Universitat Politècnica de València
Departamento de Sistemas Informáticos y Computación

Tesis Doctoral

Diseño de organizaciones virtuales ubícuas utilizando desarrollo dirigido por modelos

Presentada por:
Jorge Luis Agüero Medina

Dirigida por :
Dr. Vicente Julián Inglada
Dr. Carlos Carrascosa Casamayor
Dr. Miguel Rebollo Pedruelo

Noviembre 2014

A mi familia, en especial
a mi esposa Marlen y a mis hijos

Índice general

Índice de figuras

Índice de tablas

Agradecimientos

Quiero agradecer muy especialmente a mi director de tesis Vicente Julián por su dedicación y apoyo en todo este tiempo en España y en Venezuela, e igualmente a mis dos co-directores Carlos Carrascosa y Miguel Rebollo por toda su atención y tiempo. A todos gracias, por las amenas discusiones en las innumerables reuniones realizadas cada semana en el despacho de Carlos.

También quiero agradecer a todos los miembros del grupo de investigación de Tecnología Informática e Inteligencia Artificial (GTI-IA), un conjunto de personas que me abrieron amablemente sus puertas al llegar a España.

XVI

Resumen

Hoy en día los avances en la miniaturización de sistemas electrónicos han impulsado el desarrollo de dispositivos o artefactos que incorporan capacidades computacionales y de comunicación. Estos dispositivos pueden proveer de una serie de servicios en diferentes entornos gracias a su tecnología empotrada, como por ejemplo: reconocimiento de personas, localización de usuarios en un entorno, ajuste automático de la temperatura e iluminación de un entorno, etc. Este potencial de procesamiento y comunicación, está permitiendo crear nuevas aplicaciones distribuidas, donde el rol principal no lo tiene el computador personal, sino los diferentes dispositivos empotrados en el entorno: sensores, interfaces, actuadores, teléfonos móviles, etc., Esto ha generado novedosas áreas de aplicación como: Internet de Cosas (Internet of Things), Computación Móvil, Redes de Sensores, Sistemas Ubícuos, Inteligencia Ambiental, etc.

Estos avances han conducido al desarrollo de un nuevo paradigma, *computación orientada a la interacción*, es decir, la computación ocurre a través de los actos de comunicación entre las entidades. Por lo tanto, es lógico pensar que este paradigma requiere, desde un punto de vista de diseño, el desarrollo de aplicaciones en diferentes plataformas de software y de hardware, debido a lo heterogéneo de los sistemas de computación, lenguajes, sistemas operativos, y objetos (dispositivos físicos: sensores, actuadores, interfaces, etc.) dispersos en el entorno. Dicha heterogeneidad presente en los sistemas ubícuos, representa todo un reto a la hora de diseñarlos.

La ingeniería de software basada en sistemas multi-agente, en particular, los sistemas multi-agente abiertos (como las Organizaciones Virtuales), tiene la capacidad de abordar los retos al diseñar sistemas ubícuos. A ello hay que unir, que varias metodologías de desarrollo de software han adoptado el enfoque "dirigido por modelos" (*model-driven*) para realizar el análisis y el diseño del software. Dicho enfoque puede ser adoptado en los sistemas multi-agente, para mejorar el proceso de desarrollo y la calidad del software basado en agentes. Así mismo, el desarrollo dirigido por modelos proporciona un soporte apropiado para abordar este tipo de sistemas, ya que nos permite el uso de modelos como principal elemento abstracto para el diseño del sistema, por medio de la interconexión de un

conjunto de componentes visuales.

En este trabajo proponemos el desarrollo de sistemas ubícuos utilizando una organización virtual, creando una *Organización Virtual Ubícua*, la cual es diseñada usando el enfoque de desarrollo dirigido por modelos. De forma más detallada, este trabajo presenta tres propuestas. La primera, presenta un conjunto de meta-modelos para diseñar una *Organización Virtual Ubícua*, llamado πVOM, que utiliza conceptos generales que se abstraen de las metodologías y de las plataformas de agentes, lo permite diseñar aplicaciones utilizando abstracciones generales de alto nivel, evitando los detalles de implementación de bajo nivel. Así mismo, se presentan dos modelos de transformaciones, que permiten obtener el modelo de implantación de la organización (con los agentes, entidades y dispositivos), por medio de transformaciones semi-automáticas dadas por la metodología *model-driven*, reduciendo la brecha entre las fases de diseño y de implementación para este tipo de sistema.

