído del donante hasta que se transplanta al receptor adecuado. En todo este proceso intervienen diversos factores tales como la difícil elección del paciente más adecuado para recibir el órgano, la elección y coordinación de un equipo médico para realizar el trasplante o la forma de transportar el órgano del punto donde se encuentra hasta el lugar donde es necesario en el menor tiempo posible, y si puede ser, con el menor coste.

El grupo de investigación en Inteligencia Artificial de la Universidad Rovira i Virgili denominado Banzai [Banzai], ha abierto una línea de investigación en este ámbito, en colaboración con un hospital. El objetivo es construir un Sistema Multi-Agente de ayuda al proceso de coordinación de trasplantes de órganos. Así, dentro del grupo de alumnos GruSMA [GruSMA] creado por Aïda Valls y Toni Moreno, se está desarrollando un prototipo basado en agentes a partir de varios PFCs que se especializan en partes concretas del proceso de coordinación de trasplantes [Bocio,2000], [Ribes,2000], [Sánchez,2001], [Bocio,2002].

El proceso que se ha de realizar para llevar a cabo un trasplante, requiere de la colaboración de personal de diversos hospitales con el fin de encontrar rápidamente un receptor óptimo y coordinar el transporte y equipos. En un ámbito tan crítico, donde el tiempo es un factor fundamental, que el sistema de gestión esté bien estructurado para una respuesta rápida y precisa, resulta fundamental. En nuestro país, la Organización Nacional de Trasplantes [ONT], establece un sistema que está demostrando ser uno de los mas eficaces, conocido como “El modelo Español”: el territorio nacional está dividido en 6 zonas; cada una agrupa alguna de las comunidades autónomas que por su banda están organizadas en provincias, localidades y, finalmente hospitales. En cada hospital existirá una persona (Coordinador de Trasplantes) encargada de gestionar el proceso de asignación de órganos a los pacientes de su centro y comunicarse con sus homólogos dentro del mismo nivel jerárquico. Si no se encuentra un receptor adecuado, se pasará a un nivel superior de la jerarquía (ver **figura 1**) siguiendo el esquema: hospital -> ciudad -> provincia -> zona -> país.

Como consecuencia del corto período de duración en buen estado de los órganos donados, es fundamental un buen y efectivo contacto entre Coordinadores, que actualmente se mantiene de forma tradicional mediante (teléfono y fax) método que consecuentemente produce una importante pérdida de tiempo, para resolver éste problema se propone utilizar la metodología de Sistemas Multi-Agentes, que como veremos mas adelante permitirá trabajar de forma distribuida a cada uno de los centros involucrados y comunicarse eficientemente mediante una red global con su consecuente reducción de coste y tiempo. [Aldea et al., 2001]. El sistema seguirá el modelo de organización definido por la ONT, así como las leyes existentes sobre la gestión de trasplantes.

Por otro lado, durante el proceso de búsqueda del receptor ideal del órgano, será necesario evaluar los diversos pacientes disponibles y decidir quien o quienes resultan

los más adecuados. En este punto, algunas técnicas de inteligencia artificial pueden servir de ayuda, de forma que la elección se realice de forma mas eficiente [Valls et al., 2001].

En este proyecto se pretende desarrollar un conjunto de agentes que se dedicarán a analizar la lista de pacientes que son posibles receptores de un órgano y de ayudar al Coordinador, proporcionándole una lista ordenada según el grado de idoneidad de cada paciente con respecto al órgano disponible.

El objetivo de este programa informático es construir una herramienta que ayude al Especialista a tomar una decisión más rápida y fácil, dado que la principal dificultad que se encuentra en el proceso de evaluación de receptores, es la comparación entre pacientes y donantes por la considerable cantidad de factores que intervienen (características físicas, tiempo de espera, distancia...) que tendrán un peso mas o menos importante en la decisión final. Para conseguir dicho objetivo se implementarán métodos de ayuda a la toma de decisiones multicriterio (MCDM) [Vincke, 1992]. Existen varios métodos MCDM que nos pueden ayudar a resolver el problema de forma que el sistema consistirá en un conjunto de agentes donde cada uno de ellos implementará uno de estos métodos. Los métodos elegidos utilizan criterios cualitativos, cosa que nos resulta especialmente útil, ya que muchos de los factores que intervienen en el proceso de elección de un receptor no pueden expresarse de forma exacta, ya sea porque resulta muy difícil o porque no es necesaria tanta precisión.

El hecho de usar términos cualitativos, nos acarrea el problema de establecer una equivalencia numérica para poder manejar los datos con el método elegido, de forma que necesitamos alguna herramienta para poder expresar numéricamente cada término independientemente de la persona que lo use. En este proyecto se usan tres semánticas distintas, las funciones de negación [Torra, 1994], los conjuntos difusos [Zadeh, 1968] y las 2-tuplas [Herrera&Martínez, 2000a]. Para cada tipo de interpretación de los términos de los vocabularios tendremos un agente que aplica un MCDM basado en cada tipo de representación: ClusDM para funciones de negación (agente desarrollado por David Sánchez de GruSMA [Sánchez, 2001], LOWA para conjuntos difusos (agente que se basa en el método definido por [Herrera&Martínez, 2000b]) y 2-tupleDM para semánticas de tuplas (agente que se basa en los operadores definidos por [Herrera et al., 2000]).

El sistema no solo constará de los distintos agentes que implementan los métodos MCDM sino que en cada sistema entremos un *agente Especialista en transplantes*, el cual poseerá información sobre la reglas y restricciones que un paciente debe cumplir para un transplante determinado eliminando los que no valgan, además pasará las órdenes y datos a los diferentes agentes MCDM de su sistema recogiendo los resultados de estos. Otro agente importante es el *agente Coordinador de transplantes*, este agente es el encargado de interactuar con el usuario y de enviar las órdenes y datos a los distintos agentes especialistas de los distintos sistemas que estén conectados, además recibirá los resultados mostrándoselos al usuario.

Las ideas desarrolladas en este proyecto al igual que en el de David Sánchez, fueron presentadas en la sesión de estudiantes del *European Agent System Summer School* (EASSS) organizada en Praga por la red europea AgentLink (asociación a la que pertenece la URV) y la asociación europea de Inteligencia Artificial ECAI (de la cual el

grupo Banzai forma parte) que se celebraba dentro del *Advanced Course on Artificial Intelligence* [Valls et al., 2001].

2. AGENTES Y SISTEMAS MULTIAGENTE (SMA)

Se podría describir como un sistema que habita en un entorno complejo y dinámico; es capaz de sentir y actuar autónomamente en su entorno y tiene un conjunto de objetivos o motivaciones que intentará conseguir a través de sus acciones [Weiss, 1999]. A veces se define un agente a partir de sus propiedades, las cuales detallaremos a continuación.

2.1. Propiedades de los agentes

Las propiedades más importantes que pueden tener los agentes son las siguientes:

- Autonomía: es la capacidad de operación sin intervención directa de los humanos u otros, y tener algún tipo de control sobre las propias acciones y el estado interno.
- Sociabilidad/cooperación: los agentes han de ser capaces de interactuar con otros agentes y/o humanos a través de algún tipo de lenguaje de comunicación.
- Reactividad: los agentes perciben su entorno y responden en un tiempo razonable a los cambios detectados.
- Pro-actividad o capacidad de tomar iniciativa: han de ser capaces de exhibir algún tipo de comportamiento encaminado a llevar a cabo algún tipo de objetivo tomando la iniciativa
- Movilidad: posibilidad de moverse en otros entornos a través de una red electrónica.
- Continuidad temporal: los agentes están continuamente ejecutando procesos.
- Veracidad: Asumir que un agente no comunicará información falsa premeditadamente.
- Benevolencia: Asumir que los agentes no tienen objetivos conflictivos, y que cada agente intentará hacer aquello para lo que ha sido requerido.
- Racionalidad: el agente ha de actuar con la finalidad de conseguir su objetivo.

2.2. Arquitecturas de agentes

Las arquitecturas utilizadas en el proceso de implementación de los agentes pueden ser de tres tipos:

- Deliberativas: dada una representación del mundo real, se pueden ejecutar acciones inteligentes. Sigue el modelo BDI (Beliefs Desires and Intentions).
- Reactivos: estructura donde las acciones se llevan a cabo respondiendo a estímulos. La combinación de diversos agentes reactivos proporciona un comportamiento “inteligente”.
- Híbridas: estructuras que barajan diferentes técnicas.

2.3. Tipos de agentes

Según la finalidad con la que se implemente un agente, podemos clasificarlo en diferentes tipos:

- De información: gestionan y manipulan datos; pueden responder a requerimientos de los usuarios u otros agentes. Un ejemplo muy común son los buscadores de información en Internet.
- De interficie: sistema donde los agentes colaboran con un usuario para resolver un problema. La interacción con el individuo permite al agente desarrollar un aprendizaje basado en las acciones que se desarrollen.
- De colaboración: la característica principal es la comunicación y cooperación con otros agentes para resolver un problema común. Utilizan técnicas de IA distribuida.
- Móviles: su característica principal es la posibilidad de poder moverse por una red electrónica recogiendo información o interactuando con otros hosts.
- Reactivos: su comportamiento se basa en la respuesta a estímulos según un patrón del estado actual en que se encuentren. Son muy simples, pero la combinación de diversos agentes reactivos pueden generar comportamientos complejos.
- Híbridos: son combinaciones entre los diversos tipos de agentes anteriormente citados intentando reunir las ventajas de unos y otros.
- Heterogéneos: se refieren a un conjunto integrado de al menos dos agentes que pertenecen a dos o más clases diferentes.

2.4. Sistemas Multiagente (SMA)

Un sistema Multiagente será aquel en el que un conjunto de agentes cooperen, se coordinen y se comuniquen con el fin de conseguir un objetivo común.

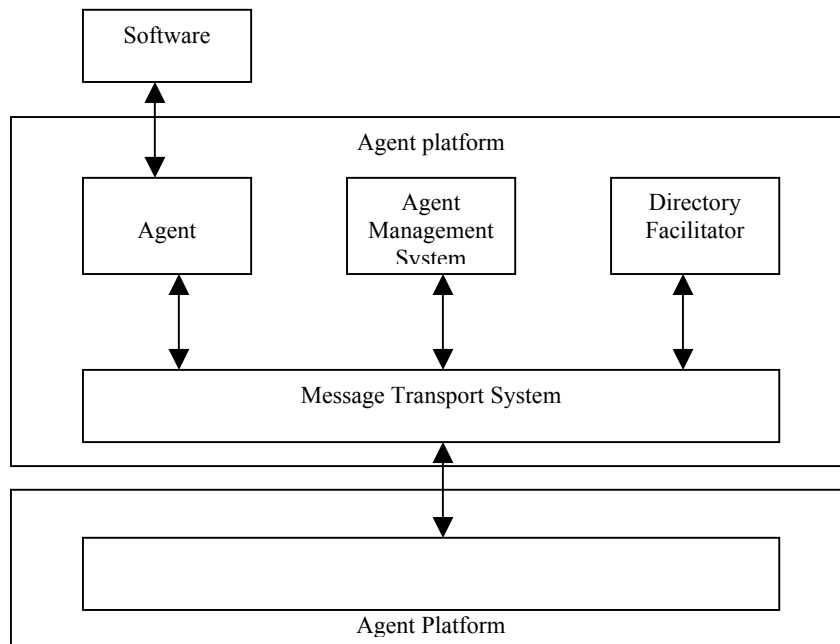
2.4.1. Ventajas de un SMA

Las principales ventajas de la utilización de un sistema Multiagente en la gestión de un problema son:

- Modularidad: se reduce la complejidad en trabajar con unidades mas pequeñas, permite una programación mas estructurada.
- Eficiencia: la programación distribuida permite repartir los trabajos entre los agentes, consiguiendo paralelismo (agentes trabajando en diferentes máquinas).
- Fiabilidad: el hecho que un elemento del sistema deje de funcionar no significa que el resto también lo hagan, además, se puede conseguir mas seguridad replicando servicios críticos y así conseguir redundancia.
- Flexibilidad: se pueden añadir y eliminar agentes dinámicamente.

2.4.2. Gestión de un SMA

La administración de agentes establece el modelo lógico para la creación, registro, comunicación, establecimiento, movilidad y destrucción de agentes. La FIPA (Foundation for Intelligent Physical Agents, organización que promueve el desarrollo de aplicaciones basadas en agentes definiendo estándares [FIPA]), establece el siguiente modelo para este entorno de administración de agentes [FIPA, 2000b]:



Los componentes lógicos principales que forman parte del sistema de la **figura 2** son:

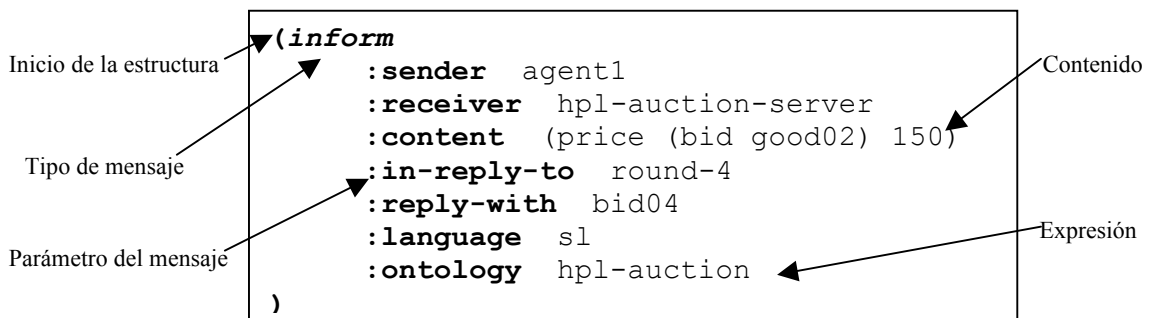
- **Agent:** unidad básica. Se puede describir como un programa que encapsula una serie de servicios.
- **Directory Facilitator (DF):** es un agente que proporciona un servicio de Páginas Amarillas dentro del sistema (conoce los servicios que pueden ofrecer los diferentes agentes del SMA). Los agentes han de registrarse en el DF para ofrecer sus servicios.
- **Agent Management System (AMS):** es el agente que controla el acceso y el uso de la plataforma. Almacena las direcciones de los agentes, ofreciendo un servicio de Páginas Blancas.
- **Message Transport System (MTS):** facilita la comunicación entre agentes de diferentes plataformas.
- **Agent Platform (AP):** proporciona la infraestructura básica en el que pueden crearse y ejecutarse los agentes.
- **Software:** cualquier programa accesible desde un agente.

2.4.3. Lenguaje de comunicación entre agentes (ACL)

Los agentes individualmente proporcionan una funcionalidad interesante, pero lo que los hace tan adecuados para determinados sistemas es su capacidad de cooperar para resolver problemas. Para poder hacerlo, los agentes han de comunicarse utilizando un lenguaje común: ACL (Agent Communication Language).

Los ACL están basados en la “Teoría del habla” en la que las comunicaciones individuales entre agentes pueden reducirse a un pequeño número de ideas o, mas generalmente, actos comunicativos. Estos actos dan forma al significado básico de la comunicación.

El elemento básico de la comunicación es el mensaje; la FIPA determina que forma ha de tener un mensaje y su utilización [FIPA, 2000a]. En la **figura 3** observamos los componentes básicos que ha de tener un mensaje.



2.4.3.1. Elementos de un mensaje

Los elementos que de forma estándar forman parte de un mensaje de comunicación entre agentes son:

- Definición de tipos de mensajes (performative): indica el tipo de acto de comunicación. Las acciones mas habituales son:
 - **accept-proposal**: aceptación de una propuesta hecha anteriormente para realizar una acción.
 - **agree**: aceptación de una acción.
 - **cancel**: cancelación de una acción pedida anteriormente.
 - **cfp**: llamada para proponer la realización de una acción.
 - **confirm**: el emisor informa al receptor que una proposición es cierta, cuando el receptor dudaba sobre la proposición.
 - **disconfirm**: el emisor informa al receptor que una proposición es falsa, cuando el receptor creía que era cierta.
 - **failure**: indica que la acción pedida anteriormente ha fallado por algún motivo.

- **inform**: el emisor informa al receptor que una acción se ha realizado correctamente.
 - **not-understood**: el emisor informa al receptor que no ha entendido una petición realizada anteriormente.
 - **propose**: acción de proponer la realización de una acción, dadas ciertas precondiciones.
 - **query-if**: acción de preguntar a otro agente si un hecho es cierto o no.
 - **refuse**: negación a la realización de una acción dada.
 - **request**: el emisor pide al receptor realizar una acción.
 - **subscribe**: el emisor pide ser avisado en el momento que se cumpla una condición.
- Interlocutores: son los identificadores de agentes. Hay 3: **sender** (quien envía el mensaje), **receiver** (quien lo recibe), **reply-to**, (a quien ha de ir destinado el siguiente mensaje de la conversación).
- Contenido del mensaje: existen cuatro tipos de contenidos predefinidos que se usan en función de las necesidades que se tengan:
- **FIPA-CCL (Constraint Choice Language)**: define una semántica que permite especificar predicados con satisfacción de restricciones (CSP).
 - **FIPA-SL (Semantic Language)**: permite formar expresiones lógicas, intercambiar conocimiento de forma óptima y expresar acciones a realizar. Es el lenguaje mas general de todos y se puede aplicar diferentes dominios. En este proyecto es el que se utiliza
 - **FIPA-KIF (Knowledge Interchange Format)**: permite expresar objetos como términos y proposiciones como sentencias.
 - **FIPA-RDF (Resource Description Framework)**: permite representar objetos, interacciones y acciones entre ellos.
- Descripción del contenido: permite que los agentes que reciben el mensaje puedan identificar que se le esta enviando y en que formato. Se definen tres descriptores:
- **language**: en que lenguaje está escrito el contenido, en nuestro caso usamos el FIPA-SL.
 - **encondig**: indica si se utiliza algún tipo de codificación especial.
 - **ontology**: es el campo que define la ontología utilizada para dar significado al contenido del mensaje, se considera el campo más importante.
- Control de la conversación: esta destinado a identificar que tipo de conversación se mantiene. Sus campos informativos son:
- **protocol**: protocolo usado en el mensaje.
 - **coversation-id**: identificador de la conversación.
 - **reply-with**: identificador del mensaje que se usara para seguir los diferentes pasos de la conversación.
 - **in-reply-to**: identifica una acción pasada a la cual responde este mensaje.
 - **reply-by**: fecha/hora de caducidad del mensaje (timeout).

2.4.3.2. Protocolos de comunicación

En el apartado anterior, hemos mencionado que algunos campos de control de las conversaciones identifican el tipo de protocolo que se utiliza, dicho protocolo es que define una serie de reglas que se han de seguir para mantener una conversación. Hay diferentes tipos dependiendo de su finalidad. Estos protocolos y su finalidad son definidos por la FIPA.

- *FIPA Request*: permite pedir la realización de acciones y devolver los resultados.
- *FIPA Query*: se utiliza para realizar preguntas
- *FIPA Contract Net*: se utiliza para negociaciones, proponer y evaluar propuestas.
- *FIPA Iterated Contract Net*: es una variante del protocolo anterior que permite la renegociación.
- *FIPA Brokering*: gestiona los servicios disponibles por los agentes.
- *FIPA Recruiting*: variante del protocolo anterior.
- *FIPA Subscribe*: permite la notificación de sucesos importantes.
- *FIPA Propose*: simplificación del Contract Net.

3. JADE

El *Java Agent Development Enviroment* o JADE, es una herramienta de programación la cual esta compuesta por diferentes librerías escritas en código Java útiles para el desarrollo de Sistemas Multiagente. Nos proporciona una serie de funciones e interfaces que facilitan la programación de los agentes, además de ofrecernos un entorno en el cual se pueden ejecutar los distintos agentes e interactuar entre ellos. La versión utilizada en este proyecto es la 2.01.

3.1 Paquetes de JADE

Los principales paquetes o librerías que JADE nos proporciona son:

- **jade.core**: es el paquete principal, el cual nos proporciona la clase fundamental para un sistema mutiagente *jade.core.Agent* al igual que los diferentes comportamientos que puede tener cada agente *jade.core.behaviour*.
- **jade.domain**: paquete que contiene todas las clases necesarias para representar la plataforma de agentes y los distintos modelos del dominio tales y como entidades para la administración de agentes, lenguajes y ontologías.
- **jade.lang**: paquete que contiene las clases necesarias para soportar un lenguaje de comunicación entre agentes. Los principales subpaquetes son *jade.lang.acl* y *jade.lang.sl*.
- **jade.proto**: paquete que contiene los protocolos de interacción entre agentes FIPA.
- **jade.tools**: paquete que contiene alguna herramientas que nos facilitan el desarrollo de aplicaciones y administración tales como:
 - **Dummy Agent**: herramienta de depuración y monitorización compuestas por un agente y una interfaz gráfica. Nos permite crear mensajes ACL y enviarlos a otros agentes visualizando los resultados.
 - **Remote Management**: herramienta que actúa como una consola gráfica para la administración y control del sistema.
 - **Sniffer**: herramienta compuesta por un agente que intercepta mensajes ACL y los visualiza.
 - **SocketProxyAgent**: herramienta compuesta por un agente que actúa como una puerta de enlace bidireccional entre la plataforma JADE y una conexión TPC/IP.