La segunda propuesta presenta una arquitectura de implantación que define una estructura de capas funcionales basada en servicios, que soporta la interacción de las entidades de la organización virtual. La arquitectura de implantación permite la interoperabilidad de diferentes entidades, plataformas de software y hardware, proporcionando a los miembros de la organización virtual la capacidad de administrar y controlar los dispositivos del entorno (del sistema ubícuo).

La tercera propuesta presenta una plataforma de ejecución de agentes empotrados llamada Andromeda, que permite ejecutar agentes empotrados sobre sistema operativo *Android* que cumplen con el modelo de agente de πVOM. Los agentes en Andromeda pueden acceder a los dispositivos del entorno, tal y como sucede en los sistemas ubícuos.

Las propuestas presentadas fueron evaluadas empíricamente con dos ejemplos, que permiten mostrar sus bondades.

Resum

Hui en dia els avanços en la miniaturització de sistemes electrònics han impulsat el desenvolupament de dispositius o artefactes que incorporen capacitats computacionals i de comunicació. Aquests dispositius poden proveir d'una sèrie de servicis en diferents entorns gràcies a la seua tecnologia encastada, com per exemple: reconeixement de persones, localització d'usuaris en un entorn, ajust automàtic de la temperatura i il · luminació d'un entorn, etc. Aquest potencial de processament i comunicació, està permetent crear noves aplicacions distribuïdes, on el rol principal no ho té el computador personal, sinó els diferents dispositius encastats en l'entorn: sensors, interfícies, actuadors, telèfons mòbils, etc., Açò ha generat noves àrees d'aplicació com: Internet de les Coses, Computació Mòbil, Xarxes de Sensors, Sistemes Ubics, Intel · ligència Ambiental, etc.

Aquests avanços han conduït al desenvolupament d'un nou paradigma, *computació orientada a la interacció*, és a dir, la computació ocorre a través dels actes de comunicació entre les entitats. Per tant, és lògic pensar que aquest paradigma requerix, des d'un punt de vista de disseny, el desenvolupament d'aplicacions en diferents plataformes de programari i de maquinari, a causa de l'heterogeni dels sistemes de computació, llenguatges, sistemes operatius, i objectes (dispositius físics: sensors, actuadors, interfícies, etc.) dispersos en l'entorn. L'heterogeneïtat present en els sistemes ubics, representa tot un repte a l'hora de dissenyar-los.

L'enginyeria de software basada en sistemes multi-agent, en particular, els sistemes multi-agent oberts (com les Organitzacions Virtuals), té la capacitat d'abordar els reptes al dissenyar sistemes ubics. A això cal unir, que diverses metodologies de desenvolupament de programari han adoptat l'enfocament "dirigit per models" (*model-driven*) per a realitzar l'anàlisi i el disseny del programari. L'enfocament pot ser adoptat en els sistemes multi-agent, per a millorar el procés de desenvolupament i la qualitat del programari basat en agents. Així mateix, el desenvolupament dirigit per models proporciona un suport apropiat per a abordar aquest tipus de sistemes, ja que ens permet l'ús de models com principal element abstracte per al disseny del sistema, per mitjà de la interconnexió d'un conjunt de components visuals.

En aquest treball proposem el desenvolupament de sistemes ubics utilitzant

una organització virtual, creant una *Organització Virtual Ubicua*, la qual és dissenyada usant l'enfoc de desenvolupament dirigit per models. De forma més detallada, aquest treball presenta tres propostes. La primera, presenta un conjunt de meta-models per a dissenyar una *Organització Virtual Ubicua*, que és diu πVOM, el cual utilitza conceptes generals que s'abstrauen de les metodologies i de les plataformes d'agents, permet dissenyar aplicacions utilitzant abstraccions generals d'alt nivell, evitant els detalls d'implementació de baix nivell. Així mateix, es presenten dos models de transformacions, que permeten obtindre el model d'implantació de l'organització (amb els agents, entitats i dispositius), per mitjà de transformacions semiautomàtiques donades per la metodologia *model-driven*, reduint la bretxa entre les fases de disseny i d'implementació per a aquest tipus de sistema.

La segona proposta presenta una arquitectura d'implantació que definix una estructura de capes funcionals basada en servicis, que suporta la interacció de les entitats de l'organització virtual. L'arquitectura d'implantació permet la interoperabilitat de diferents entitats, plataformes de programari i maquinari, proporcionant als membres de l'organització virtual la capacitat d'administrar i controlar els dispositius de l'entorn (del sistema ubic).

La tercera proposta presenta una plataforma d'execució d'agents encastats que es diu ANDROMEDA, la cual permet executar agents encastats sobre el sistema operatiu *Android* que complixen amb el model d'agent de πVOM. Els agents en ANDROMEDA poden accedir als dispositius de l'entorn, tal com succeïx en els sistemes ubics.

Les propostes presentades van ser avaluades empíricament amb dos exemples, que permeten mostrar les seues bondats.

Abstract

Nowadays the advances in miniaturization of electronic systems have driven development artifacts or devices incorporating computing and communication capabilities. These devices can provide to users (such as the people at home, workers in the office, users in a transport system, etc.) a range of services in different environments, thanks to its embedded technology. This potential for processing and communication, is allowing to create new distributed applications, where the personal computer does not have the main role, but the different devices embedded in the environment: sensors, interfaces, actuators, mobile phones, etc. This has generated novel application areas such as Internet of Things, Mobile Computing, Sensor Networks, Ubiquitous Systems, Ambient Intelligence, etc.

These advances have led to the development of a new paradigm *interaction oriented computing*, namely, computation occurs through acts of communication between entities. Therefore, it is easy to think that this paradigm requires from a design point of view, the development of applications on different software and hardware platforms, due to the heterogeneity of computer systems, languages, operating systems, and objects (physical devices: sensors, actuators, interfaces, etc.) dispersed in the environment.

However, Software Engineering-based Multi-Agent Systems, in particular Open Multi-Agent Systems (Virtual Organizations) have the ability to address these challenges. Moreover, several software development methodologies have adopted a *model-driven* approach for analysis and design. A similar approach can be adopted in multi-agent systems to improve the development process and the quality of agents-based software. The model-driven development provides an appropriate support for developing such systems since it allows us to use abstract models as main elements in the design of the system through interconnection of a set of visual components.

In this work we propose the development of ubiquitous systems using a virtual organization, creating a *Ubiquitous Virtual Organization*, which is designed using the approach of model-driven development. In more detail, this work presents three proposals. The first presents a set of meta-models to design a *Ubiquitous Virtual Organization*, called πVOM, using concepts abstracted from general met-

hodologies and platforms agents, which allows you to design applications using high-level abstractions general, avoiding implementation details of low level. Subsequently, two transformation models are presented. These transformations can generate the implantation model of the system (agents, institutions and devices), through semi-automatic transformations given by *model-driven* methodology, reducing the gap between design and implementation phases for this type of system.

The second proposal presents an architecture for implementation that structure defining a service-based functional layers, which supports the interaction entities of the virtual organization. The architecture allows interoperability implementation different entities, software and hardware platforms, providing members virtual organization the ability to manage and control devices environment (the ubiquitous system).

The third proposal presents an execution platform for embedded agents called ANDROMEDA, allowing agents to execute embedded operating system *Android* that meet the agent model πVOM. Agents in ANDROMEDA can accessing the devices in the environment, as occurs in the ubiquitous systems.

The proposals were evaluated empirically with two examples, that display their kindness.

1

Introducción

Los avances en nuevas tecnologías que se basan principalmente en Internet y la Web, como el comercio electrónico, la computación móvil/ubícua o las redes sociales, demuestran la necesidad de desarrollar aplicaciones distribuidas con algunas capacidades de inteligencia[140, 130, 149]. Estos avances han conducido al desarrollo de un nuevo paradigma: *computación orientada a la interacción*, es decir, una computación basada en la interacción entre entidades, donde la computación ocurre a través de los actos de comunicación entre entidades, convirtiéndose la computación en una actividad inherentemente *social*[108, 158, 157]. Esto implica que las capacidades computacionales son ofrecidas y solicitadas por entidades que entran y salen del sistema.

Para cumplir con estos avances, este nuevo paradigma requiere que la tecnología utilizada tenga características de interacción entre entidades independientes y también inteligencia, con las habilidades para adaptarse, coordinarse y organizarse[109, 184, 131, 129]. La *Organización Virtual* (VO, por sus siglas en inglés Virtual Organization) es un enfoque particularmente prometedor para el soporte y apoyo a este paradigma y que puede ser utilizado como un marco regulatorio para la coordinación, la comunicación, y la interacción entre las diferentes entidades computacionales[38, 155].