3.2 Principales componentes de JADE

Según la metodología de JADE, los agentes residen en un entorno que nos predefine el sistema denominado plataforma. Cada plataforma está dividida en contenedores y cada uno de estos contendrá agentes. Los contenedores se pueden entender como dominios de agentes.

Para poner en marcha el sistema, la plataforma que arrancará en una máquina determinada mientras que los distintos agentes se podrán ejecutar en diferentes máquinas. La plataforma tiene que tener como mínimo dos agentes que tienen una función vital para el funcionamiento del sistema [FIPA,1998b]:

- **Domain Facilitator (DF):** se asemeja a un servicio de páginas amarillas, contiene todos los servicios que proporciona cada agente que esta registrado.
- **Agent Management System (AMS):** éste se asemeja a un servicio de páginas blancas, es decir, registra las direcciones de los agentes que se han dado de alta.

Gracias a estos servicios, se proporciona mucha flexibilidad y transparencia, por lo que si un agente cualquiera quiere enviar un mensaje a otro que realice una tarea en concreto, este agente pregunta al DF que agentes realizan esta tarea, tras esto preguntará al AMS que dirección tienen estos agentes. Para que el sistema funcione correctamente, cuando un agente se inicie tendrá que informar al DF sobre los servicios que proporciona y que tipo de agente es, de manera que el AMS le proporcionará una dirección física para que otros agentes se puedan poner en contacto con el.

3.3 Agentes y sus diferentes Comportamientos

Para poder crear agentes en JADE, tenemos que extender la clase, ya antes mencionada, *jade.core.Agent* la cual ya se encuentra definida de forma que nosotros tendremos que rellenar los métodos y campos básicos que nos especifica la interfaz. Los dos métodos más relevantes son *setup()* que se ejecuta al iniciarse el agente y *takedown()* que se ejecuta cuando finaliza la ejecución del agente.

Una vez que hemos creado el agente, hemos de darle funcionalidad lo cual se consigue mediante la definición de comportamientos (*behaviours*) que controlarán sus acciones. Un scheduler interno será el encargado de ir controlando automáticamente la ejecución de todos los comportamientos. Para todo esto JADE nos proporciona un conjunto de comportamientos básicos que permiten implementar diferentes protocolos de comunicación. Dentro del grupo de comportamientos simples nos encontramos con los siguientes:

- **ReceiverBehaviour:** permite arrancar un comportamiento que espera la recepción de un mensaje.

- **SenderBehaviour**: permite arrancar un comportamiento para enviar un mensaje.
- **SimpleBehaviour**: comportamiento *atómico*, que se ejecuta sin interrupciones el cual se puede ejecutar una sola vez (**OneShotBehaviour**) o cíclicamente (**CyclicBehaviour**).

También existen comportamientos complejos, que se pueden crear hijos y atender diversas peticiones a la vez. Estos se agrupan dentro de la clase abstracta **ComplexBehaviour** la cual se divide en dos subtipos:

- **SequentialBehaviour**: ejecuta todos sus hijos de forma secuencial acabando cuando todos finalicen correctamente.
- **NonDeterministicBehaviour**: ejecuta sus hijos según una secuencia no determinista.

3.4 Mensajes entre Agentes

En lo referente a la comunicación, en JADE se siguen las especificaciones de la FIPA en cuanto al formato de los mensajes [FIPA,2000a], de forma que un mensaje estará dividido en *slots* los cuales nos dan una información (protocolo, lenguaje, contenido ...) y sobre estos se definen funciones de creación/modificación (*set*) y consulta (*get*).

Los métodos básicos de los cuales disponemos para enviar mensajes entre agentes son:

- **ACLMessage receive()**: recibe un mensaje de la cola del agente, además de poderse especificar que tipo de mensaje se quiere recibir mediante los patrones de búsqueda (*MessageTemplate*).
- **Send(ACLMessage)**: envía el mensaje que le pasamos por parámetros una vez cubiertos todos sus campos.

En el proyecto se usa el lenguaje que se usa para escribir los mensajes es el FIPA-SL (Semantic Language) el cual se caracteriza por tener un contenido ASCII, un formato semejante al Lisp, y al igual que el resto de lenguajes, un ámbito más general permitiendo formar expresiones lógicas, intercambiar información entre agentes y expresar acciones a realizar. La notación utilizada es muy semejante a la de Lisp.

3.5 Ontología

Para enviar un mensaje, uno de los principales campos que tendremos que definir, es la ontología, la cual usaremos para escribir el contenido del mensaje y acceder a él. Define el vocabulario empleado. Definir una ontología adaptada al problema que queremos tratar, es uno de los elementos más importantes del sistema.

Una ontología nos permite definir los siguientes puntos:

Frames: objetos sobre los que se podrá hablar los cuales estarán disponibles para rellenar el mensaje. Los frames se componen de slots. Los slots describen propiedades del objeto y pueden ser de tipo simple o ser a su vez frames, para representar datos complejos.

- **Acciones:** sirven para definir los servicios que proporcionan los agentes del sistema. De ésta forma todos los agentes conocen el nombre por el que deben invocar a estos servicios.
- **Predicados:** sirven para definir las preguntas que se podrán hacer los agentes entre sí.

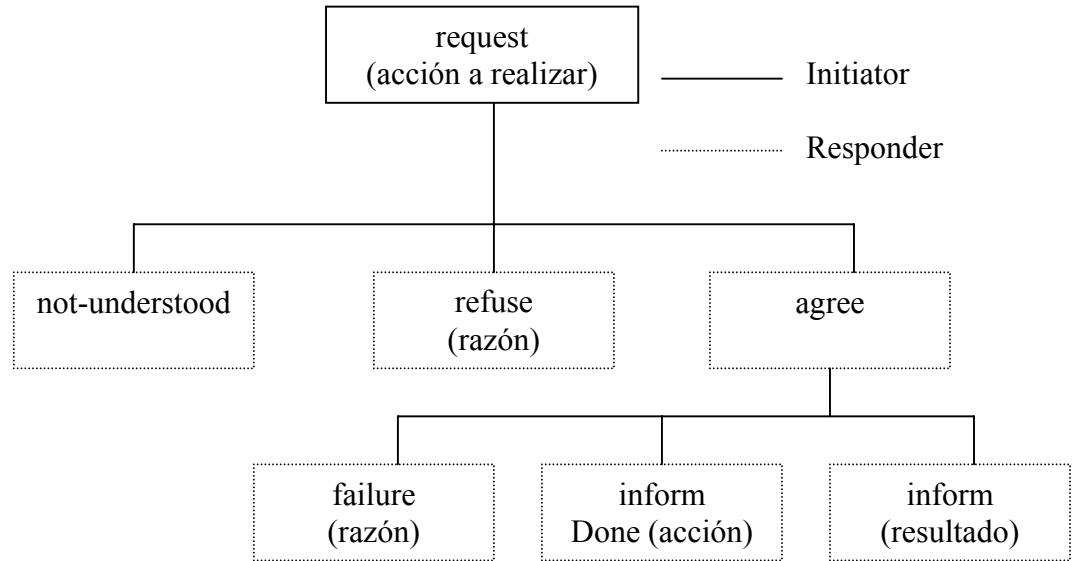
Además de implementar la ontología en un módulo a parte, también será necesario definir e implementar una clase de que se encargue de manejar cada objeto, acción o predicado creado. Esta clase contendrá métodos *get* y *set* para cada **slot** que hayamos definido, los cuales serán los que nos permitan consultar y rellenar los datos del mensaje.

3.6 Protocolos de Comunicación

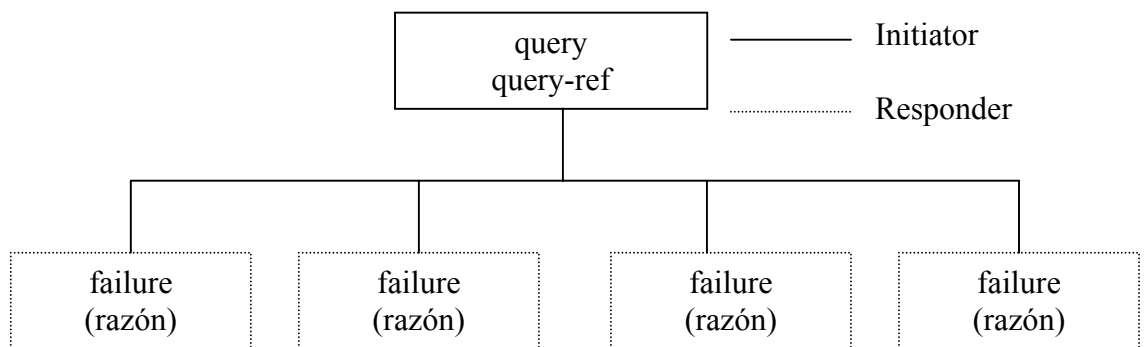
Para enviar un mensaje también será necesario escoger un protocolo de comunicación entre los agentes, el cual se adapte a nuestras necesidades. JADE nos proporciona los elementos necesarios para definir los protocolos descritos por la FIPA [FIPA,1998b]. Para cada conversación distingue dos roles, el que la inicia o *initiator* y el que responde o *responder* de forma que cada agente tendrá que extender los comportamientos según su papel.

Los distintos tipos de protocolos son:

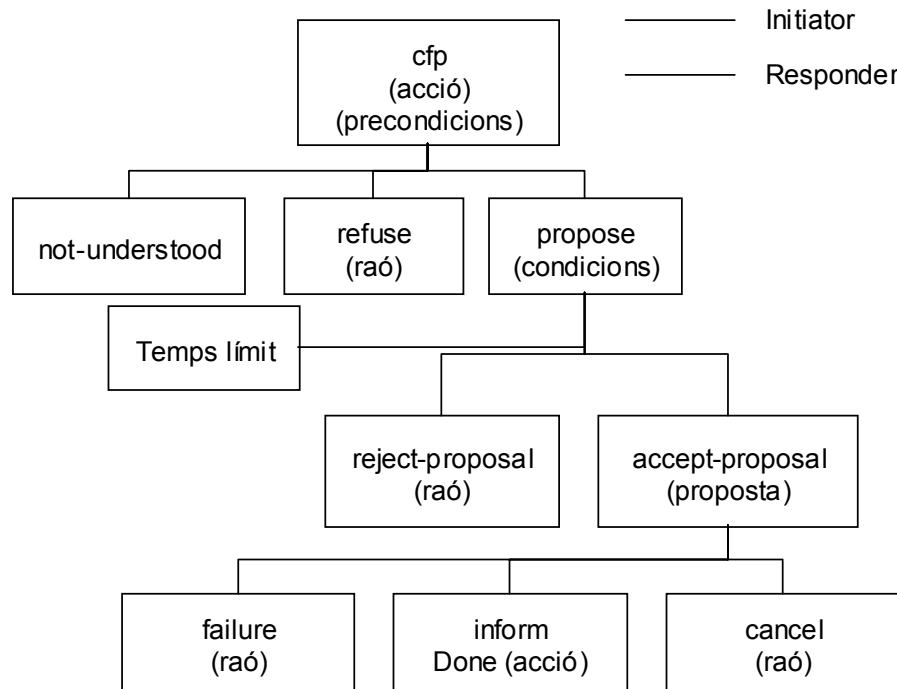
- **FIPA Request:** permite pedir la realización de acciones y devolver los resultados.



- **FIPA Query:** se utiliza para realizar preguntas.



- **FIPA Contract Net:** se utiliza para negociaciones, proponer y evaluar propuestas.



Como se ha dicho antes, para implementar estos protocolos JADE los divide en dos partes la que inicia y la que responde, de manera que cada una de estas partes es una clase abstracta de nombre **xxxInitiatorBehaviour** y **xxxResponderBehaviour**, donde xxx varia según el protocolo utilizado. Para cada comportamiento es programador solo tiene que rellenar los métodos que manejen los diferentes estados por los cuales pasa la conversación.

En nuestro caso hemos usado un protocolo de tipo FIPA Request, por lo cual hemos necesitado las siguientes clases:

- **FIPARequestInitiatorBehaviour**: esta clase esta implementada por el agente *initiator* que comienza el envío del **request** controlando las posibles respuestas. En este protocolo se considera una primera respuesta en la que el agente informa si puede o no realizar la acción de forma que si la respuesta es afirmativa, se esperará una segunda con el resultado.
- **FIPARequestResponderBehaviour**: esta clase la implementa el agente *responder*, el cual espera recibir un **request** para evaluar la acción: si la puede realizar envía una confirmación y comienza a hacerla enviando más tarde el resultado; en caso contrario la rechaza.

3.7 Herramientas de JADE

Disponemos de una interfaz gráfica para la administración de la plataforma en la que se ejecutan nuestros agentes, así como de herramientas que nos facilitan la depuración y testeo de las aplicaciones basadas en ellos.

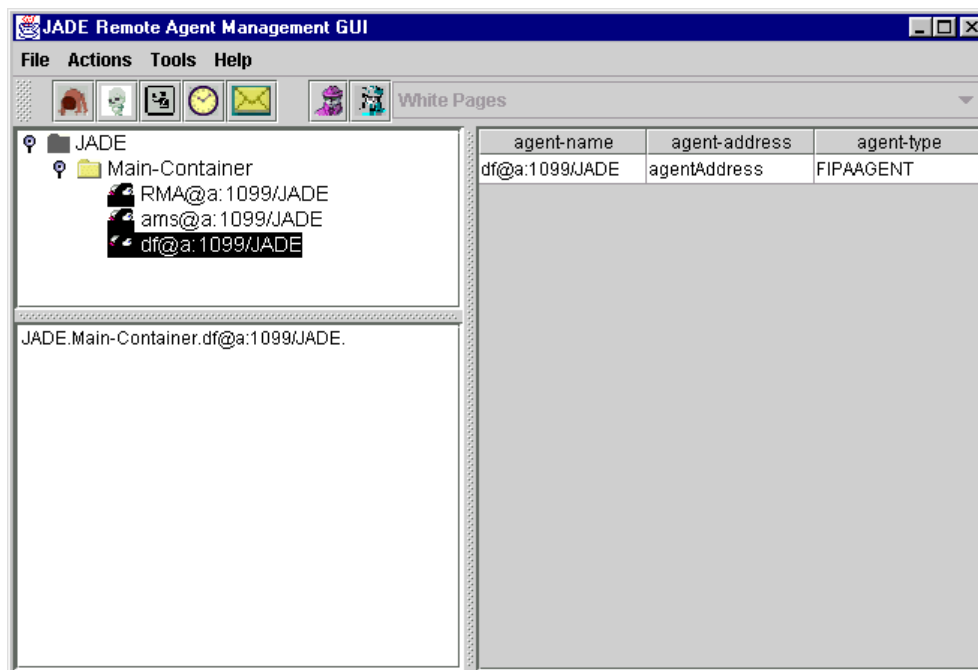
Para iniciar una sesión en JADE y crear los distintos agentes, entremos que ejecutar el siguiente comando (recordar tener la variable CLASSPATH con los paquetes necesarios):

```
java jade.Boot -platform [opciones] [lista de agentes]
```

Dentro de las opciones podemos indicar que se arranque el entorno gráfico, para lo cual añadiremos la opción `-gui`. La lista de agentes será una cadena de texto donde cada uno de ellos estará separado por espacios en blanco del resto manteniendo el siguiente formato: nombre del agente dos puntos y la clase java que lo implementa.

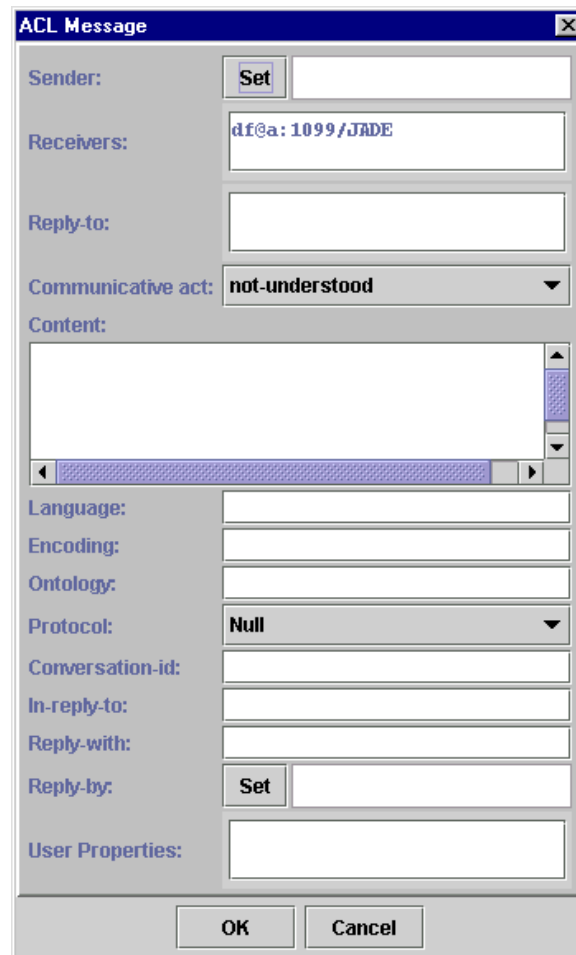
Al arrancarse el entorno gráfico y la plataforma, se crean una serie de agentes aunque no especifiquemos ninguno propio, estos agentes son:

- El agente RMA encargado de controlar la interfaz.
- El agente DF, donde se suscribirán los servicios de los agentes.
- El agente AMS donde se guardaran las direcciones de los agentes.



Además de la lista de agentes que hay en el contendor y la información sobre el que hemos seleccionado, disponemos de una serie de opciones que nos facilitarán el testeo y manejo de nuestro sistema:

- Añadir un nuevo agente
- Enviar un mensaje a los agentes seleccionados

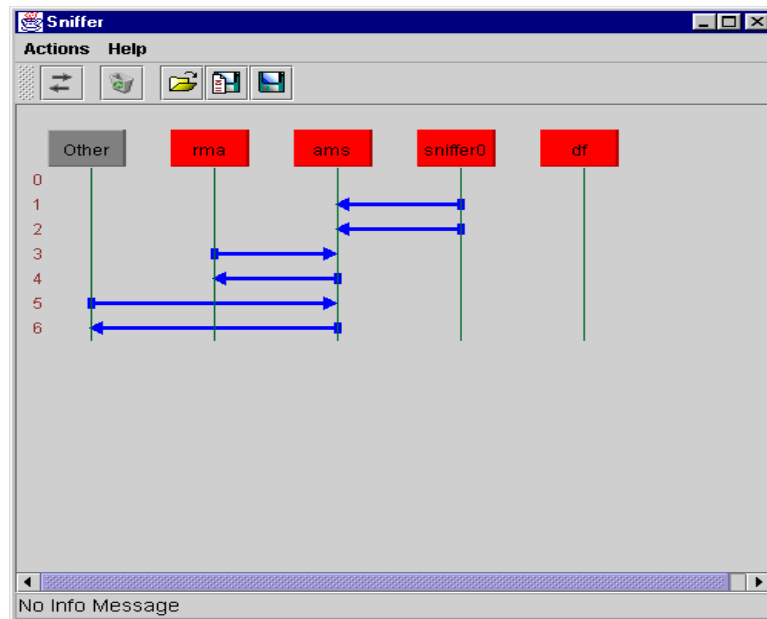


The image shows a dialog box titled "ACL Message". It contains several fields for configuring a message:

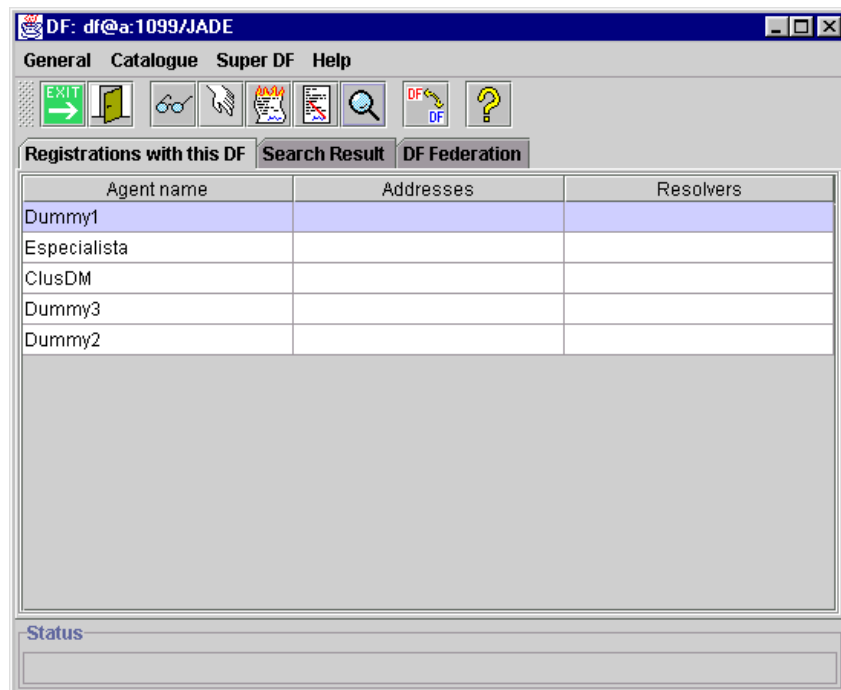
- Sender:** A button labeled "Set" next to an empty text box.
- Receivers:** A text box containing the value "df@a:1099/JADE".
- Reply-to:** An empty text box.
- Communicative act:** A dropdown menu currently showing "not-understood".
- Content:** A large empty text area with a scrollbar.
- Language:** An empty text box.
- Encoding:** An empty text box.
- Ontology:** An empty text box.
- Protocol:** A dropdown menu currently showing "Null".
- Conversation-id:** An empty text box.
- In-reply-to:** An empty text box.
- Reply-with:** An empty text box.
- Reply-by:** A button labeled "Set" next to an empty text box.
- User Properties:** An empty text box.