Las *Organizaciones Virtuales* están formadas por conjuntos de entidades (generalmente individuos e instituciones) que tienen la necesidad de coordinar recursos y servicios a través de la frontera que limita la organización[89, 38]. Además, las VO son sistemas abiertos formados por la agrupación y la colaboración de entidades heterogéneas, que tienen una estructura y funciones separadas, las cuales deben ser definidas para especificar su comportamiento dentro de la organización. El enfoque de las organizaciones ha sido empleado como paradigma para desarrollar sistemas multi-agente (MAS, por sus siglas en inglés Multi-Agent Systems),

y entre las investigaciones más relevantes podemos mencionar a: SODA[154], Electronic Institutions[80], OperA[71], OMNI[72], THOMAS[19], GORMAS[22] y PANGEA[205]. Las organizaciones permiten modelar sistemas desde un alto nivel de abstracción. Ellas incluyen la integración de las perspectivas institucionales e individuales y también permiten la adaptación dinámica a los cambios de los modelos organizacionales y del entorno[39, 82]. La organización puede describir los principales aspectos de una sociedad que está basada en diferentes puntos de vista, tales como: *Estructura*, *Funcionalidad*, *Normas*, *Interacciones*, y el *Entorno*[22, 67].

Estas sociedades (las organizaciones) requieren altos niveles de interoperabilidad para integrar diversos sistemas de información con el fin de compartir conocimientos y facilitar la colaboración entre las organizaciones. Así, la organización necesita emplear componentes básicos de software que ayuden a un rápido desarrollo y a una fácil composición de aplicaciones distribuidas, incluso en entornos heterogéneos, donde los componentes puedan ser fácilmente y de forma cooperativa integrados en otras aplicaciones para crear procesos flexibles y dinámicos. Estos niveles de flexibilidad y de cooperación entre los diferentes componentes de software es logrado usando lo que se conoce como *Ingeniería de Software Orientada a Agente*[115, 106, 148, 128] (AOSE, por sus siglas en *Agent-Oriented Software Engineering*).

La Ingeniería de Software basada en sistemas multi-agente es una tecnología con aplicaciones muy importantes en Sistemas Distribuidos e Inteligencia Artificial [114, 130, 89, 131]. Los MAS, dan soporte a todos esos desarrollos, y permiten la creación de plataformas de agentes muy heterogéneos, donde los agentes trabajan juntos a través de diferentes interacciones para dar soporte a tareas más complejas de forma colaborativa y dinámica[106, 128]. Una de las alternativas para proporcionar esas tareas complejas es considerar la noción de sistemas abiertos, que están compuestos de grupos de agentes cooperativos y heterogéneos, los cuales trabajan con objetivos locales o individuales para cumplir con los objetivos globales.

Sin embargo, las metodologías existentes en MAS proponen variados modelos, adecuados a los diferentes dominios de aplicación. Cada metodología y plataforma de MAS tiene sus propias abstracciones para el modelado conceptual y su ejecución. De esta manera, los desarrolladores con cierta frecuencia necesitan la adquisición de nuevas habilidades para comprender y diseñar con estas metodologías de sistemas multi-agente. Como consecuencia de ello, el desarrollador de sistemas multi-agente desconoce el esfuerzo requerido para la creación de aplicaciones, ya que no hay acuerdo acerca de un grupo común de elementos o abstracciones que pueda ser usado en las diferentes plataformas y metodologías MAS. Por lo tanto, un reto muy importante en el diseño MAS es proporcionar herramientas eficientes que puedan ser utilizadas por usuarios no expertos.

Sintetizar un conjunto unificado de componentes de las metodologías existentes orientadas a agente es un verdadero desafío. Sin embargo, el enfoque que brinda el Desarrollo Dirigido por Modelos (MDD, por sus siglas en inglés Model-Driven Development) puede facilitar y simplificar el proceso de diseño y la calidad del software basado en agentes, ya que permite la reutilización del software[23, 75, 180, 35] y la transformación entre modelos. MDD propone básicamente la generación automática de código usando transformaciones de modelos que tienen componentes que son independientes de la plataforma. Estos modelos son traducidos o trasladados en componentes más específicos (o en código) que dependen de la plataforma de ejecución, los cuales poseen detalles específicos sobre el sistema.

En el área MAS, existen algunos investigadores orientados en tratar de formular un conjunto de modelos que orientan el proceso de desarrollo en MAS utilizando el enfoque MDD. Algunos trabajos han concentrado sus esfuerzos en la creación de un modelo unificado muy genérico para analizar y modelar las diferentes metodologías de MAS. Entre las propuestas más significativas podemos mencionar: TAO[183], FAML[34], Agent UML(AUML)[28] y AML[56], estas propuestas crean solamente una estructura conceptual (abstracciones conceptuales) para diseñar y desarrollar MAS, pero no intentan obtener el modelo de implantación del MAS que pueda ejecutarse en plataformas específicas. Otros trabajos, tales como PIM4AGENT[99] y CAFnE[113], tienen modelos unificados (menos genéricos) y generan modelos de implantación del MAS que puedan ejecutarse en diferentes plataformas. Finalmente, propuestas como PASSI[66], TROPOS[125], e INGENIAS[91], usan MDD para obtener de algunas metodologías MAS un modelo de implantación del MAS que pueda ejecutarse sólo en una plataforma específica. Sin embargo, a pesar de que algunas de las propuestas anteriores utilizan el concepto de organización en sus modelos, ninguno de ellos se centra el desarrollo de la organización como es propuesto por el enfoque de las organizaciones virtuales, donde es necesario crear implementaciones diferentes: uno para el nivel de organización y el otro para el nivel de los agentes.