At the bottom of the dialog box are two buttons: "OK" and "Cancel".

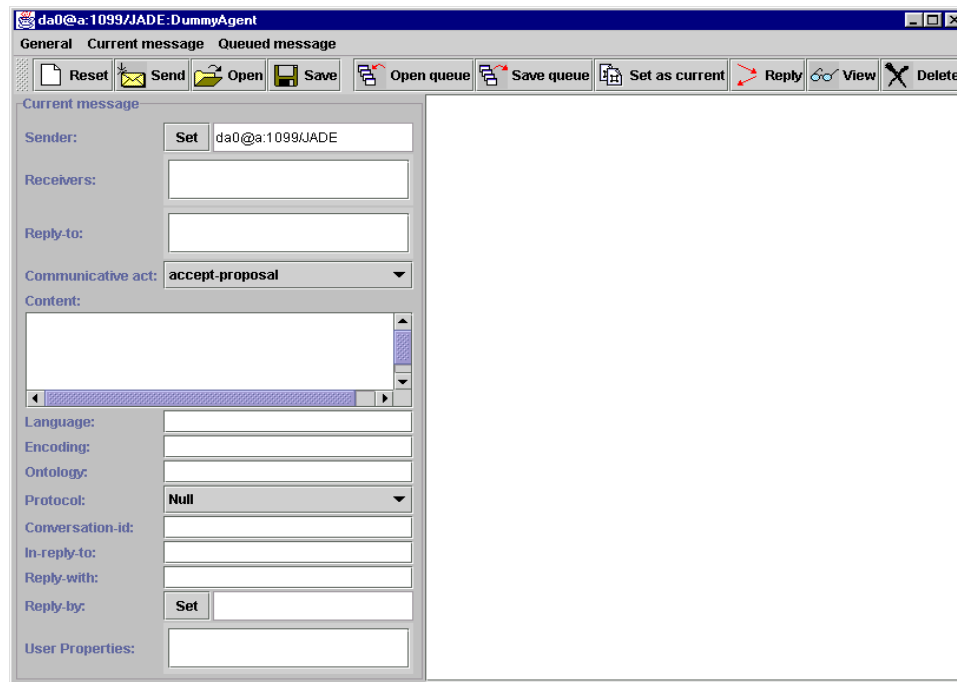
- Parar los agentes seleccionados
- Eliminar los agentes seleccionados
- Ejecutar los agentes seleccionados
- Continuar los agentes seleccionados
- Ejecutar un agente *sniffer*: aplicación JAVA creada para rastrear el intercambio de mensajes entre los agentes de JADE. Al activarlo, se nos abrirá una ventana como la que se muestra a continuación, en la cual podremos observar la evolución del paso de mensajes a lo largo del tiempo. Seleccionando laguna de las flechas se mostrará el mensaje con todos sus parámetros.



- Activar la interface del agente DF: Al realizar esta acción se abrirá una ventana como la que se muestra a continuación, en la podremos observar los agentes que se han registrado, así como que tipo de servicio ofrecen.



- Abrir un *Dummy-Agent*: Este agente permite enviar mensajes ACL entre los agentes, así como recibirlos e inspeccionarlos para leerlos y si queremos guardarlos en un fichero.



4. ARQUITECTURA DEL SISTEMA

El sistema es una parte de un sistema Multi-Agente que gestiona el transplante de órganos. En concreto nos centraremos en la elección de los receptores mas adecuados para un determinado donante. El sistema que presentamos aquí forma parte de un sistema mayor para gestionar la búsqueda y asignación de órganos a nivel español. A continuación, en la figura X, se muestra un diagrama de la estructura jerárquica del sistema multi-agente de búsqueda de receptores adecuados. En la figura XX, se puede ver el sistema-multiagente que funcionaría dentro de cada hospital [Aldea et al., 2001].

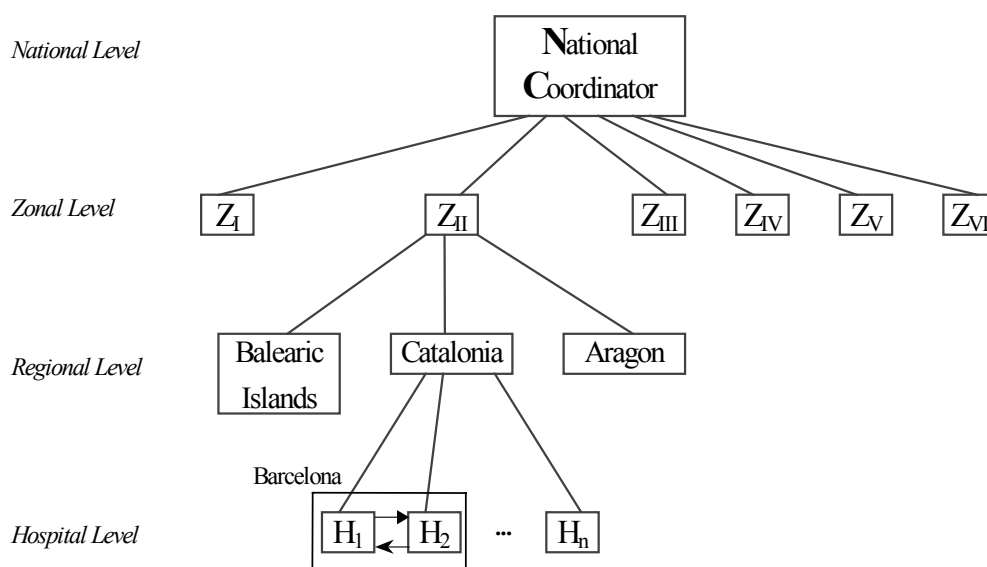


Figura X. Arquitectura del sistema multiagente de búsqueda de receptores

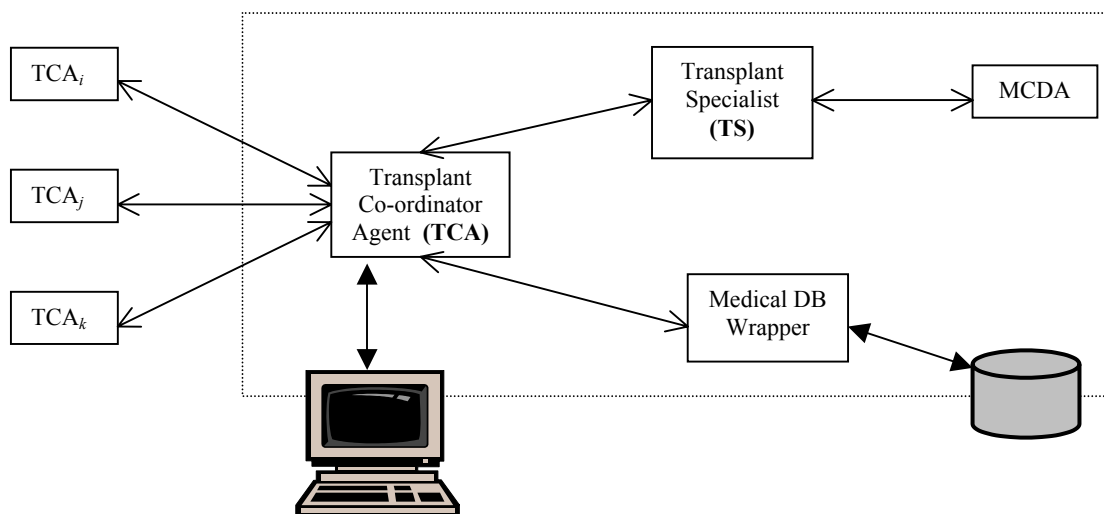


Figura XX. Arquitectura del sistema multiagente de un hospital

Nosotros nos hemos centrado en la parte interna del hospital. Como ya se ha dicho, el propósito de este proyecto es estudiar diferentes agentes MCDM y comparar su comportamiento.

En este proyecto se ha implementado el agente Coordinador y se ha modificado el funcionamiento del agente Especialista para que pueda recibir mensajes del coordinador así como enviar resultados a éste. Los agente MCDM implementados son dos, uno que usa el método basado en la semántica *2-tuple* [Herrera y Martínez, 2000] y otro basado en la semántica *Fuzzy sets* [Zadeh, 1968], además el sistema trabaja con otro agente MCDM que implementa un método basado en las Funciones de Negación y Clustering implementado en el PFC de David Sánchez.

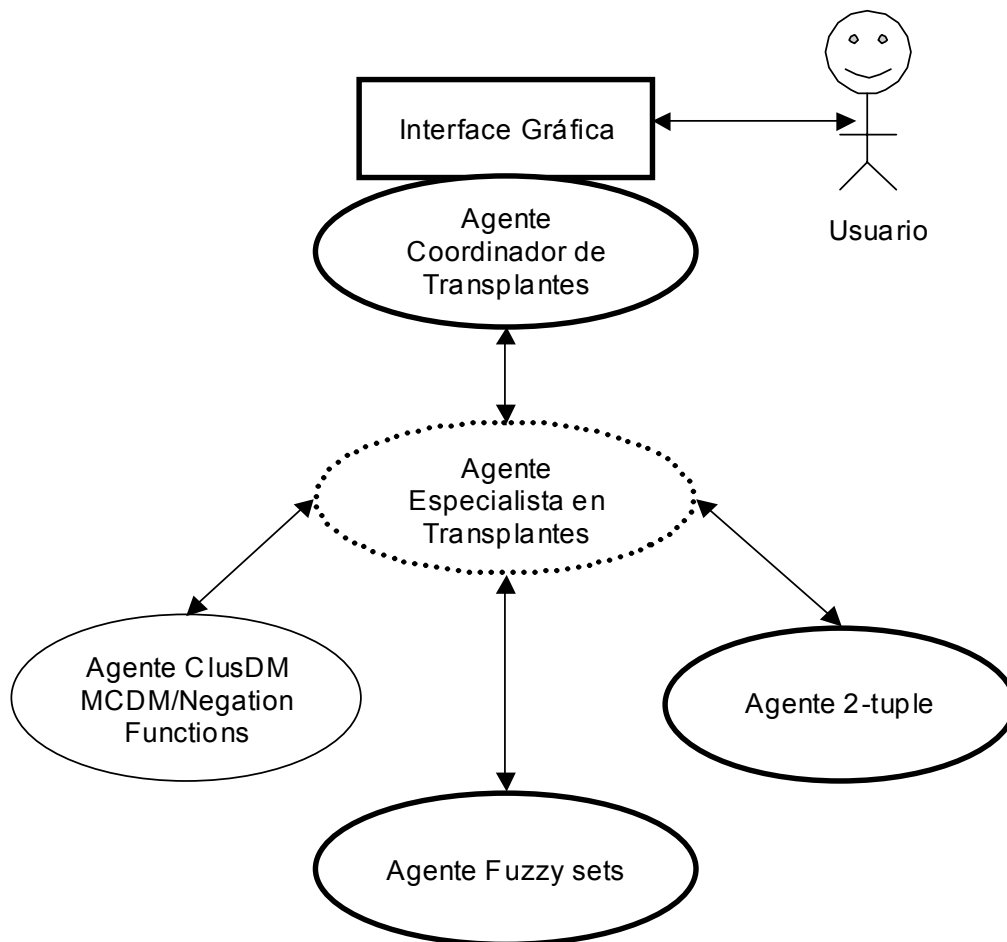


Figura XXX. Arquitectura del sistema multiagente implementado

Como podemos observar en la figura XXX, la estructura esta formada por tres niveles principales:

- *El agente Coordinador:* Su función principal es la de comunicarse con el usuario, este agente es el encargado de recoger los datos del donante y de los pacientes que se van a analizar. Este agente realiza un primer filtraje para enviar tan solo los pacientes que están esperando el órgano que esta disponible para el transplante a tratar. Otra de las labores que realizará el Coordinador es recomendar el método MCDM a utilizar a partir de puntuaciones que anteriormente el usuario ha asignado a cada uno de los diferentes métodos. Cuando el agente obtenga el resultado, se lo mostrará al usuario por pantalla y le pedirá que puntúe el resultado obtenido. Hay que resaltar que el agente Coordinador esta preparado para interactuar con varios agentes Especialistas.
- *El agente Especialista:* Su función principal es la de realizar un filtrado sobre la lista de pacientes que le envía el Coordinador, aplicando una serie de reglas que se han de cumplir para poderse hacer el transplante. Otra de sus funciones, no menos importante que la anterior, es la de transformar cada uno de los atributos que se consideran a criterios de adecuación con el donante utilizando valores cualitativos. El conjunto de atributos que se evalúan para cada paciente son los siguientes:
 - Distancia.
 - Adecuación del peso del individuo.
 - Adecuación del tamaño del órgano
 - Edad
 - Tiempo de espera.

Una vez hecho esto, el agente se comunica con los diferentes agentes MCDM que existen en ese momento en el sistema, les envía los datos que ha obtenido y una manera de interpretar los vocabularios según la semántica utilizada, negaciones, conjuntos difusos, ... Cuando el agente obtenga los resultados de los distintos agentes MCDM, los enviará al agente Coordinador, el cual, como ya hemos explicado en el punto anterior, se los mostrará al usuario.

- *Agentes MCDM (Método de Decisión Multicriterio):* El objetivo de estos agentes consiste en a partir de una lista de receptores y de sus valores de adecuación al donante en función de diversos atributos, proporcionar una lista ordenada según las probabilidad de éxito del transplante. Los algoritmos de ayuda a la toma de decisiones multicriterio no pretende sustituir a los especialista médicos en el proceso de elección del candidato a para un determinado órgano, sino que facilita el trabajo ofreciendo una ordenación de las alternativas. La forma de expresar la viabilidad del transplante será también utilizando términos cualitativos que simplifiquen la comprensión de los resultados. Los diferentes agentes MCDM que componen el sistema son:
 - MCDM/Negation Functions
 - MCDM/2-Tuple
 - MCDM/Fuzzy-Sets

Con esta estructura, añadir nuevos agentes será tan sencillo como implementarlos y asociarlos al agente de su tipo.

4.1 Ventajas del uso de sistemas multiagente

El sistema presenta una serie de ventajas gracias al hecho de usar un Sistema Muti-Agente:

- *Eficiencia*: Los agentes MCDM se ejecutarán de forma paralela o concurrente (dependiendo si están en una o varias máquinas) mejorando considerablemente el tiempo de respuesta general del sistema.
- *Fiabilidad*: Debido a la existencia de diversos agentes MCDM, la probabilidad de que todos los agentes falles es muy baja (principalmente si se encuentran en máquinas diferentes) de forma que siempre obtendremos algún resultado.
- *Flexibilidad*: El sistema nos permite añadir o eliminar agentes MCDM del sistema de una forma sencilla, así como dar preferencia a un cierto método gracias a la capacidad de adaptación y aprendizaje.
- *Modularidad*: Podemos dividir el problema estructurándolo en diferentes niveles como hemos representado en el esquema anterior.

4.2 Protocolo de Comunicación

En el proceso de comunicación es muy importante el protocolo usado, ya que éste define que comportamiento tendrán los distintos agentes durante la comunicación. En nuestro caso el objetivo principal es pedir la realización de acciones, enviar datos y resultados. Para tal efecto, entro todos los protocolos que define la FIPA y que JADE permite implementar, hemos elegido el FIPA-Request dado que es el que más se ajusta a nuestras necesidades. En líneas generales el comportamiento es el que se muestra en la siguiente figura:

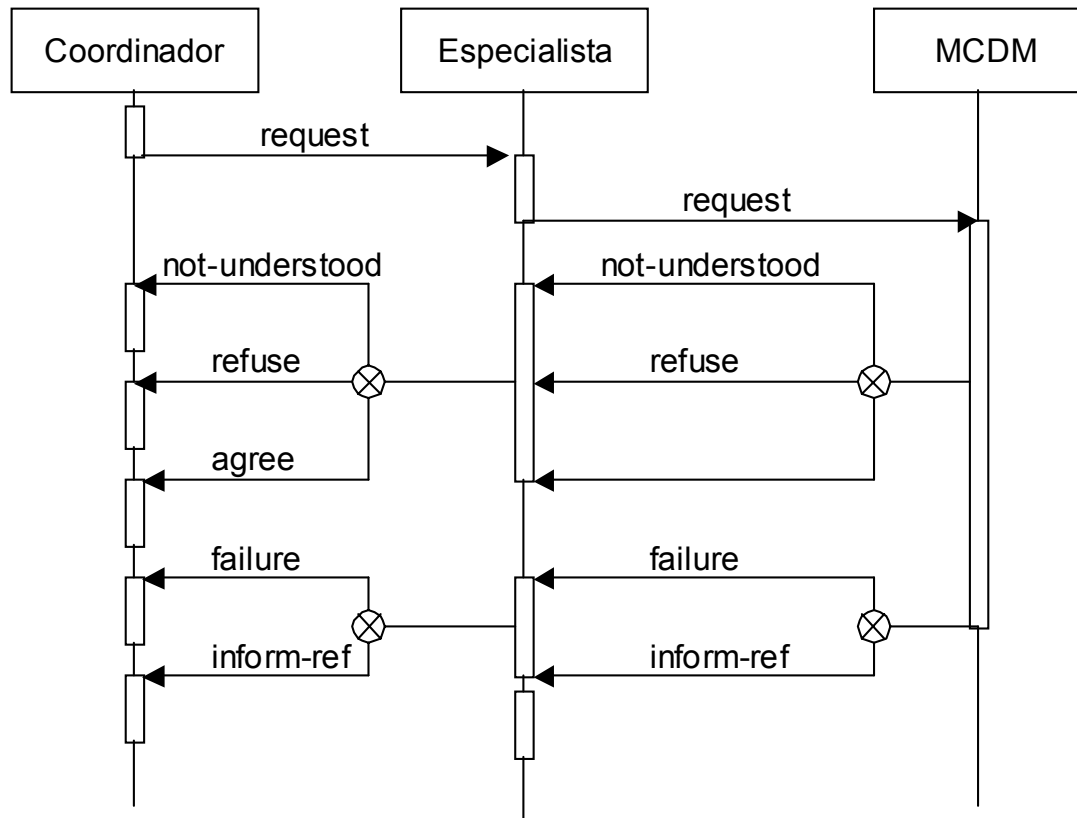


Figura XXXX. Diagrama de intercambio de mensajes

4.3 Mensajes

En el sistema se utilizan los siguientes mensajes para las distintas conversaciones entre los agente:

- **Request:** Con este mensaje un agente pide a otro que realice una determinada acción. En nuestro sistema se usan dos mensajes de este tipo, uno que envía el agente Coordinador al agente Especialista pasándole un donante y una lista de pacientes para que los ordene, el otro mensaje de este tipo es el que el Especialista envía a los agentes MCDM para que ordenen una lista de pacientes que le pasa en el mensaje.
- **Agree:** Indica que se ha aceptado la acción comenzándose a realizar, de forma que cuando se complete el proceso se enviará un nuevo mensaje.
- **Failure:** Indica que se ha producido algún error en el proceso de realización de la acción por lo que no se ha podido llegar a obtener el resultado.
- **Inform-ref:** Este mensaje contendrá el resultado obtenido tras la realización de la acción pedida. En nuestro sistema se envía una lista de receptores ordenada.

- **Not-understood:** Indica que el agente receptor del mensaje no ha sido capaz de extraer el contenido del mensaje correctamente por lo que no puede interpretar los datos. Esta situación no debería de producirse a no ser que el Especialista o el Coordinador compongan mal el mensaje.
- **Refuse:** Indica que el agente ha analizado los datos y ha encontrado algo que no es correcto y por tanto no realizará la acción (por ejemplo los pesos no suman 100).

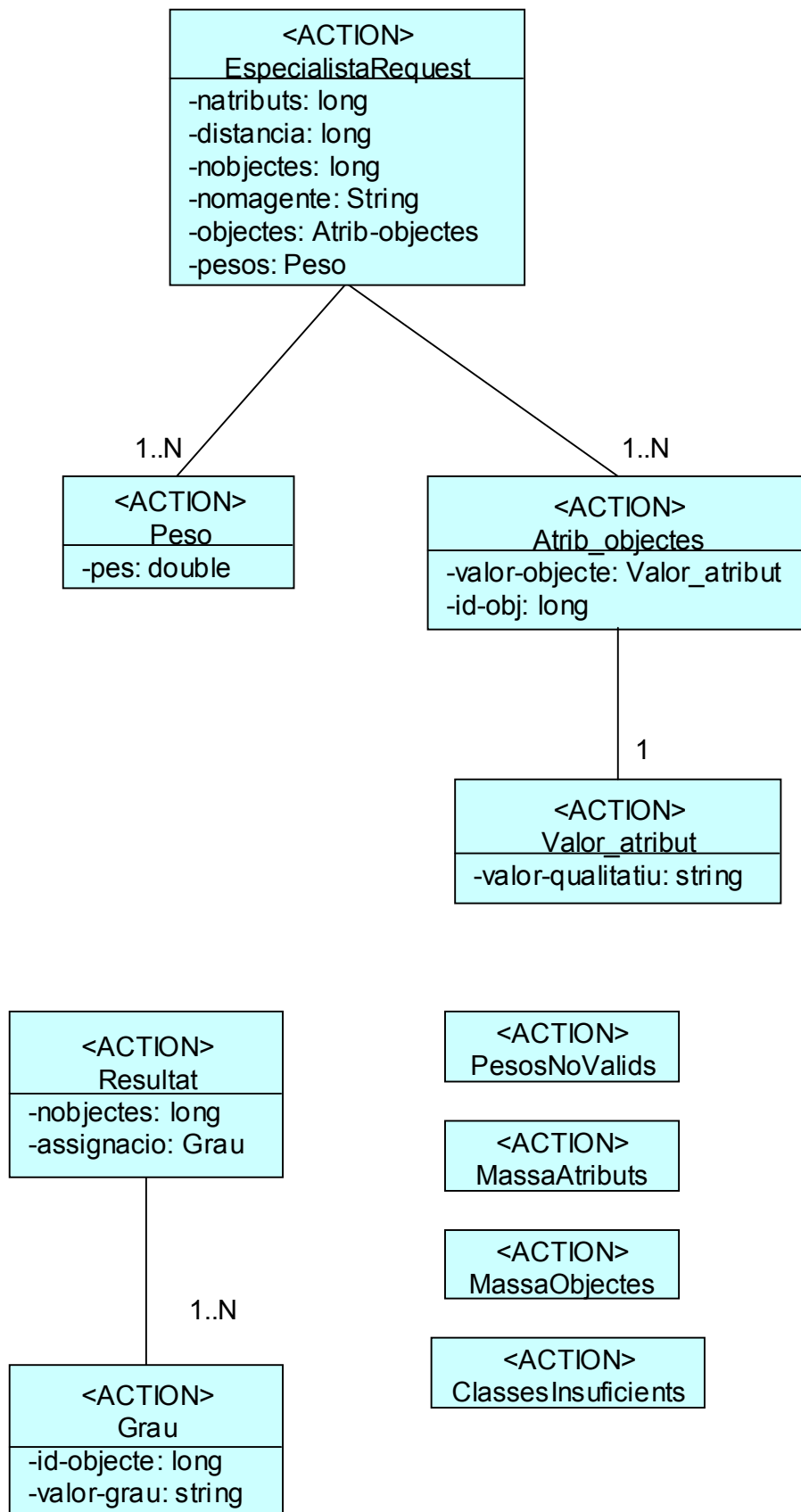
4.4 Ontología

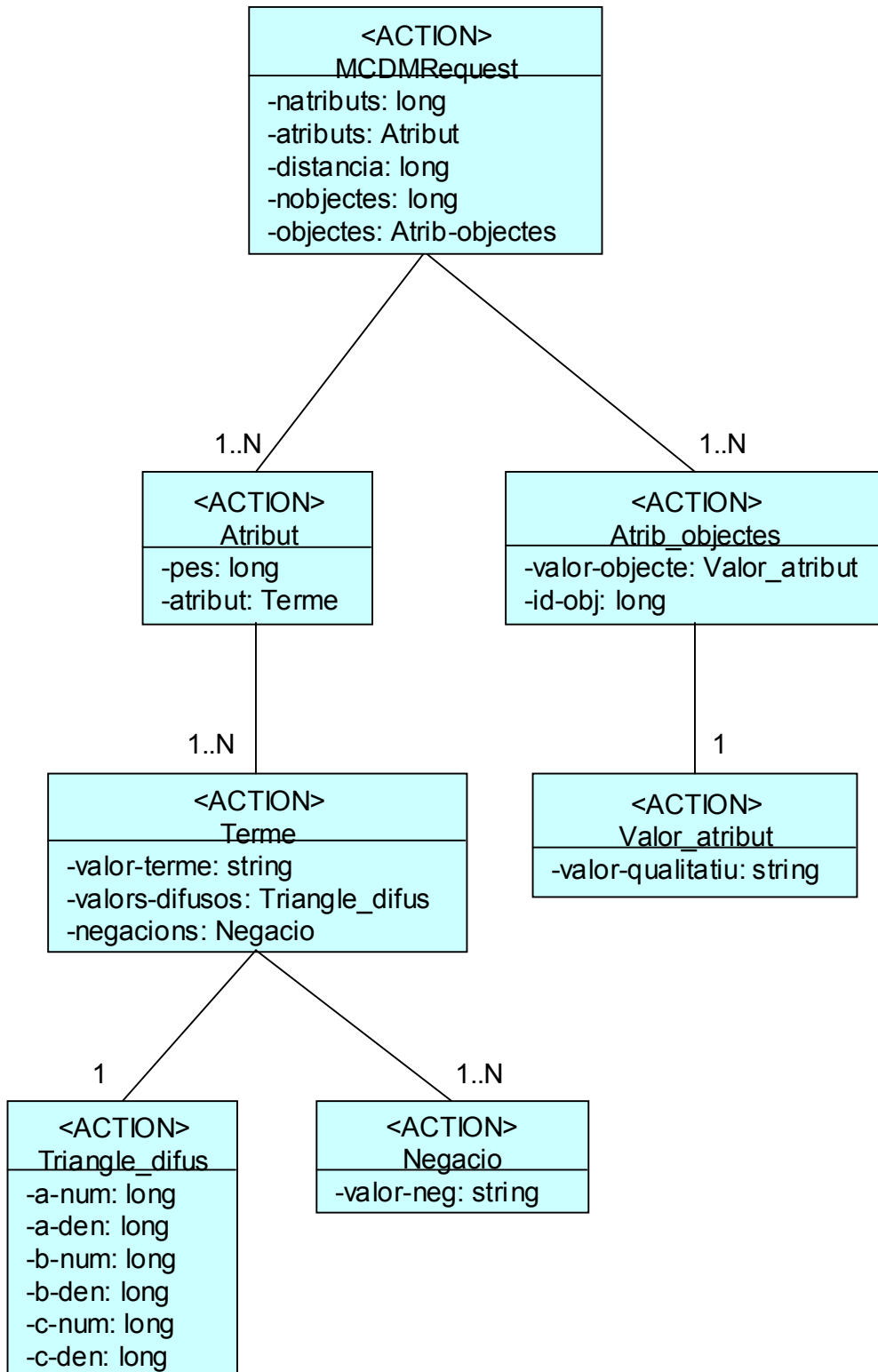
El intercambio de información entre agentes es muy abundante, por lo que la correcta interpretación de los datos resulta fundamental para conseguir un resultado correcto. Por tanto es necesario definir unas reglas y estructuras que tendrán que seguir los datos de forma que los dos interlocutores de la conversación puedan entender el significado.

Este es el punto donde se define una Ontología que se utilizará para componer el contenido de los mensajes. Sus características ya se han explicado anteriormente pero a modo de resumen, consiste en la definición de objetos o *frames* que están formados por *slots* que pueden ser de tipo simple o otros *frames*. En nuestro caso, se enviarán datos durante los procesos de *request* y en el momento de retornar los resultados. Los objetos que se utilizan para el intercambio de información son los siguientes::

- MCDMREQUEST: Objeto que utiliza el Especialista para enviar la información necesaria a los diferentes agentes MCDM.
- ESPECIALISTAREQUEST: Objeto que se usa el Coordinador para enviar a los Especialistas la información necesaria para comenzar una ordenación de pacientes.
- RESULTAT: Objeto que utilizan tanto el Especialista como los agentes MCDM para enviar los resultados tras terminar el proceso que se les ha pedido.
- MASSA_OBJECTES: Se usa por el Especialista y por los agentes MCDM para indicar que el número de pacientes máximo se ha sobrepasado.
- MASSA_ATRIBUTS: Se usa por el Especialista y por los agentes MCDM para indicar que el número de atributos máximo se ha sobrepasado.
- CLASSES_INSUFICIENTS: Objeto utilizado por el agente ClusDM y por el Especialista para indicar que el número de clases finales insuficientes.
- PESOS_NO_VALIDS: Objeto que usan el Especialista y los agentes MCDM para indicar que los pesos pasados en un mensaje anterior, no son válidos.

A continuación se muestra un esquema de la ontología siguiendo la notación definida por la FIPA [FIPA, 2001]:





5. TOMA DE DECISIONES MULTICRITERIO CON ATRIBUTOS LINGÜÍSTICOS

Como ya se ha dicho, el núcleo principal de todo el sistema será el proceso de ordenación de los diversos receptores de un órgano a partir del donante. Para hacerlo, se tienen en cuenta el conjunto de criterios (de tipo cualitativos) explicados en el apartado del agente Especialista.

Utilizaremos métodos de ayuda a la toma de decisiones multicriterio, los cuales nos van a proporcionar una lista de pacientes ordenados descrita por una serie de valores cualitativos donde los pacientes no compatibles se habrán eliminado. Como el sistema trabaja manejado por un usuario humano y muchos términos son difíciles de expresar de una forma exacta, los métodos que se usarán trabajan con términos cualitativos mucho más fáciles de expresar por el usuario.

Hay muchos algoritmos de este tipo los cuales utilizan uno u otra semántica según su forma de tratar los términos cualitativos. Las semánticas que soporta el sistema son las siguientes:

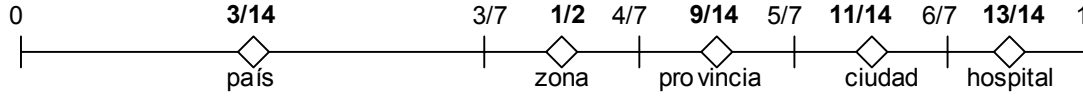
- *Negation Functions* [Torra, 1994]: descrita en la sección del agente Especialista.
- *Two-tuple* [Herrera y Martínez, 2000]: descrita en el próximo apartado.
- *FuzzyDM* [Delgado,2000]: descrita en un próximo apartado.

5.1. Funciones de Negación

Esta semántica consiste en definir una serie de funciones sobre los términos que forman el vocabulario de un criterio, expresando así la relación existente entre los valores y permiten construir un dominio entre 0 y 1 dividiendo en intervalos donde cada uno de ellos se refiere al rango de validez de uno de los términos. Si, por ejemplo, consideramos el caso de la distancia y definimos las etiquetas *país*, *zona*, *provincia*, *ciudad* y *hospital*, podemos observar como el rango de valores numéricos correspondientes a cada uno de ellos es diferente. Para poder saber cual es la distribución de los términos lingüísticos, se utilizan las funciones de negación [Torra, 1994]. Esta es una forma de expresar cual o cuales son los términos opuestos a un valor determinado. El hecho de que una etiqueta tenga diversas negaciones, nos informará que cubre un rango mas grande que otras.

Funciones de negación para el atributo distancia:

Neg(país)=provincia, ciudad, hospital
Neg(zona)=zona
Neg(provincia)=país
Neg(ciudad)=país
Neg(hospital)=país



Para trabajar con términos lingüísticos durante el proceso de ayuda a la toma de decisiones, nos interesará utilizar valores numéricos en algún momento. Tal como se define en [Torra, 1994], para cada vocabulario se puede hacer corresponder un intervalo a cada término de manera que se cumplan las relaciones explicadas por la función de negación. Estos intervalos cubren el dominio real $[0,1]$.

Se puede calcular los extremos inferiores (m_i) y superiores (M_i) del i -ésimo según la fórmula:

$$[m_i, M_i] = \left[\frac{\sum_{x < x_i} |Neg(x)|}{\sum_{x \in X} |Neg(x)|}, \frac{\sum_{x \leq x_i} |Neg(x)|}{\sum_{x \in X} |Neg(x)|} \right]$$

5.2. 2-tupla

En este caso los objetos tendrán una serie de atributos expresados mediante términos. Dichos términos pertenecerán a un vocabulario conocido acotado y continuo de forma que el término i de cualquiera de los objetos pertenecerá a un mismo vocabulario V_i . Para representar el significado (semántica) de cada término se puede usar la notación 2-tupla.

Una 2-tupla es un par (s, α) , donde la s es un término del vocabulario permitido y α es un valor real entre -0.5 y 0.5 el cual significa la desviación respecto al centro del término.

5.3. Conjuntos Difusos (Fuzzy Sets)

Otra aproximación consiste en que los objetos se describan mediante una serie de atributos expresados mediante términos difusos. Dichos términos pertenecerán a un vocabulario conocido acotado y continuo de forma que el término i de cualquiera de los objetos pertenecerá a un mismo vocabulario V_i .

Un conjunto difuso se caracteriza por asignar diferentes grados de pertenencia al conjunto a los diferentes elementos del dominio. Por ejemplo, si consideramos la altura de una persona como universo de discurso, o dominio, podemos definir diferentes conjuntos difusos asociados a términos cualitativos: alto, normal, bajo. Dada la altura de una persona concreta, su grado de pertenencia a cada uno de estos tres conjuntos difusos será diferente.

La pertenencia a los conjuntos difusos puede expresarse mediante diferentes tipos de funciones. En nuestro caso consideraremos que la pertenencia tiene una forma triangular, excepto en los extremos, tal y como se muestra en la figura TT. El grado de pertenencia máximo es 1.

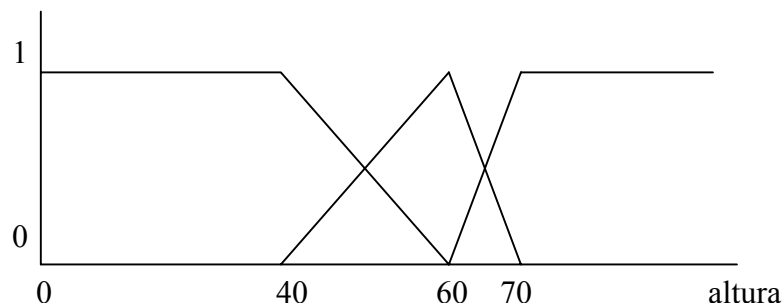


Figura TT. Representación de un vocabulario difuso

6. AGENTES DEL SISTEMA

En este apartado se explicarán los distintos agentes que intervienen en el sistema de este proyecto, los cuales han sido implementados por distintas personas de la siguiente forma:

- CoordinadorAgent (Coordinador de transplantes): Agente implementado por Héctor Alonso.
- EspecialistaAgent (Especialista en transplantes): Agente implementado por David Sánchez [Sánchez, 2001] y modificado por Héctor Alonso para adaptarlo al nuevo sistema.
- ClusDMResponderAgent: Agente MCDM que ejecuta el método ClusDM, implementado por David Sánchez [Sánchez, 2001].
- TwoTupleAgent: Agente MCDM que ejecuta el método 2-tupla, implementado por Héctor Alonso.
- FuzzyAgent: Agente MCDM que ejecuta el método Fuzzy Sets, implementado por Héctor Alonso.

6.1. Coordinador de transplantes

Este agente es el que más cerca se encuentra del usuario, es decir, el usuario a trabes de la interface gráfica se comunica con él. Una vez obtenidas las órdenes, el agente entrará en contacto con los agentes Especialistas para comunicarles el trabajo a realizar.

Datos de entrada

El agente Coordinador recibirá los siguientes datos de entrada mediante la interface gráfica que ofrece al usuario:

- Lista de pacientes: El usuario deberá cargar un fichero de pacientes en el que cada paciente contendrá los siguientes datos:
 1. Código del paciente (identificador único del paciente): Dato de tipo Long.
 2. Nombre del paciente: Dato de tipo String.
 3. Prime Apellido del paciente: Dato de tipo String.
 4. Segundo Apellido del paciente: Dato de tipo String.
 5. Hospital donde se encuentra el paciente: Dato de tipo String.
 6. Localidad donde se encuentra situado el hospital: Dato de tipo String.
 7. Provincia donde se encuentra la localidad: Dato de tipo String.

8. Tipo de Sangre: Dato de tipo enumerado que puede tener los valores O, A, B o AB.
 9. Peso del paciente: Dato de tipo decimal de doble precisión.
 10. Tamaño del órgano del paciente: Dato de tipo decimal de doble precisión.
 11. Antígenos: Cada paciente consta de 4 tipos de antígenos B y DR (dos de cada tipo), los cuales se nos expresarán en forma de cuatro boléanos ciertos o falsos según tengan o no ese tipo de antígeno.
 12. Fecha de nacimiento: Dato de tipo fecha que nos servirá para conocer la edad del paciente.
 13. Fecha de entrada en la lista de espera: Dato de tipo fecha que nos valdrá para saber cuanto tiempo lleva esperando un órgano.
- Donante con el que se compararán todas las alternativas. Constará de los mismos datos que el paciente a excepción de la fecha de espera.
 - Tipo de órgano del donante con el que se realizará el transplante.

Evaluación de los Resultados

Siempre que el usuario haya cargado un fichero de puntuaciones para los agentes, el agente especialista podrá valorar los resultados de búsquedas anteriores para así poder recomendar al usuario que tipo de agente MCDM utilizar. Para poder realizar esta función, el usuario dispondrá de un selector que se le mostrará en cada resultado con el cual podrá asignar una valoración a cada agente, las diferentes valoraciones que puede asignar son Muy_bueno, Bueno, Aceptable, Regular, Malo.

En función del número de veces que un agente haya recibido una puntuación, el agente elaborará una opinión siguiendo la siguiente escala:

- Si más de la mitad de las valoraciones son:
 - “malo”: El método no se recomienda.
 - “regular”: El método no da resultados satisfactorios.
 - “bueno”: El método es aceptable pero se tienen que estudiar bien los resultados.
 - “muy_bueno”: Se recomienda el método

Si más de un 80% de las valoraciones son “Muy_bueno” o “Bueno”: El método de recomendación especialmente.

6.2. Especialista en Transplantes

Como ya hemos comentado anteriormente, función principal de este agentes es la de realizar un filtrado sobre la lista de pacientes que le envía el Coordinador, aplicando una serie de reglas que se han de cumplir para poderse hacer el transplante. Tras recibir el mensaje y realizar el filtraje, con los pacientes válidos realiza la transformación de cada uno de los atributos que se consideran a criterios de adecuación con el donante utilizando valores cualitativos. El conjunto de atributos que se evalúan para cada paciente son los siguientes:

- Distancia.
- Adecuación del peso del individuo.
- Adecuación del tamaño del órgano
- Edad
- Tiempo de espera.

Tras este paso, el agente se comunica con los diferentes agentes MCDM que existen en ese momento en el sistema, les envía los datos que ha obtenido y una manera de interpretar los vocabularios según la semántica utilizada, negaciones, conjuntos difusos, ... Cuando el agente obtenga los resultados de los distintos agentes MCDM, los enviará al agente Coordinador, el cual, como ya hemos explicado en el punto anterior, se los mostrará al usuario. Para obtener mas información acerca del funcionamiento del agente Especialista se puede consultar el proyecto de David Sánchez [Sánchez, 2001].