Adicionalmente, cuando se requiere utilizar las propuestas anteriores para resolver problemas en computación móvil y ubícua, como por ejemplo en escenarios donde existan redes de sensores, el desarrollador se encuentra con una limitada o nula interacción de las VOs con su entorno, pocas propuestas tienen la capacidad de acceder al entorno físico o real (a través de un sensor o un actuador). Los *Sistemas Ubícuos* son algo común en nuestros días y lo serán cada vez más en los años venideros. Esto se debe al surgimiento de *nuevos objetos* (de uso cotidiano) con diferentes capacidades tecnológicas, debido a que ellos tienen incorporados diferentes dispositivos electrónicos[69]. Es importante contar con propuestas que soporten servicios basados en dispositivos físicos presentes en el entorno de una VO.

Los sistemas ubícuos es un paradigma en el cual la tecnología es virtualmente invisible en nuestro entorno. Así, es fácil pensar que este paradigma requiere, desde el punto de vista del diseñador, el desarrollo de aplicaciones en diferentes plataformas de hardware y software en función de la diversidad de objetos en el entorno. Esto plantea grandes desafíos. De esta manera, el desarrollo de sistemas ubícuos es una tarea compleja, con múltiples actores, dispositivos y diferentes ambientes hardware y computacionales; donde es difícil encontrar una visión compacta e integradora de todos los componentes. Los requerimientos de esta clase de sistemas son muy diferentes[77]. Sin embargo, algunos de ellos son básicos:

- La integración de los dispositivos externos y de los sistemas de software. Los servicios que son suministrados por el sistema ubícuo pueden ser ofrecidos por dispositivos físicos y también por los sistemas de software existentes, y es esencial que el sistema soporte ambos enfoques.
- El aislamiento de la tecnología y los dispositivos que dependen del fabricante. Con el objetivo de facilitar el desarrollo de esta clase de sistema, las características de los dispositivos que dependen del fabricante deben ser encapsuladas en funcionalidades independientes y genéricas.

Por ello, el uso de enfoques como MDD y las VOs, tienen la capacidad de soportar esos requerimientos[22, 109], ya que puede facilitar y simplificar el proceso de diseño y la calidad del software en el proceso de desarrollo de un *Sistema Ubícuo* basado en agentes. El uso de VOs puede ser aplicado en el desarrollo de agentes empotrados para sistemas ubícuos, que soportan la integración de plataformas altamente heterogéneas, donde los agentes trabajan juntos para completar tareas complejas, en una forma dinámica y colaborativa, con la habilidad de adaptarse, coordinarse y organizarse unos con otros[109]. Los agentes facilitan la orquestación e integración de los servicios y de las funcionalidades de los dispositivos físicos. El MDD permite el re-uso y la transformación entre modelos[151]. El proceso de diseño de aplicaciones es semi-automático o automático, las herramientas guían al desarrollador en el proceso de diseño, usando modelos unificados que al aplicarle las transformaciones nos permiten obtener código específico para la plataforma de implantación.

Nuestro propósito es utilizar el enfoque MDD para el diseño de *Organizaciones Virtuales Ubícuas*. En este trabajo se describe un enfoque para el desarrollo de MAS que pueda acceder al entorno físico como un *Sistema Ubícuo* y que pueda ser implementado en diferentes plataformas orientadas a la *Organización* aplicando las ideas del MDD. Es decir, este trabajo propone un enfoque para facilitar el proceso de diseño de organización virtual ubícua usando MDD. De forma más detallada, este contempla desarrollar:

- Un conjunto unificado de meta-modelos de organización virtual genéricos, los cuales fueron creados principalmente usando iterativamente una perspectiva "*bottom-up*" sobre las metodologías de agentes orientadas a la organización. Particularmente, el meta-modelo de entorno, el cual permite incorporar en el proceso de modelado los diferentes dispositivos del entorno de un sistema ubícuo, el cual está basado en la integración de las metodologías de MAS y sistemas ubícuos.