6.3 Agente ClusDM

Este agente MCDM implementa el algoritmo denominado igual que su nombre, éste usa la semántica de funciones de negación y se base en realizar un proceso de clasificación o *clustering* con los pacientes que el agentes Especialista le enviará mediante un mensaje de *Request* construyendo un árbol de clases que agrupará los diferentes pacientes pasados. El agente cogerá un corte de este árbol que tenga un número de clases alrededor de siete y a cada una de ellas le asignará un término cualitativo que exprese su grado de adecuación. Para más información a cerca de este método y este agente, se puede consultar el PFC de David Sánchez [Sánchez, 2001].

6.4 Agente TwoTuple

Este agente MCDM implemente el algoritmo de 2-tupla. Recibe un mensaje del agente Especialista con todos los pacientes así como los datos que necesita para poder realizar la ordenación (conjuntos difusos de los vocabularios en los que están expresados los términos de los pacientes, pesos de cada término, ...).

El método se divide en 3 fases:

- Transformar todos los valores para que pertenezcan a un mismo vocabulario cualitativo con tuplas.
- Agregar los valores de las tuplas de cada alternativa.
- Ordenar las alternativas según los valores obtenidos.

Los objetos inicialmente dados nos servirán para formar una matriz de preferencias, la cual es la que se usará como entrada para el método. Una matriz de preferencias consiste en una tabla de dos dimensiones donde las filas corresponden a las diferentes alternativas de la decisión y las columnas son los criterios que nos permiten expresar las preferencias sobre estas alternativas según diferentes aspectos (diferentes propiedades, características, puntos de vista, ...).

	Criterio 1	Criterio 2	Criterio 3	...
Alternativa 1	Valor11	Valor12	Valor13	
Alternativa 2	Valor21	Valor22	Valor23	
Alternativa 3	Valor31	Valor32	Valor33	
...				

Los valores que toman los criterios se denominan Términos, y el conjunto de términos posibles para un criterio es su Vocabulario. Los términos de un vocabulario están ordenados de menos preferido a mas preferido en relación a la decisión que se ha de tomar. En el caso de nuestro sistema (pacientes, donante), el valor mas preferido es el que tendría que tener un paciente para poder recibir el órgano con éxito. Los términos serán de tipo cualitativo.

Datos de entrada

Los datos de entrada que necesita el agente son:

- Tabla con los términos del vocabulario de cada atributo a evaluar.
- Tabla con los triángulos difusos del vocabulario de cada atributo a evaluar.
- Tabla con los objetos (pacientes), que contiene los valores cualitativos que el Especialista ha asignado para cada atributo.

Etapas del proceso

Transformar los valores iniciales a un vocabulario común

A partir de los vocabulario iniciales, encontraremos un vocabulario que es el que se usará como vocabulario común. Para hacerlo se sigue el siguiente algoritmo:

1. Miramos la granularidad (numero de términos) de cada vocabulario.

2. Si hay un único vocabulario con granularidad máxima, entonces ésta será la granularidad del vocabulario común.
3. Si no hay un vocabulario con granularidad máxima:
 - a. Si todos los vocabularios tienen exactamente los mismos términos y con el mismo significado, entonces elegimos la granularidad de uno de estos vocabularios como la del vocabulario común.
 - b. Si son diferentes, entonces se toma como granularidad del vocabulario común una más grande que la de cualquiera de los vocabularios iniciales.
4. Cuando tengamos la granularidad del vocabulario común, calcularemos este vocabulario de forma uniforme con el número de términos que nos indique la granularidad.

Una vez hemos calculado el vocabulario común, podemos transformar los términos de los otros vocabularios. Para cada término, I_i , a traducir haremos lo siguiente:

- Para cada termino, C_k , del vocabulario común, S_t , calcularemos el punto de máxima intersección

$$\alpha_k = \max \min \{ \mu_{I_i}(y), \mu_{C_k}(y) \} \text{ guardándonos la tupla } (C_k, \alpha_k)$$

Al final nos quedará una lista de tuplas, con tantas como términos hay en el vocabulario común. Eso será un conjunto de conjuntos difusos, de los cuales lo que nos interesa son los valores α_k , porque el resto son los términos y no varían. $r_{ij} = (\alpha_0, \alpha_1, \dots, \alpha_g)$.

Con esta representación de conjuntos difusos, r , obtendremos un valor real mediante la siguiente formula:

$$\beta = \frac{\sum_{j=0}^g j \alpha_j}{\sum_{j=0}^g \alpha_j}$$

A continuación debemos pasar la representación que hemos obtenido a una representación 2-tupla.

Para pasar un número, β , entre 0 y la granularidad del vocabulario (g), a 2-tupla, realizaremos la siguiente fórmula:

$$\Lambda(\beta) = (S_i \text{ tal que } i = \text{round}(\beta), \alpha = \beta - i)$$

Dicha fórmula nos viene a decir que redondeando β obtenemos la posición del término que necesitamos, y la diferencia entre este redondeo y el valor real es lo que denominamos α .

Agregar los valores de las tuplas de cada alternativa

Para agregar los valores de las 2-tuplas, primeramente debemos trasformarlo en números usando la inversa de la función Λ . Una vez realizada esta operación haremos un cálculo de agregación numérica. En nuestro caso usaremos una media ponderada, pero el código se ha implementado de tal forma que sea fácilmente cambiable por otro método.

Ordenar las alternativas según el valor obtenido

Con los resultados numéricos obtenidos en la fase de agregación, podemos ordenar las diferentes alternativas. En nuestro caso usaremos un algoritmo de “Burbuja”, pero este está implementado de forma que será fácilmente cambiable en un futuro.

6.5 Agente FuzzyDM

Este agente MCDM implemente el algoritmo de Fuzzy-sets. El agente recibe un mensaje del agente Especialista con la lista de pacientes así como los datos que necesita para poder realizar la ordenación de dichos pacientes, conjuntos difusos de los vocabularios en los que están expresados los términos de los pacientes, pesos de cada término, ...

El método se divide en 3 fases:

- Transformar todos los valores para que pertenezcan a un mismo vocabulario cualitativo difuso.
- Agregar los valores de las tuplas de cada alternativa.
- Ordenar las alternativas según los valores obtenidos.

Los objetos inicialmente dados nos servirán para formar una matriz de preferencias, la cual es la que se usará como entrada para el método. Una matriz de preferencias consiste en una tabla de dos dimensiones donde las filas corresponden a las diferentes alternativas de la decisión y las columnas son los criterios que nos permiten expresar las preferencias sobre estas alternativas según diferentes aspectos (diferentes propiedades, características, puntos de vista, ...).

	Criterio 1	Criterio 2	Criterio 3	...
Alternativa 1	Valor11	Valor12	Valor13	
Alternativa 2	Valor21	Valor22	Valor23	
Alternativa 3	Valor31	Valor32	Valor33	
...				

Los valores que toman los criterios se denominan Términos, y el conjunto de términos posibles para un criterio es su Vocabulario. Los términos de un vocabulario están ordenados de menos preferido a mas preferido en relación a la decisión que se ha de tomar. En el caso de nuestro sistema (pacientes, donante), el valor mas preferido es el que tendría que tener un paciente para poder recibir el órgano con éxito. Los términos serán de tipo cualitativo.

Datos de entrada

Los datos de entrada que necesita el agente son:

- Tabla con los términos del vocabulario de cada atributo a evaluar.
- Tabla con los triángulos difusos del vocabulario de cada atributo a evaluar.
- Tabla con los objetos (pacientes), que contiene los valores cualitativos que el Especialista ha asignado para cada atributo.

Etapas del proceso

Transformar los valores iniciales a un vocabulario común

A partir de los vocabulario iniciales, encontraremos un vocabulario que es el que se usará como vocabulario común. Para hacerlo se sigue el siguiente algoritmo:

5. Miramos la granularidad (numero de términos) de cada vocabulario.
6. Si hay un único vocabulario con granularidad máxima, entonces ésta será la granularidad del vocabulario común.
7. Si no hay un vocabulario con granularidad máxima:
 - a. Si todos los vocabularios tienen exactamente los mismos términos y con el mismo significado, entonces elegimos la granularidad de uno de estos vocabularios como la del vocabulario común.
 - b. Si son diferentes, entonces se toma como granularidad del vocabulario común una más grande que la de cualquiera de los vocabularios iniciales.
8. Cuando tengamos la granularidad del vocabulario común, calcularemos este vocabulario de forma uniforme con el número de términos que nos indique la granularidad.

Una vez hemos calculado el vocabulario común, podemos transformar los términos de los otros vocabularios. Para cada término, I_i , a traducir haremos lo siguiente:

- Para cada termino, C_k , del vocabulario común, S_t , calcularemos el punto de máxima intersección

$$\alpha_k = \max \min \{ \mu_{I_i}(y), \mu_{C_k}(y) \} \text{ guardándonos la tupla } (C_k, \alpha_k)$$

Al final nos quedará una lista de tuplas, con tantas como términos hay en el vocabulario común. Eso será un conjunto de conjuntos difusos, de los cuales lo que nos interesa son los valores α_k , porque el resto son los términos y no varían. $r_{ij} = (\alpha_0, \alpha_1, \dots, \alpha_g)$.

Agregar los valores de las tuplas de cada alternativa

Una vez tenemos todos los datos de la matriz representados con los vectores r_{ij} , se aplica algún operador numérico a cada elemento del vector de una misma alternativa, así obtenemos otro vector con los valores consenso.

Nosotros aplicaremos una media ponderada, pero el código se ha implementado de tal forma que sea fácilmente cambiable por otra forma de agregación.

Ordenar las alternativas según el valor obtenido

El modelo de conjuntos difusos, r_{ij} , usado hasta ahora, complica el proceso de ordenación y selección de las mejores alternativas. Por eso cambiaremos la notación a relaciones de preferencias difusas. Para realizar el cambio utilizamos la siguiente fórmula:

$$b_{ij} = \max_{cl} \min_{ch \leq cl} \{ \mu_{ri}(cl), \mu_{rj}(ch) \}$$

Dicha fórmula nos viene a decir que para cada elemento del primer vector r_i , se miran todos los valores anteriores en r_j y se calcula el mínimo de cada par. Después se coge el máximo de todos los mínimos encontrados.

Si representamos los valores obtenidos en forma de matriz, obtendremos una matriz de este tipo:

	X	Y	Z
X	-	0.5	0.4
Y	0.3	-	0.4
Z	0.4	0.4	-

Con estos valores se puede calcular el grado de no-dominanza de una alternativa, NDD_i , de la siguiente manera:

$$NDD_i = \min_{x_j} \{1 - b_{sji}\} \text{ donde } j \text{ es diferente de } i$$

Y el valor b_{sji} se obtiene calculando el máximo entre $(b_{ji} - b_{ij})$ y 0

Según la matriz anterior obtendríamos lo siguiente:

	X	Y	Z
X	-	0.2	0
Y	0	-	0
Z	0	0	-

$$NDD_x = \min((1-0), (1-0)) = 1$$

$$NDD_y = \min((1-0.2), (1-0)) = 0.8$$

$$NDD_z = \min((1-0), (1-0)) = 1$$

Para finalizar, ordenaremos los valores numéricos de los NDD_i , de forma que en nuestro ejemplo tenemos que las mejores alternativas son X y Z y después Y. El algoritmo de ordenación usado es el de Burbuja, pero se ha implementado de tal forma que sea fácilmente sustituible por otro algoritmo.

7. DISTRIBUCIÓN MODULAR DEL SISTEMA

Para poder estructurar todo el sistema y permitir su ejecución en entornos distribuidos, el código se ha dividido en una serie de módulos que se describen en los próximos apartados.

7.1. CoordinadorAgent.java

Es el objeto Java que implementa las funciones para enviar los datos necesarios a los agentes Especialista. Al instanciar este objeto, se realiza una instancia al objeto `gui.java`, el cual se describe más a delante, a traves de éste entrará en comunicación con el usuario el cual le indicará que tiene que hacer.

Al ser un agente, esta clase implementa una serie de comportamientos que definen su manera de actuar. Los comportamientos de éste son:

- **class MCDMInputBehaviour**: clase que extiende el comportamiento simple **SequentialBehaviour** que se utiliza para la ejecución lineal de una serie de acciones. Este comportamiento será el principal del agente y se encargará de esperar las órdenes del usuario (mediante la comunicación con el objeto `gui.java`). Cuando reciba la orden, será el encargado de mantener la comunicación con los agentes Especialistas del sistema, además captará las valoraciones que el usuario de a los diferentes resultados obtenidos en las búsquedas. Las funciones principales de este comportamiento son:
 - **MCDMInputBehaviour (Agent myAgent)**: constructor del comportamiento.
 - **void preAction()**: espera a que el usuario de la orden de comenzar una nueva búsqueda, una vez esto ocurra, creará un mensaje de *Request* a enviar al agente Especialista elegido por el usuario, tras esto instanciará el subcomportamiento que se encarga de implementar el protocolo *FIPA-Request* con el agente Especialista.
 - **ACLMessage createEspecialistaRequest()**: función auxiliar que inicializa los campos necesarios de un mensaje Request como son: el agente que lo envía, la ontología usada, lenguaje usado, ... También se encarga de recoger los datos de los ficheros de pacientes y donantes así como el agente MCDM al que va destinada la búsqueda para introducirlos en el mensaje.
 - **void postAction()**: función que nos sirve para realizar las acciones necesarias después de cada ejecución del comportamiento principal. Nosotros la usamos para reiniciar el comportamiento de nuevo.
 - **void reset()**: reinicia el comportamiento principal comprobando que no exista ningún subcomportamiento secundario en ejecución.
- **class EspecialistaBehaviour**: clase que extiende el comportamiento **FipaRequestInitiatorBehaviour** que implementa la parte *Initiator* del protocolo *FIPA-Request*. En este caso se desarrollarán una serie de funciones definidas por JADE que se encarguen de manejar los posibles mensajes que nos pueda contestar un agente Especialista (*inform*, *failure*, ...) generando las respuestas. Dentro de éste nos encontramos con las siguientes funciones:

- **EspecialistaBehaviour (Agent agent, ACLMessage requestMsg):** constructor del comportamiento.
 - **void handleAgree(ACLMessage msg):** controlador de mensajes *Agree*.
 - **void handleInform(ACLMessage msg):** controlador de mensajes *Inform*. En este mensaje vendrá el resultado, por lo que se extrae y se muestra por pantalla al usuario.
 - **void handleNotUnserstood(ACLMessage msg):** controlador de mensajes *not-understood*.
 - **void handleFailure(ACLMessage msg):** controlador de mensajes *Failure*. Se recupera la causa del fallo y se muestra por pantalla al usuario.
 - **void handleRefuse(ACLMessage msg):** controlador de mensajes *Refuse*, se recupera la razón el rechazo y se muestra por pantalla al usuario.
-
- **void setup():** este método es llamado una sola vez al comienzo, y se encarga registrar al agente en el DF indicando que es de tipo Coordinador y que servicio proporciona así como de inicializar la ontología y el lenguaje. Otra de las funciones más importantes que realiza es la de instanciar el objeto gráfico que entrara en contacto con el usuario, y de iniciar el comportamiento principal definido anteriormente.
 - **void takeDown():** función encargada de finalizar correctamente al agente cuando se acaba su ejecución. Eliminará los posibles subcomportamientos que pueda tener y se desregistrará del DF.
 - **Vector BuscarAgents():** función auxiliar que realiza la búsqueda de los agentes de tipo Especialista disponibles en el sistema. Retorna un vector con el nombre del agente y su semántica.
 - **Vector BuscarAgentsMCDM():** función auxiliar que realiza la búsqueda de los agentes de tipo MCDM disponibles en el sistema. Retorna un vector con el nombre del agente y su semántica.
 - **void Parse_Missatge(EspecialistaRequest cr):** otra función auxiliar que se usa para rellenar un mensaje *Request* con los datos que ha seleccionado el usuario, mediante las reglas definidas en la ontología.

7.2. DadesEspecialista.java

Objeto que define el conjunto de estructuras de datos que necesita el objeto Especialista para realizar el proceso de filtrado y comparación de los pacientes. También incluye las estructuras donde se guardan los resultados que necesitarán los diversos algoritmos de ordenación. Los diferentes componentes del objeto son:

- Funciones:
 - **void Inicialitza (int n):** función que inicializa las tablas con la dimensión que se le pasa por parámetros.
- Datos internos del objeto:
 - **int tatulaABO[][]:** tabla de tipos de sangre. Cada fila indica un tipo de sangre. Cada columna un tipo compatible con la fila a la que pertenece.
 - **String zones[][]:** tabla donde se definen las zonas del país. Cada fila indica una zona y cada columna es una provincia que pertenece a la zona de la fila.
- Datos de entrada:
 - **String NomAtributs[]:** nombre de los atributos.
 - **int nPacients:** número de pacientse.
 - **int nAtributs:** número de atributos.
 - **Pacient pacients[]:** tabla de pacientes a analizar.
 - **Donant donant:** objeto donante.
- Datos a enviar a los agentes de ordenación:
 - **String Atributs[][]:** tabla de vocabularios de cada atributo. Cada fila es un atributo y cada columna es un término del vocabulario del atributo.
 - **String NegAtributs[][][]:** tabla de negaciones de los elementos del vocabulario de los atributos. La primera dimensión es el atributo, la segunda es el término dentro del vocabulario del atributo y la tercera es la lista de negaciones para el término del vocabulario del atributo.
 - **int ValorsDifusos[][][][]:** tabla de triángulos difusos para los elementos del vocabulario. La primera dimensión es el atributo, la segunda es el término, la tercera es la lista de vértices del triángulo y la cuarta son los valores de los vértices (numerador y denominador).
 - **double pesos[]:** pesos de los atributos.
 - **int tipus_dist:** función de distancia escogida.
 - **String pacients_atrib[][]:** tabla de los pacientes con los valores de cada atributo. Las filas son los pacientes y las columnas son los valores asignados a cada atributo.
 - **int idPac[]:** tabla de identificadores de pacientes.

7.3. Especialista.java

Es el objeto Java que implementa las funciones de filtraje de los pacientes y de transformación de los datos comparándolos con el donante. El modulo no tiene funcionalidad propia por lo que requiere que otro lo instancie. Los métodos principales son:

- **Especialista (DadesEspecialista dades):** es el constructor y el único accesible externamente. Pasándole los datos de entrada por parámetro realiza todo el proceso de filtrado y transformación guardando los resultados en las estructuras que se han definido en el objeto de datos para tal efecto. Los códigos de los pacientes serán los que se le pase como identificador de objeto a los algoritmos de ordenación.
- Funciones auxiliares internas del objeto:
 - **String Avalue_distancia (Pacient pacient, Donant donant, String zones[]):** compara los hospitales, localidades y provincias del paciente con las del donante y retorna el atributo “distancias”.
 - **boolean Avalua_sang (Pacient pacient, Donant donant, int taulaABO[]):** Compara el tipo de sangre del donante con el del receptor retornando si son o no compatibles.
 - **String Avalua_pes(Pacient pacient, Donant donant):** a partir del peso del donante y del receptor retorna el término del vocabulario del atributo “peso” o nulo si el paciente no es válido.
 - **String Avalua_tamany(Pacient pacient, Donant donant):** a partir del tamaño del órgano del donante y del receptor retorna el término del vocabulario del atributo “tamaño del organo”.
 - **String Avalua_antigens(Pacient pacient, Donant donant):** a partir de los antígenos del donante y del receptor retorna el término del vocabulario del atributo “antígenos”.
 - **String Avalua_edat(Pacient pacient, Donant donant):** a partir de la edad del donante y del receptor retorna el término del vocabulario del atributo “edad”.
 - **String Avalua_espera(Pacient pacient):** comparando la fecha de entrada a la lista de espera del paciente y la fecha actual, se retorna el término del vocabulario del atributo “tiempo de espera” correspondiente.
 - **boolean Avalua_organ(Pacient pacient):** función que nos sirve para filtra pacientes a partir de su edad y del órgano que necesitan. Se retorna si el trasplante es o no posible.

7.4. EspecialistaAgent.java

Este modulo añade la funcionalidad de agente al módulo Especialista.java y junto con el le denominamos Especialista en transplantes. Espera las órdenes de un agente Coordinador le transmita para así iniciar una comunicación con un agente MCDM.