- El trabajo también se centra en la conversión de modelos, en concreto se presenta como transformar el modelo unificado de la organización virtual a dos plataformas de agentes orientadas a la organización diferentes. Estas transformaciones se proponen como ejemplo y permiten validar la utilidad de la propuesta. Las plataformas destino orientadas a organizaciones utilizadas son: THOMAS[1] [19] y E-Institutions[2][80].

- Una arquitectura de implantación por capas. Esto permite diseñar aplicaciones ubícuas usando alto niveles de abstracción (previniendo de los detalles de implementación de bajo nivel). La implantación del sistema ubícuo (una VO con agentes empotrados y dispositivos) se genera mediante el uso de transformaciones semi-automáticas.

- Finalmente, una plataforma de implementación de agentes empotrados, que permite ejecutar agentes sobre el sistema operativo *Android* de Google, utilizando los conceptos de los meta-modelos propuestos.

Este enfoque permite al desarrollador especificar un modelo de *Organización Virtual Ubícua* (utilizando los meta-modelos propuestos) y usar las reglas de transformación para convertir los meta-modelos unificados en modelos específicos de las plataformas de ejecución del MAS. De esta manera, un desarrollador puede diseñar e implementar con mayor facilidad organizaciones virtuales ubícuas, reduciendo la brecha entre el diseño y las fases de implementación.

1.1. Objetivos

Este trabajo tiene como objetivo general el trasladar las técnicas MDD hacia el diseño de sistemas basados en agentes. El propósito es aplicar un enfoque MDD en el diseño de agentes: iniciando el diseño del agente utilizando componentes o estructuras conceptuales (los modelos) para luego por medio de transformaciones

[1] http://users.dsic.upv.es/grupos/ia/sma/tools/Thomas
[2] http://e-institutions.iiia.csic.es

obtener el código ejecutable del mismo, de una forma que sea fácil y transparente para el usuario.

La idea básica de este trabajo es diseñar agentes u organizaciones de agentes usando modelos o conceptos abstractos olvidándose de los detalles de implementación y de cualquier detalle de la plataforma de ejecución. Posteriormente, a través de transformaciones, generar el código del agente usando los detalles específicos de la plataforma donde será ejecutado. Esto permitirá que un programador desarrolle MAS de una forma menos compleja, reduciendo la brecha entre el diseño de agentes y su implementación.

Entre las plataformas contempladas están las plataformas ubícuas y empotradas, que permiten a los agentes ejecutarse en computadores de recursos limitados. Además permite a los agentes interactuar y administrar su ambiente a través de dispositivos físicos ubicados en el entorno, como sensores y actuadores, para crear una *Organización Virtual Ubícua*. Un entorno con capacidades superiores o potenciadas (gracias a los agentes), es decir, un *Ambiente Inteligente*. Así, los objetivos específicos de este trabajo los podemos concretar en:

- Estudiar diferentes metodologías utilizadas para el diseño de MAS y de organizaciones de agentes abiertas. Este estudio se centra en buscar las características más importantes y comunes de las metodologías.

- Estudiar los diferentes enfoques aplicados en los Sistemas Ubícuos. Este estudio se centra en buscar las características más importantes, y las diferentes aproximaciones de cómo integrar los dispositivos de hardware o software al entorno.

- Diseñar los meta-modelos de la *Organización Virtual Ubícua*, que utilice componentes que sean independientes de la plataforma de ejecución. Este análisis se enfoca en la integración de componentes para crear un metamodelo unificado y genérico. Se analiza particularmente, la combinación y composición de conceptos muy diferentes del sistema, como son, por ejemplo, las *Unidades Organizaciones* usadas en los MAS abiertos y otros conceptos usados habitualmente en los sistemas ubícuos como es el dispositivo físico.

- Proponer un proceso de transformación del modelo a plataformas de ejecución, que soporten MAS abiertos y ubícuos. Se explora la problemática presente en las transformaciones entre modelos y las transformaciones finales de modelos a código.

- Diseñar una nueva arquitectura de agentes ubícua que permita la ejecución de organizaciones de agentes que puedan administrar y el controlar

los dispositivos del entorno. Se analiza la interoperabilidad de diferentes plataformas de ejecución de los agentes, como "*frameworks*" de ejecución distribuidos, fijos, móviles y/o empotradas (de recursos limitados).

- Evaluación y Pruebas. Finalmente se realiza una evaluación del trabajo probando el sistema con ejemplos empíricos (prototipos) desarrollados en esta tesis y comparándolo con otras aproximaciones, mostrando las ventajas del mismo.