Al ser un agente, esta clase implementa una serie de comportamientos que definen su manera de actuar. Los comportamientos de éste son:

- **class MCDMInputBehaviour**: clase que extiende el comportamiento simple **SequentialBehaviour** que se utiliza para la ejecución lineal de una serie de acciones. Este comportamiento será el encargado de mantener la comunicación con los agentes MCDM del sistema iniciando con ellos un **Request**. Tras esto iniciara el subcomportamiento **MCDMBehaviour** para valorar la respuesta del agente con el que ha iniciado la comunicación. Las funciones principales de este comportamiento son:
 - **MCDMInputBehaviour (Agent myAgent)**: constructor del comportamiento.
 - **void preAction()**: crea un mensaje de *Request* a enviar al agente MCDM que le haya indicado el agente Coordinador, tras esto instanciará el subcomportamiento que se encarga de implementar el protocolo *FIPA-Request* con el agente MCDM.
 - **ACLMessage createMCDMRequest()**: función auxiliar que inicializa los campos necesarios de un mensaje *Request* como son: el agente que lo envía, la ontología usada, lenguaje usado, ... También se encarga de realizar la instancia al módulo Especialista (el cual realizara los filtrajes de los pacientes y las transformaciones de datos), a partir del objeto **DadesEspecialista** que se habrá rellenado con los datos que nos ha enviado el Coordinador.
 - **void postAction()**: función que nos sirve para realizar las acciones necesarias después de cada ejecución del comportamiento principal. Nosotros la usamos para reiniciar el comportamiento de nuevo.
 - **void reset()**: reinicia el comportamiento principal comprobando que no exista ningún subcomportamiento secundario en ejecución.
- **class MCDMBehaviour**: clase que extiende el comportamiento **FipaRequestInitiatorBehaviour** que implementa la parte *Initiator* del protocolo *FIPA-Request*. En este caso se desarrollarán una serie de funciones definidas por JADE que se encarguen de manejar los posibles mensajes que nos pueda contestar un agente MCDM (*inform*, *failure*, ...) generando las respuestas. Dentro de éste nos encontramos con las siguientes funciones:
 - **MCDMBehaviour (Agent agent, ACLMessage requestMsg)**: constructor del comportamiento.
 - **void handleAgree(ACLMessage msg)**: controlador de mensajes *Agree*.

- **void handleInform(ACLMessage msg):** controlador de mensajes *Inform*. En este mensaje vendrá el resultado, por lo que lo enviaremos al agente Coordinador que inicio la búsqueda..
 - **void handleNotUnderstood(ACLMessage msg):** controlador de mensajes *not-understood*.
 - **void handleFailure(ACLMessage msg):** controlador de mensajes *Failure*. Se recupera la causa del fallo y se le envía al agente Coordinador que inició la búsqueda.
 - **void handleRefuse(ACLMessage msg):** controlador de mensajes *Refuse*, se recupera la razón el rechazo y se le envía al agente Coordinador que inició la búsqueda.
- class **EspecialistaBehaviour**: subclase que extiende el comportamiento **FipaRequestResponderBehaviour** para poder manejar peticiones de tipo *Request* enviadas por el agente Coordinador. Este es un comportamiento simple que se instancia se ejecuta y se destruye cada vez que el *scheduler* de JADE recibe una petición de *Request*. Las funciones principales de este comportamiento son:
 - **EspecialistaBehaviour (Agent agent, ACLMessage requestMsg):** Constructor del comportamiento.
 - **action()**: en esta función tenemos que introducir las acciones que se ejecutarán cuando recibamos una petición. En primer lugar extrae los datos contenidos en el mensaje e inicializa el objeto **DadesEspecialista**. Tras esto añade el comportamiento **MCDMInputBehaviour** para que se ejecute iniciando así el *Request* con el agente MCDM que toque. Nos guardamos el mensaje original para que el nuevo comportamiento sea capaz de finalizar el *Request* que inicio el Coordinador.
 - **boolean done()**: conjunto de acciones a realizar cuando se acaba el comportamiento, en nuestro caso nada.
 - **void reset()**: acciones a realizar cuando se reinicia el comportamiento. Como en nuestro caso esto comportamiento solo se ejecuta una vez, no hace falta que definamos nada.
 - **void setup()**: este método es llamado una sola vez al comienzo, y se encarga registrar al agente en el DF indicando que es de tipo Especialista y que servicio proporciona así como de inicializar la ontología y el lenguaje. Otra de las funciones más importantes que realiza es la de iniciar el comportamiento que se tiene que ejecutar en primer lugar **EspecialistaBehaviour**.
 - **void takeDown()**: función encargada de finalizar correctamente al agente cuando se acaba su ejecución. Eliminará los posibles subcomportamientos que pueda tener y se desregistrará del DF.
 - **Vector BuscarAgents()**: función auxiliar que realiza la búsqueda de los agentes de tipo MCDM disponibles en el sistema. Retorna un vector con el nombre del agente y su semántica.

- **void Parse_Missatge_Especialista(EspecialistaRequest cr):** función auxiliar que se usa para extraer los datos de un mensaje *Request* enviado por el coordinador e iniciar el objeto **DadesEspecialista**.
- **void Parse_Missatge(MCDMRequest cr):** otra función auxiliar que se usa para rellenar un mensaje *Request* con los datos retornados por el objeto Especialista mediante las reglas definidas en la ontología.

7.5. DatosTwoTuple.java

Objeto que define todas las estructuras de datos que tiene que recibir el algoritmo **TwoTuple** para realizar el proceso de ordenación. Consta de los siguientes elementos:

- Funciones:
 - **void Inicialitza (int nObjectes, int nAtributs):** función que inicializa la dimensión de las tablas dinámicamente.
- Conjunto de datos:
 - **int nObjectes:** número de objetos a evaluar.
 - **int nAtributs:** número de atributos para cada alternativa.
 - **String Atributs[][]:** tabla con los vocabularios de los atributos.
 - **double pesos[]:** pesos de los atributos.
 - **String objetctes[][]:** tabla de objetos con los valores de cada atributo.
 - **int idObj[]:** tabla con los identificadores de los objetos.
 - **int ValorsDifusos[][][][]:** tabla de triángulos difusos para los elementos del vocabulario.

7.6. TwoTuple.java

Es el objeto Java que implementa el algoritmo de Two-tuple. El objeto no tiene funcionalidad propia por lo que tiene que ser instanciado por otro, en nuestro caso por el agente TwoTupleAgent. Las funciones más importantes que posee este objeto son:

- **TwoTuple(DatosTwoTuple dades):** es el constructor del objeto. Se encarga de calcular el vocabulario común a partir de los datos que se le pasan. Convierte los atributos de los pacientes al vocabulario común.
- **ResultatMCDM[] getResultado():** función que se encarga de realizar el resto de los cálculos necesarios para completar el algoritmo. Retorna una tabla de objetos ResultatMCDM en los que están los identificadores de los objetos así como el término asignado a cada uno de ellos.
- Funciones internas:

- **vocabularioBase calculaVocabulario()**: Función que calcula el vocabulario común al cual convertiremos los atributos de cada objeto expresados en distintos vocabularios.
- **double max(double a, double b)**: función que retorna el número máximo de dos números dados.
- **double[] pasar_a_vocab_comun(triangulo term,vocabularioBase interv)**: función que pasa un término expresado en un vocabulario cualquiera al vocabulario común. Para ello se le pasa como parámetros el triangulo difuso del término a convertir y el vocabulario común al cual lo vamos a convertir.
- **double[] computar_2tupla(double[][][] valores,double peso[])**: función que desarrolla el algoritmo de TwoTuple con los objetos que se le pasan por parámetros teniendo en cuenta el peso de cada uno de los términos, los cuales también le son pasados.
- **void ordena_objetos(objetoGrado[] pg)**: ordena los objetos según los cálculos que hemos realizado en la función anterior.

7.7. TwoTupleAgent.java

Este es el agente asociado al objeto de *TwoTuple* explicado anteriormente. Tiene un comportamiento simple el cual recibe peticiones de ordenación. Cuando recibe el mensaje de *Request* procedente del Especialista, extrae los datos de éste y se los pasa al objeto *TwoTuple* el cual los procesa y nos pasa el resultado, éste lo introducimos en un mensaje de respuesta y se lo enviamos al Especialista. El protocolo de comunicación utilizado es de tipo *FIPA-Request*. El agente está compuesto por las siguientes partes:

- **class ResponseBehaviour**: subclase que extiende el comportamiento **FipaRequestResponderBehaviour** para poder controlar peticiones de tipo **Request** enviadas por el agente Especialista. Este es un comportamiento simple que se instancia, se ejecuta y se destruye cada vez que el scheduler de JADE recibe una petición de **Request**. Las funciones principales del comportamiento son las siguientes:
 - **ResponderBehaviour (Agent agent, ACLMessage requestMsg)**: constructor del comportamiento.
 - **action()**: en esta función tenemos que introducir las acciones que se ejecutarán cuando recibamos una petición. En primer lugar extrae los datos contenidos en el mensaje e inicializa el objeto **DatosTwoTuple**. Tras esto instanciamos al objeto **TwoTuple** para que realice los cálculos del algoritmo. Cuando obtenemos el resultado construimos con estos un mensaje de respuesta de tipo **inform** y se lo enviamos al Especialista. En caso de que haya algún fallo prepararemos un mensaje de respuesta de tipo **failure** y también se lo enviaremos al Especialista.
 - **boolean done()**: conjunto de acciones a realizar cuando se acaba el comportamiento, en nuestro caso nada.
 - **void reset()**: acciones a realizar cuando se reinicia el comportamiento. Como en nuestro caso este comportamiento solo se ejecuta una vez, no hace falta que definamos nada.

- **void setup()**: este método es llamado una sola vez al comienzo, y se encarga registrar al agente en el DF indicando que es de tipo MCDM y que servicio proporciona así como de inicializar la ontología y el lenguaje. Otra de las funciones más importantes que realiza es la de iniciar el comportamiento que se tiene que ejecutar **ResponseBehaviour**.
- **void takeDown()**: función encargada de finalizar correctamente al agente cuando se acaba su ejecución. En nuestro caso se desregistrará del DF.
- **int Parse_missatge (MCDMRequest cr, DatosTwoTuple dades)**: función auxiliar en el agente que se encarga de extraer los datos de un mensaje de *Request* y rellenar un objeto de tipo **DatosTwoTuple** el cual será pasado a un objeto **TwoTuple** para que lo procese. También comprueba que los datos que nos están pasando sean correctos (pesos válido, datos dentro de rango, ...), en caso contrario retornaremos que ha habido un fallo.

7.8. DatosFuzzySet.java

Objeto que define todas las estructuras de datos que tiene que recibir el algoritmo **FuzzySet** para realizar el proceso de ordenación. Consta de los siguientes elementos:

- Funciones:
 - **void Inicialitza (int nObjectes, int nAtributs)**: función que inicializa la dimensión de las tablas dinámicamente.
- Conjunto de datos:
 - **int nObjectes**: número de objetos a evaluar.
 - **int nAtributs**: número de atributos para cada alternativa.
 - **String Atrituts[][]**: tabla con los vocabularios de los atributos.
 - **double pesos[]**: pesos de los atributos.
 - **String objetctes[][]**: tabla de objetos con los valores de cada atributo.
 - **int idObj[]**: tabla con los identificadores de los objetos.
 - **int ValorsDifusos[][][][]**: tabla de triángulos difusos para los elementos del vocabulario.

7.9. FuzzySet.java

Es el objeto Java que implementa el algoritmo de FuzzyDM. El objeto no tiene funcionalidad propia por lo que tiene que ser instanciado por otro, en nuestro caso por el agente FuzzyAgent. Las funciones más importantes que posee este objeto son:

- **FuzzySet(DatosFuzzySet dades)**: es el constructor del objeto. Se encarga de calcular el vocabulario común a partir e los datos que se le pasan. Convierte los atributos de los pacientes al vocabulario común.

- **ResultatMCDM[] getResultado():** función que se encarga de realizar el resto de los cálculos necesarios para completar el algoritmo. Retorna una tabla de objetos ResultatMCDM en los que están los identificadores de los objetos así como el término asignado a cada uno de ellos.
- Funciones internas:
 - **vocabularioBase calculaVocabulario():** Función que calcula el vocabulario común al cual convertiremos los atributos de cada objeto expresados en distintos vocabularios.
 - **double max(double a, double b):** función que retorna el número máximo de dos números dados.
 - **double min(double a, double b):** función que retorna el número mínimo de dos números dados.
 - **double max_min(double[] v1, double[] v2):** función que realiza el cálculo de la función maxmin para dos vectores dados.
 - **double[] pasar_a_vocab_comun(triangulo term,vocabularioBase interv):** función que pasa un término expresado en un vocabulario cualquiera al vocabulario común. Para ello se le pasa como parámetros el triangulo difuso del término a convertir y el vocabulario común al cual lo vamos a convertir.
 - **double[][] agregar_valores(double[][][] valores,double[] peso):** función que realiza la tarea de agregar valores definida en el algoritmo de FuzzyDM, para realizar la tarea tiene en cuenta los objetos con sus términos ya pasados a vocabulario común y los pesos de cada uno de los términos.
 - **double[][] preferencias_difusas(double[][] valores):** calcula la matriz de preferencias difusas a partir de los datos que ha calculado la función anterior.
 - **objetoGrado[] ordena_objetos(double[][] mat):** ordena los objetos según la matriz de preferencias difusas calculada en la función anterior terminando así la ejecución del algoritmo.

7.10. FuzzySetAgent.java

Este es el agente asociado al objeto *FuzzySet* explicado anteriormente. Tiene un comportamiento simple el cual recibe peticiones de ordenación. Cuando recibe el mensaje de *Request* procedente del Especialista, extrae los datos de éste y se los pasa al objeto *FuzzySet* el cual los procesa y nos pasa el resultado, éste lo introducimos en un mensaje de respuesta y se lo enviamos al Especialista. El protocolo de comunicación utilizado es de tipo *FIPA-Request*. El agente está compuesto por las siguientes partes:

- **class ResponseBehaviour:** subclase que extiende el comportamiento **FipaRequestResponderBehaviour** para poder controlar peticiones de tipo **Request** enviadas por el agente Especialista. Este es un comportamiento simple que se instancia, se ejecuta y se destruye cada vez que el scheduler de JADE recibe una petición de **Request**. Las funciones principales del comportamiento son las siguientes:

- **ResponderBehaviour (Agent agent, ACLMessage requestMsg):** constructor del comportamiento.
 - **action():** en esta función tenemos que introducir las acciones que se ejecutarán cuando recibamos una petición. En primer lugar extrae los datos contenidos en el mensaje e inicializa el objeto **DatosFuzzySet**. Tras esto instanciamos al objeto **FuzzySet** para que realice los cálculos del algoritmo. Cuando obtenemos el resultado construimos con estos un mensaje de respuesta de tipo **inform** y se lo enviamos al Especialista. En caso de que haya algún fallo prepararemos un mensaje de respuesta de tipo **failure** y también se lo enviaremos al Especialista.
 - **boolean done():** conjunto de acciones a realizar cuando se acaba el comportamiento, en nuestro caso nada.
 - **void reset():** acciones a realizar cuando se reinicia el comportamiento. Como en nuestro caso este comportamiento solo se ejecuta una vez, no hace falta que definamos nada.
-
- **void setup():** este método es llamado una sola vez al comienzo, y se encarga registrar al agente en el DF indicando que es de tipo MCDM y que servicio proporciona así como de inicializar la ontología y el lenguaje. Otra de las funciones más importantes que realiza es la de iniciar el comportamiento que se tiene que ejecutar **ResponseBehaviour**.
 - **void takeDown():** función encargada de finalizar correctamente al agente cuando se acaba su ejecución. En nuestro caso se desregistrará del DF.
 - **int Parse_missatge (MCDMRequest cr, DatosFuzzySet dades):** función auxiliar en el agente que se encarga de extraer los datos de un mensaje de *Request* y rellenar un objeto de tipo **DatosFuzzySet** el cual será pasado a un objeto **FuzzySet** para que lo procese. También comprueba que los datos que nos están pasando sean correctos (pesos válido, datos dentro de rango, ...), en caso contrario retornaremos que ha habido un fallo.

7.11. ResultatMCDM.java

Objeto que define los datos que se han de retornar para cada paciente, como resultado del proceso de ordenación. Elementos del objeto:

- **String grau:** término asignado al objeto según su adecuación al modelo.
- **int id:** identificador (código del paciente).

7.12. Donant.java

Objeto que contiene todos los datos con la información que necesita el Coordinador en transplantes y el Especialista sobre el donante. El objeto consta de los siguientes elementos:

- Constructores del objeto:
 - **Donant(String Hospital, String Localitat, String Provincia, int Tipus_sang, double Pes, double Tamany, boolean[][] Antigens, Calendar Data_naix, String Organ)**
 - **Donant(long Codi, String Nom, String Cognom1, String Cognom2, String Hospital, String Localitat, String Provincia, int Tipus_sang, double Pes, double Tamany, boolean[][] Antigens, Calendar Data_naix, String Organ)**
- Atributos del objeto:
 - **long Codi:** código identificativo del donante.
 - **String Nom:** nombre del donante.
 - **String Cognom1:** primer apellido del donante.
 - **String Cognom2:** segundo apellido del donante.
 - **String Hospital:** hospital donde se encuentra el órgano.
 - **String Localitat:** localidad donde se encuentra situado el hospital.
 - **String Provincia:** provincia a la que pertenece la localidad.
 - **int Tipus_sang:** tipo de sangre del donante (A, B, AB, O).
 - **double Pes:** peso del donante (kg).
 - **double Tamany:** Tamaño del órgano del donante.
 - **boolean Antigens[][]:** tabla que indica si el donante posee o no alguno de los cuatro antígenos.
 - **Calendar Data_naix:** fecha de nacimiento del donante.
 - **String Organ:** tipo de órgano donado.

7.13. MantDonant.java

Objeto que nos sirve para realizar un mantenimiento de donantes. Consta de unas funciones para facilitarnos el manejo de los donantes. Se ha implementado mediante Streams a ficheros, pero la estructura es la adecuada para que sea fácilmente sustituible por una base de datos accediendo mediante JDBC. El objeto consta de las siguientes funciones:

- **int inicializa(String nomFichero):** función que inicializa las tablas y variables necesarias para el mantenimiento. Se le pasa como parámetros el path absoluto de fichero donde se encuentran los datos y retorna un número negativo si se ha producido algún error.
- **int finaliza():** finaliza el mantenimiento escribiendo los datos temporales al fichero.
- **void selectReset():** Resetea una selección, de donantes. Esta función se usa cuando seguidamente se va a llamar a la función *selectNext()*.
- **Donant selectNext():** Función que se usa para seleccionar el siguiente donante en la lista total de donantes.
- **Donant selectDonant(long codigo):** función que retorna el donante que tiene el código que le pasamos por parámetros. En caso de que no exista retorna *null*.

- **void insertDonant(Donant donante):** inserta un nuevo donante en el fichero asignándole un código único.
- **void deleteDonant(long codigo):** borra el donante que tenga el código que le pasamos por parámetros.
- **void deleteDonant(Donant donante):** lo mismo que la anterior pero en este caso le pasamos el objeto Donant entero.
- **void updateDonant(Donant donante):** modifica el donante de la lista que tenga el mismo código que el donante que le pasamos por parámetros. Le asigna los datos del nuevo donante.
- **boolean isIniciado():** función que nos retorna si ya tenemos un fichero en uso o no.
- **int CreaFichero(String nomFichero):** función que usaremos para crear un nuevo fichero de donantes. Le pasamos como parámetro el path absoluto del nuevo fichero. Retorna un número negativo si se produce algún error.
- **int getNumDonantes():** retorna el número total de donantes que tenemos.
- **MantDonant(String fichero):** constructor al cual se le pasa el path absoluto de un fichero de donantes para que lo cargue.
- **MantDonant():** constructor vacío del objeto.

7.14. Pacient.java

Objeto que contiene todos los datos con la información que necesita el Coordinador en transplantes y el Especialista sobre un paciente. El objeto consta de los siguientes elementos:

- Constructores del objeto:
 - **Pacient(String Hospital, String Localitat, String Provincia, int Tipus_sang, double Pes, double Tamany, boolean[][] Antigens, Calendar Data_naix, Calendar Data_esp, String Organ)**
 - **Pacient(long Codi, String Nom, String Cognom1, String Cognom2, String Hospital, String Localitat, String Provincia, int Tipus_sang, double Pes, double Tamany, boolean[][] Antigens, Calendar Data_naix, Calendar Data_esp, String Organ)**
- Atributos del objeto:
 - **long Codi:** código identificativo del paciente.
 - **String Nom:** nombre del paciente.
 - **String Cognom1:** primer apellido del paciente.
 - **String Cognom2:** segundo apellido del paciente.
 - **String Hospital:** hospital donde se encuentra el órgano.
 - **String Localitat:** localidad donde se encuentra situado el hospital.
 - **String Provincia:** provincia a la que pertenece la localidad.
 - **int Tipus_sang:** tipo de sangre del paciente (A, B, AB, O).
 - **double Pes:** peso del paciente (kg).
 - **double Tamany:** Tamaño del órgano del paciente.
 - **boolean Antigens[][]:** tabla que indica si el paciente posee o no alguno de los cuatro antígenos.
 - **Calendar Data_naix:** fecha de nacimiento del paciente.