1.2. Estructura del documento

El presente documento está estructurado de la siguiente manera:

- En el presente capítulo 1 se hace una introducción al tema, se describe la motivación para realizar este trabajo y se especifican los objetivos que guían esta investigación.

- El capítulo 2 realiza una revisión global del estado del arte de los temas más importantes relacionados con este trabajo. Por ello se introduce en los aspectos fundamentales del desarrollo de software que utiliza la filosofía de agentes. También se explica qué es el desarrollo dirigido por modelos y cómo aplicarlo al diseño de agentes. En la última parte del capítulo se describen diferentes propuestas de los sistemas ubícuos.

- El capítulo 3 explica como se ha diseñado y desarrollado el meta-modelo del MAS propuesto y el uso que tienen las diferentes vistas o partes. Además, se describe cómo usar el proceso de transformación en el diseño de agentes y se explica la forma de traducir el modelo que representa el marco de la organización de agentes usando el enfoque MDD. También en este capítulo se especifican las reglas de transformación de los sistemas, modelados utilizando la propuesta, a dos plataformas de agentes que soportan organizaciones. Además, se proponen reglas de transformación para obtener código del agente para las plataformas Andromeda y JADE.

- El capítulo 4 describe el sistema de implantación, que está soportado en una arquitectura por capas (basada en servicios y dividida según su funcionalidad), que permite la operación de las diferentes entidades del sistema, como son las unidades organizacionales, los agentes y los dispositivos físicos.

- El capítulo 5 describe una plataforma para la ejecución de agentes empotrados en teléfonos móviles, tabletas y otras tarjetas electrónicas que soporten el sistema operativo *Android*.

- En el capítulo 6 se muestran dos ejemplos que permiten evaluar la propuesta y demuestran la viabilidad de desarrollar organizaciones virtuales ubícuas usando MDD, donde las entidades activas de la organización pueden usar distintos servicios ofrecidos por los agentes o por los dispositivos físicos ubicados en el entorno.
- Finalmente, el capítulo 7 presenta las conclusiones de la propuesta y el trabajo futuro que puede ser desarrollado.

2

Estado del Arte

Índice

Este capítulo presenta un estudio del estado del arte del trabajo que se recoge en esta memoria, y dado que este trabajo emplea temas de diversos ámbitos, se ha estructurado el capítulo en siete secciones. La primera corresponde a la descripción del paradigma de agente y los conceptos fundamentales. La segunda explica cómo se utilizan las organizaciones virtuales en el diseño de sistemas multi-agentes. La tercera describe una visión global del enfoque desarrollo dirigido por modelos. La cuarta sección especifica cómo puede ser usado el enfoque MDD en el software basado en agentes. La quinta describe los conceptos básicos de la computación ubícua. La sexta resume propuesta de como se usa MDD en los sistemas ubícuos. Finalmente, en la séptima sección se exponen algunas conclusiones.

2.1. Agentes y sistemas multi-agente

En esta sección se hace una breve introducción al paradigma de agente y sistema multi-agente. Para ello, se realiza una revisión del concepto de agente y de las arquitecturas de agente existentes, así como del concepto de sistemas multi-agente y de los aspectos que conlleva un sistema de este tipo.

2.1.1. Agentes

En la actualidad el paradigma de agentes constituye uno de los campos de mayor desarrollo. Esto se debe a que su aparición en escena permite el desarrollo de aplicaciones que anteriormente era imposible desarrollar bien por su complejidad o porque la eficiencia de las posibles soluciones las hacía inabordables.

Los agentes software surgen dentro del campo de la Inteligencia Artificial, y a partir de los trabajos desarrollados en el área de la Inteligencia Artificial Distribuida (DAI, de sus siglas en inglés Distributed Artificial Intelligence), surge el concepto de sistemas multi-agente. Las investigaciones iníciales progresan y se crea el área de la Programación Orientada a Agentes y los Lenguajes de Comunicación de Agentes. Posteriormente este concepto se extiende al resto de la Ingeniería del software, planteándose hoy en día la Ingeniería del Software Orientada a Agentes[134].

Pero la pregunta inicial sería: ¿qué es un agente software?. Se pueden encontrar un gran número de definiciones del concepto de agente propuestas en la literatura, sin que ninguna de ellas haya sido plenamente aceptada por la comunidad científica, siendo quizás la más simple la de Russell[176], que considera un *agente* como una entidad que percibe y actúa sobre un entorno (ver Figura 2.1).