- **Calendar Data_esp**: fecha de entrada en la lista de espera del paciente.
- **String Organ**: tipo de órgano paciente.

7.15. MantPacient.java

Objeto que nos sirve para realizar un mantenimiento de pacientes. Consta de unas funciones para facilitarnos el manejo de los pacientes. Se ha implementado mediante Streams a ficheros, pero la estructura es la adecuada para que sea fácilmente sustituible por una base de datos accediendo mediante JDBC. El objeto consta de las siguientes funciones:

- **int inicializa(String nomFichero)**: función que inicializa las tablas y variables necesarias para el mantenimiento. Se le pasa como parámetros el path absoluto de fichero donde se encuentran los datos y retorna un número negativo si se ha producido algún error.
- **int finaliza()**: finaliza el mantenimiento escribiendo los datos temporales al fichero.
- **void selectReset()**: Resetea una selección, de pacientes. Esta función se usa cuando seguidamente se va a llamar a la función *selectNext()*.
- **Pacient selectNext()**: Función que se usa para seleccionar el siguiente paciente en la lista total de pacientes.
- **Pacient selectPacient(long codigo)**: función que retorna el paciente que tiene el código que le pasamos por parámetros. En caso de que no exista retorna *null*.
- **void insertPacient(Pacient paciente)**: inserta un nuevo paciente en el fichero asignándole un código único.
- **void deletePacient(long codigo)**: borra el paciente que tenga el código que le pasamos por parámetros.
- **void deletePacient(Pacient paciente)**: lo mismo que la anterior pero en este caso le pasamos el objeto Pacient entero.
- **void updatePacient(Pacient paciente)**: modifica el paciente de la lista que tenga el mismo código que el paciente que le pasamos por parámetros. Le asigna los datos del nuevo paciente.
- **boolean isIniciado()**: función que nos retorna si ya tenemos un fichero en uso o no.
- **int CreaFichero(String nomFichero)**: función que usaremos para crear un nuevo fichero de pacientes. Le pasamos como parámetro el path absoluto del nuevo fichero. Retorna un número negativo si se produce algún error.
- **int getNumPacientes()**: retorna el número total de pacientes que tenemos.
- **MantPacient(String fichero)**: constructor al cual se le pasa el path absoluto de un fichero de pacientes para que lo cargue.
- **MantPacient()**: constructor vacío del objeto.

7.16. agentePuntuacion.java

Objeto que se usa para el mantenimiento de puntuaciones de agentes. Consta de los siguientes elementos:

- **String nombre**: nombre del agente.
- **int numResultados**: número de puntuaciones que posee.
- **int resultados[]**: lista de las puntuaciones que le han signado.

7.17. MantResultados.java

Objeto que nos sirve para realizar un mantenimiento de las puntuaciones dadas a los agentes. Consta de unas funciones para facilitarnos el manejo de éstas. Se ha implementado mediante Streams a ficheros, pero la estructura es la adecuada para que sea fácilmente sustituible por una base de datos accediendo mediante JDBC. El objeto consta de las siguientes funciones:

- **int inicializa(String nomFichero)**: función que inicializa las tablas y variables necesarias para el mantenimiento. Se le pasa como parámetros el path absoluto de fichero donde se encuentran los datos y retorna un número negativo si se ha producido algún error.
- **int finaliza()**: finaliza el mantenimiento escribiendo los datos temporales al fichero.
- **void selectReset()**: Resetea una selección, de pacientes. Esta función se usa cuando seguidamente se va a llamar a la función *selectNext()*.
- **agentePuntuacion selectNext()**: Función que se usa para seleccionar los resultados del siguiente agente en la lista total de agentes.
- **agentePuntuacion selectPacient(String nombre)**: función que retorna los resultados del agente que tiene el nombre que le pasamos por parámetros. En caso de que no exista retorna *null*.
- **void insertAgente(agentePuntuacion agente)**: inserta un nuevo agente en el fichero.
- **void updateAgente(String nombre, int result)**: modifica el agente de la lista que tenga el mismo nombre que el que le pasamos por parámetros. Le añade la nueva puntuación que le pasamos.
- **boolean isIniciado()**: función que nos retorna si ya tenemos un fichero en uso o no.
- **int CreaFichero(String nomFichero)**: función que usaremos para crear un nuevo fichero de puntuaciones de agentes. Le pasamos como parámetro el path absoluto del nuevo fichero. Retorna un número negativo si se produce algún error.
- **MantResultados(String fichero)**: constructor al cual se le pasa el path absoluto de un fichero de puntuaciones para que lo cargue.
- **MantResultados()**: constructor vacío del objeto.

7.18. Paquete Gráfico

Este paquete consta de los módulos necesarios para proporcionar una interface gráfica, en ellos se definen las funciones necesarias para que el usuario final pueda interactuar con el sistema, de forma que sea semitransparente el funcionamiento de éste para él. El conjunto de ficheros que se han implementado para este fin son:

- **gui.java**: Fichero principal del paquete cuyo constructor llamará el agente Coordinador para mostrar la ventan principal del entorno gráfico al usuario. Consta de diferentes opciones de menú para poder interactuar con el sistema. Desde este objeto se llamará al resto de objetos que forman el paquete.
- **guiAlta.java**: Objeto que se usa para dar de alta pacientes y donantes. Consta de una interface que nos facilita la introducción necesaria de datos que se necesitan conocer para un paciente y para un donante. Consta de un constructor que es llamado por un objeto **gui** cuando el usuario acceda a la opción de *Alta*.
- **guiAviso.java**: Objeto que se usa para mostrar mensajes al usuario. Consta de un constructor al cual se le pasa el mensaje a visualizar. La ventana que llama a una ventana de aviso, se quedará deshabilitada hasta que el usuario pulse el botón de *Aceptar* en la ventana de aviso.
- **guiBuscar.java**: Este objeto es el principal para realizar las búsquedas de pacientes para un donante dado. Nos abre una ventana gráfica en la cual podemos elegir todo lo que nos hace falta para realizar la búsqueda, desde el agente especialista que queremos usar, el agente MCDM y las distancia, hasta los pesos que pondremos a cada atributo. Abre una ventana para que nos muestre el rastro de la búsqueda mientras que esta se realiza.
- **guiBuscarLog.java**: Objeto que abre una ventana en el cual se nos muestra el rastro o log de una búsqueda de pacientes que haya iniciado el usuario.
- **guiFile.java**: Objeto que se usa para que el usuario puede seleccionar el fichero que desee ya sea de pacientes, donantes o puntuaciones de agentes según cual este intentando cargar en el aplicación.
- **guiGrafico.java**: Objeto el cual se usa para mostrar los resultados de los diferentes agente MCDM de un modo gráfico mediante una gráfica de líneas y puntos. Nos da la posibilidad de seleccionar el agente que queremos ver.

7.19. Paquete Ontology

La ontología es lo que se usa para que los agentes puedan entender el contenido de los mensajes que reciben. Esta formada por un conjunto de *frames* y *slots* que se tienen que seguir cuando se crea un mensaje para que quien lo vaya a recibir pueda extraer correctamente la información. La ontología esta formada por dos partes principales:

- **Definición:** en esta fase es cuando se definen que *frames*, acciones y predicados hay en la ontología. En cada una de éstas tendremos que incluir el conjunto de *slots* que las forman y la clase que se encarga de manejarlos. El objeto en el que se incluyen estas definiciones es **MCDMOntology.java**.
- **Implementación:** para cada elemento definido, tendremos que crear una clase que implemente el método de inicialización y consulta para cada uno de los *slots* que contenga. El conjunto de ficheros java que implementan cada uno de los elementos definidos son:
 - **AlgoritmeNoImplementat.java:** implementa el error que se produce cuando a un agente MCDM se le demanda una petición que no tiene un algoritmo de decisión asociado.
 - **Atrib_objectes.java:** permite manejar los datos de los objetos a clasificar (valores de sus atributos e identificadores).
 - **Atribut.java:** permite acceder a los datos de los atributos (pesos y vocabulario).
 - **ClassesInsuficients.java:** implementa un error que se produce cuando el número de clases finales obtenidas son insuficientes para el vocabulario definido (agente ClusDM).
 - **DatosNoValidos.java:** implementa un error que se produce en el mensaje que envía el coordinador al especialista.
 - **Domain_OntologyObj.java:** implementa la interficie **OntolotyEntity** y da soporte a las variables de forma que el resto de clases la extenderán para simplificar la implementación.
 - **EspecialistaRequest.java:** implementa la acción del envío de un *request* a un agente Especialista. El conjunto de datos que se envían son el número de atributos, atributos, función de distancia elegida, número de objetos, lista de objetos a evaluar con el valor de sus atributos e identificadores y el nombre del agente MCDM al cual va dirigida la búsqueda.
 - **Grau.java:** permite el acceso a los datos de cada uno de los objetos del resultado.
 - **MassaAtributs.java:** implementación de un error que se produce cuando el número de atributos supera el máximo permitido.
 - **MassaObjectes.java:** implementación de un error que se produce cuando el número de objetos supera el máximo permitido.
 - **MCDMRequest.java:** implementa la acción del envío de un *request* a un agente MCDM. El conjunto de datos que se envían son el número de atributos, atributos con sus vocabularios, funciones de negación y valores difusos, función de distancia elegida, número de objetos y lista de objetos a evaluar con el valor de sus atributos e identificadores.
 - **Negacio.java:** permite acceder a las negaciones de cada termino del un vocabulario.
 - **PesosNoValids.java:** implementación del error que se produce cuando el valor de los pesos no es válido.
 - **Resultat.java:** da acceso a los datos de los resultados (numero de objetos, lista de estos con sus identificadores y el término del vocabulario final asignado).
 - **Terme.java:** permite acceder a los datos de cada término de un vocabulario (valor, negaciones y valores difusos).

- **Triangel_difus.java**: implementa una estructura con la tripleta de valores difusos para un termino de un vocabulario.
- **Valor_atribut.java**: permite acceder a los valores de los atributos de cada objeto.

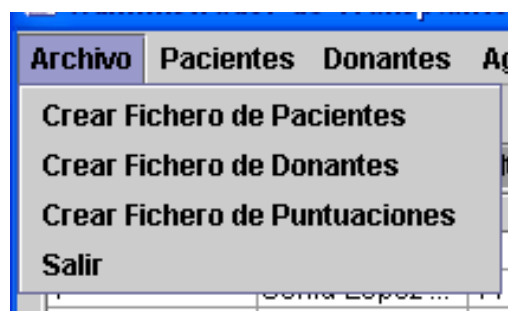
8. Manual de Usuario

En este apartado se intenta explicar como trabajar con la interfaz gráfica que muestra el Agente Coordinador de transplantes para interactuar con las personas. El programa se divide en varias partes que se detallan en los próximos sub-partados.

Codigo	Nombre y Ape...	Edad	Peso	Sangre	Organo	Tamaño	Hospital	Ciudad	Provincia
0	Miguel Alonso...	45	84.0	O	cor	34.0	Hospital Marq...	Santander	Cantabria
1	Sonia Lopez ...	44	67.3	A	cor	28.1	Hospital de S...	Reus	Tarragona
2	Mirian Soto Al...	30	63.1	B	cor	25.2	Hospital de S...	Barcelona	Barcelona
3	Jose Luis Lo...	19	60.0	A	cor	27.9	Hospital Sant...	Tarragona	Tarragona
4	Juan Alonso ...	45	84.0	AB	pulmo	34.0	Hospital Marq...	Santander	Cantabria
5	Eva Espin Sa...	19	67.2	B	cor	30.2	Hospital Ram...	Madrid	Madrid
6	Rosa Gonzal...	32	65.4	O	cor	24.9	Hospital Univ...	Sevilla	Sevilla
7	Sara Perez P...	25	63.0	A	pulmo	27.1	Hospital de S...	Valencia	Valencia
8	Jose Martin T...	39	79.8	B	pulmo	29.2	Hospital del ...	Mataro	Barcelona
9	Javier Gonzal...	15	78.1	O	cor	32.5	Hospital del ...	Ponferrada	Leon
10	Juan Carlos A...	34	85.2	AB	cor	36.1	Hospital Obis...	Teruel	Teruel
11	Mario Gomez ...	31	76.0	A	cor	33.0	Hospital Gen...	Vic	Barcelona
12	Miguel Angel ...	25	78.1	B	pulmo	32.3	Hospital Clinic	Barcelona	Barcelona
13	Maria Lopez ...	44	67.3	AB	pulmo	28.1	Hospital de S...	Reus	Tarragona
14	Teresa Espin ...	19	67.2	AB	pulmo	30.2	Hospital Ram...	Madrid	Madrid
15	Carmen Gonz...	32	65.4	O	pulmo	24.9	Hospital Univ...	Sevilla	Sevilla
16	Montse Soto ...	30	63.1	B	pulmo	25.2	Hospital de S...	Barcelona	Barcelona
17	Ana Perez Per...	25	63.0	A	cor	27.1	Hospital de S...	Valencia	Valencia
18	Jordi Martin T...	39	79.8	B	cor	29.2	Hospital del ...	Mataro	Barcelona
19	Santiago Ruiz...	25	78.1	AB	cor	32.3	Hospital Clinic	Barcelona	Barcelona
20	Alberto Segur...	45	75.9	O	pulmo	24.1	Hospital La P...	Madrid	Madrid
21	Tomas Tetua...	33	65.1	O	pulmo	31.7	Hospital Univ...	La Laguna	Tenerife
22	Alvaro Segur...	45	75.9	A	cor	24.1	Hospital La P...	Madrid	Madrid
23	Francisco Tet...	33	65.1	A	cor	31.7	Hospital Univ...	La Laguna	Tenerife
24	Gonzalo Long...	19	60.0	AB	pulmo	27.9	Hospital Sant...	Tarragona	Tarragona

8.1. Menú

8.1.1. Archivo



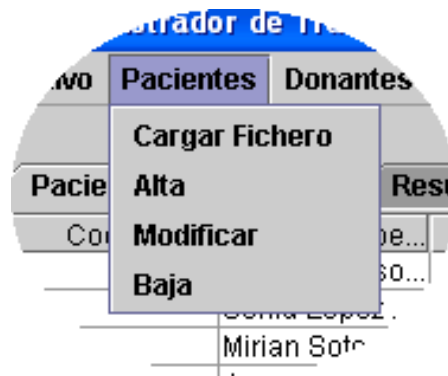
Dentro de este submenú nos encontramos con 4 acciones básicas de la aplicación:

- Crear Fichero de Pacientes: Sirve para crear un archivo de pacientes, éste será el archivo donde se guardarán todos los pacientes que demos de alta en la aplicación a partir de ese momento, además este archivo podrá ser abierto en futuras ocasiones, desde el submenú de pacientes. Al pinchar sobre esta opción

se nos abrirá una ventana en la que tendremos que indicar la ruta completa y el nombre del fichero a crear.

- Crear Fichero de Donantes: Realiza la misma función que la opción anterior, solo que en este caso con respecto a los donantes.
- Crear Fichero de Puntuaciones: Igual a que las anteriores opciones, pero en este caso refiriéndose a las puntuaciones que el usuario asigna a los distintos agentes MCDM.
- Salir: Al pinchar sobre esta opción nos saldremos de la aplicación.

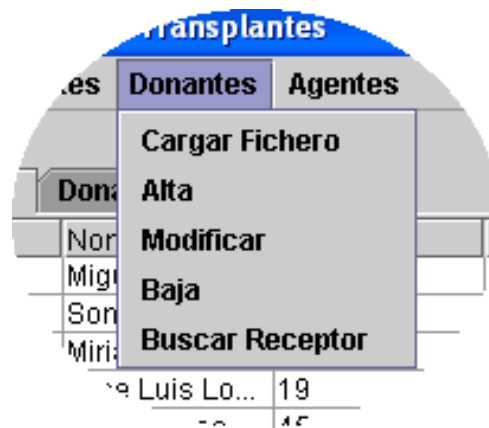
8.1.2. Pacientes



En este submenú se incluyen todas las acciones que necesitamos para manipular pacientes:

- Cargar Fichero: Esta opción se utiliza para cargar un archivo de pacientes (.dbp) que hayamos creado con anterioridad.
- Alta: Opción que usaremos para dar de alta pacientes. Según vayamos introduciendo nuevos pacientes, estos aparecerán en la tabla que contiene la pestaña de pacientes.
- Modificación: Opción que nos permite modificar el paciente que tengamos seleccionado en la tabla de pacientes.
- Baja: Al pinchar sobre está, daremos de baja, es decir, borraremos el pacientes que tengamos seleccionado en la tabla de pacientes.

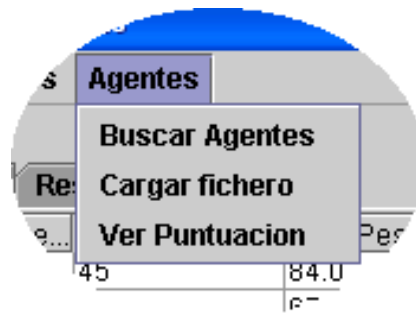
8.1.3. Donantes



En este submenú se incluyen todas las acciones que necesitamos para manipular a los donantes:

- **Cargar Fichero:** Opción que se utiliza para cargar un archivo de donantes (.dbd) que hayamos creado con anterioridad.
- **Alta:** Opción que usaremos para dar de alta donantes, de forma que según los vayamos introduciendo, estos aparecerán en la tabla que contiene la pestaña de donantes.
- **Modificación:** Opción que nos permite modificar el donante que tengamos seleccionado en la tabla de donantes.
- **Baja:** Al pinchar sobre está, daremos de baja el donante que tengamos seleccionado en la tabla de donantes.
- **Buscar Receptor:** Esta opción es la principal de la aplicación, nos sirve para buscar en toda la lista de pacientes que tenemos cargada en ese momento, a los posibles receptores para el órgano del donante que tengamos seleccionado en la tabla de donantes. Al pinchar en esta opción se nos abrirá una ventana en la que tendremos que elegir a un agente especialista en transplantes así como a un agente MCDM que realicen la ordenación, también podremos elegir el peso que queremos asignar a cada uno de los atributos así como la función de distancia que queremos utilizar (solo para el agente ClusDM). Tras pinchar en Aceptar, se nos abrirá una ventana que nos irá mostrando un log de lo que está sucediendo en el sistema. Cuando la ordenación acabe, los pacientes nos serán mostrados en orden en la tabla que contiene la pestaña de resultados.

8.1.4. Agentes



Este es el submenú que contiene las acciones referentes a los agentes que hay en el sistema:

- Buscar agentes: Opción que busca todos los agentes disponibles en el sistema, los cuales luego nos aparecerán al pinchar sobre la opción de *buscar receptor*.
- Cargar fichero: Opción que se utiliza para cargar un archivo de puntuaciones de a agentes (.dbp) que hayamos puntuado con anterioridad.
- Ver puntuaciones: Al pinchar sobre esta opción, se nos abrirá una ventana en la que se muestra un gráfico con los resultados de las puntuaciones al agente que seleccionemos.

8.2. Pestañas

El programa tiene tres pestañas, una de pacientes que contiene una tabla con los pacientes que tenemos cargados actualmente. La segunda pestaña es la de donantes, en la que al igual que la otra, se nos muestran los donantes que tenemos cargados en el sistema en este momento. La ultima pestaña (Resultados) es en la que se nos mostrará el resultado de la ordenación de los pacientes que iniciemos en la opción de menú correspondiente, además tendremos un selector para puntuar al agente que ha realizado la búsqueda.

9. Juego de pruebas y comparativa

Para realizar las pruebas usaremos los siguientes pacientes:

Paciente 0:

Nombre: Miguel Alonso Sanchez

Edad: 45

Peso: 84.0

Tipo de Sangre: O

Organo: cor

Tamaño: 34.0

Hospital: Hospital Marques de Valdecilla

Localidad: Santander

Provincia: Cantabria

Paciente 1:

Nombre: Sonia Lopez Merino

Edad: 44

Peso: 67.3

Tipo de Sangre: A

Organo: cor

Tamaño: 28.1

Hospital: Hospital de Sant Joan

Localidad: Reus

Provincia: Tarragona

Paciente 2:

Nombre: Mirian Soto Alonso

Edad: 30

Peso: 63.1

Tipo de Sangre: B

Organo: cor

Tamaño: 25.2

Hospital: Hospital de Sant Pau

Localidad: Barcelona

Provincia: Barcelona

Paciente 3:

Nombre: Jose Luis Lopez Pacios

Edad: 19

Peso: 60.0

Tipo de Sangre: A

Organo: cor

Tamaño: 27.9

Hospital: Hospital Santa Tecla

Localidad: Tarragona

Provincia: Tarragona

Paciente 4:

Nombre: Juan Alonso Sanchez
Edad: 45
Peso: 84.0
Tipo de Sangre: AB
Organo: pulmo
Tamaño: 34.0
Hospital: Hospital Marques de Valdecilla
Localidad: Santander
Provincia: Cantabria

Paciente 5:
Nombre: Eva Espin Sanchez
Edad: 19
Peso: 67.2
Tipo de Sangre: B
Organo: cor
Tamaño: 30.2
Hospital: Hospital Ramon y Cajal
Localidad: Madrid
Provincia: Madrid

Paciente 6:
Nombre: Rosa Gonzalez Martinez
Edad: 32
Peso: 65.4
Tipo de Sangre: O
Organo: cor
Tamaño: 24.9
Hospital: Hospital Universitario Valme
Localidad: Sevilla
Provincia: Sevilla

Paciente 7:
Nombre: Sara Perez Perez
Edad: 25
Peso: 63.0
Tipo de Sangre: A
Organo: pulmo
Tamaño: 27.1
Hospital: Hospital de Sagunto
Localidad: Valencia
Provincia: Valencia

Paciente 8:
Nombre: Jose Martin Teruel
Edad: 39
Peso: 79.8
Tipo de Sangre: B
Organo: pulmo
Tamaño: 29.2

Hospital: Hospital del Sagrado Corazon
Localidad: Mataro
Provincia: Barcelona

Paciente 9:
Nombre: Javier Gonzalez Valles
Edad: 15
Peso: 78.1
Tipo de Sangre: O
Organo: cor
Tamaño: 32.5
Hospital: Hospital del Bierzo
Localidad: Ponferrada
Provincia: Leon

Paciente 10:
Nombre: Juan Carlos Alberdi Vilaubi
Edad: 34
Peso: 85.2
Tipo de Sangre: AB
Organo: cor
Tamaño: 36.1
Hospital: Hospital Obispo Palanca
Localidad: Teruel
Provincia: Teruel

Paciente 11:
Nombre: Mario Gomez Viza
Edad: 31
Peso: 76.0
Tipo de Sangre: A
Organo: cor
Tamaño: 33.0
Hospital: Hospital General de Vic
Localidad: Vic
Provincia: Barcelona

Paciente 12:
Nombre: Miguel Angel Ruiz Sanchez
Edad: 25
Peso: 78.1
Tipo de Sangre: B
Organo: pulmo
Tamaño: 32.3
Hospital: Hospital Clinic
Localidad: Barcelona
Provincia: Barcelona

Paciente 13:
Nombre: Maria Lopez Merino

Edad: 44
Peso: 67.3
Tipo de Sangre: AB
Organo: pulmo
Tamaño: 28.1
Hospital: Hospital de Sant Joan
Localidad: Reus
Provincia: Tarragona

Paciente 14:
Nombre: Teresa Espin Sanchez
Edad: 19
Peso: 67.2
Tipo de Sangre: AB
Organo: pulmo
Tamaño: 30.2
Hospital: Hospital Ramon y Cajal
Localidad: Madrid
Provincia: Madrid

Paciente 15:

Nombre: Carmen Gonzalez Martinez
Edad: 32
Peso: 65.4
Tipo de Sangre: O
Organo: pulmo
Tamaño: 24.9
Hospital: Hospital Universitario Valme
Localidad: Sevilla
Provincia: Sevilla

Paciente 16:
Nombre: Montse Soto Alonso
Edad: 30
Peso: 63.1
Tipo de Sangre: B
Organo: pulmo
Tamaño: 25.2
Hospital: Hospital de Sant Pau
Localidad: Barcelona
Provincia: Barcelona

Paciente 17:
Nombre: Ana Perez Perez
Edad: 25
Peso: 63.0
Tipo de Sangre: A
Organo: cor
Tamaño: 27.1

Hospital: Hospital de Sagunto
Localidad: Valencia
Provincia: Valencia

Paciente 18:
Nombre: Jordi Martin Teruel
Edad: 39
Peso: 79.8
Tipo de Sangre: B
Organo: cor
Tamaño: 29.2
Hospital: Hospital del Sagrado Corazon
Localidad: Mataro
Provincia: Barcelona

Paciente 19:
Nombre: Santiago Ruiz Sanchez
Edad: 25
Peso: 78.1
Tipo de Sangre: AB
Organo: cor
Tamaño: 32.3
Hospital: Hospital Clinic
Localidad: Barcelona
Provincia: Barcelona

Paciente 20:
Nombre: Alberto Segura Martinez
Edad: 45
Peso: 75.9
Tipo de Sangre: O
Organo: pulmo
Tamaño: 24.1
Hospital: Hospital La Paz
Localidad: Madrid
Provincia: Madrid

Paciente 21:
Nombre: Tomas Tetuan Gomez
Edad: 33
Peso: 65.1
Tipo de Sangre: O
Organo: pulmo
Tamaño: 31.7
Hospital: Hospital Universitario de Canarias
Localidad: La Laguna
Provincia: Tenerife

Paciente 22:
Nombre: Alvaro Segura Martinez

Edad: 45
Peso: 75.9
Tipo de Sangre: A
Organo: cor
Tamaño: 24.1
Hospital: Hospital La Paz
Localidad: Madrid
Provincia: Madrid

Paciente 23:
Nombre: Francisco Tetuan Gomez
Edad: 33
Peso: 65.1
Tipo de Sangre: A
Organo: cor
Tamaño: 31.7
Hospital: Hospital Universitario de Canarias
Localidad: La Laguna
Provincia: Tenerife

Paciente 24:
Nombre: Gonzalo Lopez Pacios
Edad: 19
Peso: 60.0
Tipo de Sangre: AB
Organo: pulmo
Tamaño: 27.9
Hospital: Hospital Santa Tecla
Localidad: Tarragona
Provincia: Tarragona

Paciente 25:
Nombre: Jose Gonzalez Valles
Edad: 15
Peso: 78.1
Tipo de Sangre: O
Organo: pulmo
Tamaño: 32.5
Hospital: Hospital del Bierzo
Localidad: Ponferrada
Provincia: Leon

Paciente 26:
Nombre: Sebastian Alberdi Vilaubi
Edad: 34
Peso: 85.2
Tipo de Sangre: B
Organo: pulmo
Tamaño: 36.1
Hospital: Hospital Obispo Palanca

Localidad: Teruel

Provincia: Teruel

Paciente 27:

Nombre: Marcos Gomez Viza

Edad: 31

Peso: 76.0

Tipo de Sangre: A

Organo: pulmo

Tamaño: 33.0

Hospital: Hospital General de Vic

Localidad: Vic

Provincia: Barcelona

El donante con el que se han realizado las pruebas es el que sigue:

Donante 0:

Nombre: Fernando Rodriguez Fernandez

Edad: 24

Peso: 78.0

Tipo de Sangre: O

Organo: cor

Tamaño: 32.4

Hospital: Hospital Clinic

Localidad: Barcelona

Provincia: Barcelona

Prueba 1:

Pesos:

Distancia:	17%
Peso:	17%
Tamaño:	17%
Antigenos:	17%
Edad:	16%
Espera:	16%

Método ClusDM:

optim	19
bo	9
favorable	11
adequat	18,5
factible	1
desfavorable	10,10,23
bastant_dolent	17

Método 2-tupla:

bo	19
recomenable	9

acceptable	11,5,18
No_recomenable	1,10,0,23,17

Método FuzzySets:

Molt_bo	9,19,11
Recomenable	18,5,1
No_recomenable	0,23,17
Malo	10

Prueba 2:

Pesos:

Distancia:	30%
Peso:	14%
Tamaño:	14%
Antigenos:	14%
Edad:	14%
Espera:	14%

Método ClusDM:

optim	19
bastant_bo	9
favorable	11
factible	18
flux	5,1
desfavorable	23,0,10
bastant_dolent	17

Método 2-Tupla:

bo	19
Aceptable	9,11,18
No_recomenable	5,1,10,0,23
Malo	17

Método FuzzySets:

Molt_bo	9,19
Bo	11
Recomenable	5,18
Aceptable	0
No_recomenable	1
Malo	10,17,23

Prueba 3:

Pesos:

Distancia:	14%
Peso:	30%

Tamaño:	14%
Antigenos:	14%
Edad:	14%
Espera:	14%

Método ClusDM:

optim	19
molt_bo	9
bastant_bo	11
adequat	18,5
factible	1
fluix	10,0
bastant_dolent	23,17

Método 2-tupla:

bo	19
Recomenable	9
Aceptable	11,18,5
No_recomenable	1,10,0,23
Malo	17

Método FuzzySets:

Molt_bo	9,11,19
Bo	18
Recomenable	5,1
Aceptable	0
Malo	10,23,17

Prueba 4:

Pesos:

Distancia:	14%
Peso:	14%
Tamaño:	30%
Antigenos:	14%
Edad:	14%
Espera:	14%

Método ClusDM:

optim	19
molt_bo	9
bastant_bo	11
adequat	18,5
factible	23,1
fluix	10,0
bastant_dolent	17

Método 2-tupla:

bo	19
Recomenable	9
Aceptable	11,5,18
No_recomenable	1,10,23,0,17

Método FuzzySets:

Molt_bo	9,11,19
Bo	23
Recomenable	18,5,1
Aceptable	0
Malo	10,17

Prueba 5:

Pesos:

Distancia:	14%
Peso:	14%
Tamaño:	14%
Antigenos:	30%
Edad:	14%
Espera:	14%

Método ClusDM:

optim	19
molt_bo	9
adequat	5,1
fluix	0,11
desfavorable	18
bastant_dolent	17,10
dolent	23

Método 2-tupla:

bo	19
Recomenable	9
Aceptable	5,1,11
No_recomenable	18,0,10,17,23

Método FuzzySets:

Molt_bo	9,19,11
Bo	1,18
Aceptable	5,17
No_recomenable	10,0
Malo	23

Prueba 6:

Pesos:

Distancia: 14%
 Peso: 14%
 Tamaño: 14%
 Antigenos: 14%
 Edad: 30%
 Espera: 14%

Método ClusDM:

optim 19
 bo 9
 favorable 11,5
 factible 18
 fluix 17,10
 desfavorable 23,1
 bastant_dolent 0

Método 2-tupla:

bo 19
 Recomendable 9
 Aceptable 11,5
 No_recomenable 18,17,10,1,23,0

Método FuzzySets:

Molt_bo 9,19,11,5
 Bo 18,17
 Recomendable 1
 Aceptable 0
 No_recomenable 23
 Terrible 10

Prueba 7:

Pesos:

Distancia: 14%
 Peso: 14%
 Tamaño: 14%
 Antigenos: 14%
 Edad: 14%
 Espera: 30%

Método ClusDM:

optim 19
 bo 9
 adequat 11,5
 factible 18
 fluix 10,1
 desfavorable 0
 bastant_dolent 23,17

Método 2-tupla:

bo	19
Recomenable	9
Aceptable	5,11,18
No_recomenable	10,1,0,23,17

Método FuzzySets:

Molt_bo	9,19,11
Aceptable	1,18
No_recomenable	5
Malo	0,23,17
Terrible	10

Tras la realización de las distintas pruebas, se puede observar que los tres métodos se han mantenido medianamente constantes en los resultados finales. Si comparamos los resultados que nos proporcionan los tres métodos observamos que ClusDM y 2-tupla se aproximan más en la valoración de los pacientes, mientras que FuzzySets se parece a los dos anteriores pero es el que más difiere principalmente en cuales son los peores pacientes. Cuando cambiamos los pesos de los atributos, se observa que aparecen nuevos términos cualitativos en los tres métodos, en consecuencia de los cambios que se producen en los resultados, pero en líneas generales los tres métodos mantienen cual es el paciente más recomendable.

Además de estas pruebas se han realizado otras en las que se comprobaba la tolerancia de fallos como por ejemplo enviar pesos inválidos, recibiendo la siguiente respuesta en la ventana de log:

```
MCDM refusat
La rao del refus es: Pesos-no-valids
```

También se han probado casos extremos, por ejemplo poner el 100% del peso en el atributo “distancia”, obteniendo los siguientes resultados:

Método ClusDM:

```
Especialista fallado
La razon del fallo es: Classes-insuficients
```

Método 2-tupla:

bo	19
Recomenable	18,11
No_recomenable	1
Terrible	10,0,17,5,9,23

Método FuzzySets:

Molt_bo	19
Terrible	18,11,1,10,0,17,5,9,23

11. Conclusiones y Valoración Personal

Tras la implementación de este proyecto y gracias a la diversa documentación consultada, he observado que los agentes y suponen una gran herramienta para la Inteligencia Artificial, debido a las características que hemos ido enumerando a lo largo de este documento. El simple hecho de poder aplicar una tecnología tan novedosa como esta a algo tan importante como los trasplantes de órganos, me hace pensar que las aplicaciones basadas en sistemas multiagente pueden llegar en un futuro a solucionar de una forma rápida y segura problemas que hoy en día llevan mucho tiempo, trabajo y dinero.

Este proyecto forma parte de otro mucho más grande que será el que pueda tener una aplicación global de cara a los trasplantes. Con la implementación del mismo he intentado colaborar en lo posible con el objetivo marcado por mi tutora Aïda Valls y por el grupo de sistemas multiagente de la universidad (GruSMA).

Otro punto que me ha supuesto ampliar los conocimientos adquiridos en la carrera, ha sido el aprender diferentes métodos de ayuda a la toma de decisiones multicriterio (MCDM) y trabajar con conjuntos difusos y términos cualitativos para comunicar el sistema con los humanos de una forma mucho mas comprensible para la persona que tenga que interactuar con el sistema.

Por último quiero agradecer a mi tutora Aïda Valls el apoyo y ayuda que me ha dado en todo momento, puedo decir que gracias a esto y a su constancia me ha sido mucho más sencillo realizar este proyecto, el cual, sin su inestimable ayuda no creo que hubiese podido llevar a término en el plazo de tiempo en que lo he hecho. GRACIAS!!!

12. BIBLIOGRAFIA

[Aldea et al., 2001]

Aldea, A., López, B., Moreno, A., Riaño, D., Valls, A. (2001), A multi-agent system for organ transplant co-ordination, 8th. *European Conference on Artificial Intelligence in Medicine*, Portugal, 2001. Publicat també a *Lecture Notes in Artificial Intelligence*, 2101, pp. 174-383.

[Banzai]

Banzai, Grup de recerca en Intel·ligència Artificial, DEIM, Universitat Rovira i Virgili, plana web: <http://www.etse.urv.es/recerca/banzai>

[Bocio, 2000]

Bocio, J. (2000), *Diseño e Implementación de un Sistema Multi-Agente para la Gestión de Equipos Médicos*, Projecte Final de Carrera, Enginyeria Tècnica en Informàtica de Sistemes, Universitat Rovira i Virgili

[FIPA]

Foundation for Intelligent Physical Agents, web page: <http://www.fipa.org>

[FIPA, 2001]

Foundation for Intelligent Physical Agents (2001), *FIPA Device Ontology*, FIPA

[FIPA, 2000a]

Foundation for Intelligent Physical Agents (2000), *FIPA ACL Message Structure*, FIPA

[FIPA, 2000b]

Foundation for Intelligent Physical Agents (2000), *FIPA Agent Management Specification*, FIPA

[FIPA, 2000c]

Foundation for Intelligent Physical Agents (2000), *FIPA SL Content Language Specification*, FIPA

[FIPA, 1998a]

Foundation for Intelligent Physical Agents (1998), *Agent Communication Language*, FIPA

[FIPA, 1998b]

Foundation for Intelligent Physical Agents (1998), *Agent Management*, FIPA

[FIPA, 1998c]

Foundation for Intelligent Physical Agents (1998), *FIPA97 Developers Guide*, FIPA

[Godo i Torra, 2000]

Godo, L., Torra, V. (2000), On aggregation operators for ordinal qualitative information, *IEEE Transactions on Fuzzy Systems*, vol.8, n.2, pp.143-154

[GruSMA]

GruSMA, Grup de Sistemes Multi-Agent, DEIM, Universitat Rovira i Virgili, plana web: <http://www.etse.urv.es/recerca/banzai/toni/MAS>

[GruSMA, 2000a]

GruSMA (2000), *Agents i Sistemes Multi-Agents*, Universitat Rovira i Virgili

[GruSMA, 2000b]

GruSMA (2000), *Especificación FIPA de la Gestión de Agentes*, Universitat Rovira i Virgili

[GruSMA, 2000c]

GruSMA (2000), *Jade 1.3*, Universitat Rovira i Virgili

[GruSMA, 2000d]

GruSMA (2000), *Lenguaje de comunicación entre agentes FIPA*, Universitat Rovira i Virgili

[Herrera&Martínez, 2000a]

Herrera, F., Martínez, L. (2000), A 2-tuple fuzzy linguistic representation model for computing with words, *IEEE Transactions on Fuzzy Systems* vol. 8, n. 6, pp. 746-752

[Herrera&Martínez, 2000b]

Herrera, F., Martínez, L., An Approach for Combining Numerical and Linguistic Information based on the 2-tuple fuzzy linguistic representation model in Decision Making, *International Journal of Uncertainty, Fuzziness and Knowledge -Based Systems*, vol.8:5 pp. 539-562 (2000)

[Herrera et al., 2000]

Herrera, F., Herrera-Viedma, E., Martínez, L., A fusion method for multi-granularity linguistic term sets in decision making , *Fuzzy Sets and Systems*, vol.114, pp.43-58 (2000)

[Isern, 2001]

Isern, D. (2001), *JADE: Conceptes fonamentals*, apunts GruSMA, Universitat Rovira i Virgili

[Isern, 2000]

Isern, D. (2000), *Desenvolupament de Sistemes Multiagent: JADE*, apunts GruSMA, Universitat Rovira i Virgili

[Isern, 1999]

Isern, D. (1999), *Avaluació d'entorns de desenvolupament de sistemes Multiagent*, Projecte Final de Carrera, Enginyeria Tècnica en Informàtica de Sistemes, Universitat Rovira i Virgili

[Isern et al., 2001]

Isern, D., Moreno, A., Valls, A. (2001), *Desenvolupament de sistemes Multi-Agent en JADE*, Report de Recerca DEIM-RR-01-003, Dept. Enginyeria Informàtica i Matemàtiques, Universitat Rovira i Virgili

[ONT]

Organización Nacional de Trasplantes, Ministerio de Sanidad y Consumo,
<http://www.msc.es/ont/esp/home.htm>

[Ribes, 2000]

Ribes, A. (2000), *SMA d'ajuda a la gestió dels trasplants d'òrgans entre hospitals. Logística del transport dels òrgans*, Projecte Final de Carrera, Enginyeria Tècnica en Informàtica de Sistemes, Universitat Rovira i Virgili

[Sánchez, 2001]

Sánchez, D. (2001), *Un Sistema Multi-Agent d'ajuda a l'assignació d'òrgans en trasplantaments. El mètode ClusDM amb criteris lingüístics*, Projecte Final de Carrera, Enginyeria Tècnica en Informàtica de Sistemes, Universitat Rovira i Virgili

[Torra, 1994]

Torra, V., (1994) Negation functions based semantics for ordered linguistic labels, *International Journal of Intelligent Systems*, vol.11, pp.975-988

[Valls, 1995]

Valls, A. (1995), *Sedàs, Mòdul general de classificació de dominis poc estructurats*, Projecte Final de Carrera, Enginyeria Tècnica en Informàtica de Gestió, Universitat Rovira i Virgili

[Valls et al., 2001]

Valls, A., Moreno, A., Alonso, H., Sánchez, D. (2001), A Multi-Agent System to Support the Assignment of Transplantable Organs, Proceedings of the Student Session at the joint *European Agent Systems Summer School (EASSS)* and the *Advanced Course on Artificial Intelligence (ACAI)*, pp. 200-207, ISBN: 80-01-02387-7, Prague, Czech Republic

[Valls i Torra, 2000]

Valls, A., Torra, V. (2000), Using classification as an aggregation tool in MCDM, *Fuzzy sets and systems*, vol.115, n.1, pp.159-168, número especial dedicat a *Soft Decision Analysis*

[Valls, 2002]

Valls, A., *ClusDM: A multiple criteria decision making method for heterogeneous data sets*, Ph.D. Thesis, Universitat Politècnica de Catalunya (2002)

[Vincke, 1992]

Vincke P. (1992), *Multicriteria Decision-Aid*, John Wiley & Sons, ISBN 0-471-93184-5

[Weiss, 1999]

Weiss, G., (1999) (Ed.) *Multiagent Systems. A modern approach to distributed artificial intelligence*, MIT Press.

[Yager, 1998]

Yager, R.R. (1998), Fusion of ordinal information using weighted median aggregation, *International Journal of Approximate Reasoning*, vol.18, pp.35-52

[Zadeh, 1968]

Zadeh, L.A. (1968), Fuzzy Sets, *Information Control*, vol. 8, pp. 338-358