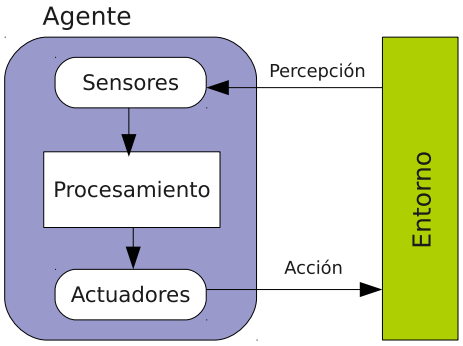

Figura 2.1: Esquema de un agente según la definición de Russell

Una de las definiciones más aceptadas hoy en día es la propuesta por Wooldridge y Jennings[200], según la cual, un *agente* es un sistema computacional ca-

paz de una acción autónoma y flexible en un determinado entorno, entendiendo por flexible que sea:

- *Reactivo*, responda al entorno en que se encuentra en base a las percepciones que recibe del mismo.
- *Pro-activo*, que sea capaz de intentar cumplir sus propios planes u objetivos.
- *Social*, que sea capaz de comunicarse con otros agentes mediante algún tipo de lenguaje.

Por último, es importante mencionar otra perspectiva más amplia, la de Nwana[147], que considera que un *agente* debe comprender todos los componentes software y/o hardware que presentan un comportamiento pro-activo que le permite realizar las actividades a su cargo. El autor propone una definición que es una extensión del término general de agente como la unión de todos los tipos de agentes (los que autor considera). Para Nwana el *agente* se forma en base a tres características: *autonomía*, *capacidad de aprendizaje* y *cooperación*. A partir de las combinaciones de estas características aparecen diferentes tipos de agentes, además se mencionan otros tipos de agentes que dependen de características del entorno en donde se encuentra o de las funciones que realiza. Para resumir estos agentes se pueden clasificar como:

- *Agentes de Interfaz:* Estos agentes mantienen la interacción con el usuario de forma gráfica y con componentes convencionales. Por lo tanto el usuario no necesita saber de manera explícita los procesos que el agente lleva a cabo, solo los resultados que éste le devuelve. Lo cual permite a los agentes tener un cierto grado de autonomía con respecto a los usuarios.
- *Agentes Colaborativos:* Este tipo de agente se centra en los aspectos de autonomía y cooperación. Estos agentes son capaces de establecer mecanismos de negociación para lograr una cooperación satisfactoria que permita resolver un problema.
- *Agentes Móviles:* Estos agentes se desplazan por redes (como Internet), interactuando con cada uno de los servidores, reuniendo información en beneficio de su propietario y regresando de vuelta después de haber ejecutado las tareas asignadas por su usuario.
- *Agentes de Información:* Estos agentes cumplen con el papel del manejo, de la manipulación o la recopilación de la información que se encuentran en diferentes fuentes distribuidas, como son los repositorios de documentos digitales.

- *Agentes Reactivos:* los agentes de esta clase se caracterizan por no contener ninguna representación simbólica del entorno, y su funcionamiento está regulado por unos mecanismos estímulo-respuesta.
- *Agentes Híbridos:* Estos agentes son la combinación de dos o más filosofías dentro de un agente simple (móvil, interfaz, colaborativo, etc.).

En la Figura 2.2 se muestra la clasificación de los agentes propuesta por Nwana.

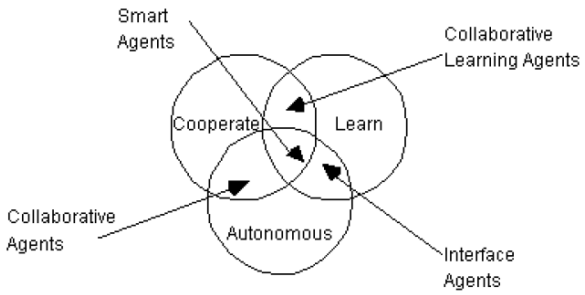

Figura 2.2: Tipos de agentes según Nwana

Atendiendo a esta idea, para poder asociar a un sistema informático el término agente debe ser capaz de cumplir algunos de los requerimientos anteriormente expuestos.

2.1.2. Características de los agentes

Algunas de las características que se suelen atribuir a los agentes para resolver problemas particulares y que han sido descritas en diversos trabajos[117, 54] se pueden resumir en:

- **Autonomía:** un agente es completamente autónomo si es capaz de actuar basándose en su experiencia, sin la intervención de un ser humano u otro agente. El agente es capaz de continuar aunque el entorno cambie severamente. Por otra parte, una definición menos estricta de autonomía será cuando el agente percibe el entorno.
- **Sociabilidad:** este atributo permite a un agente comunicar con otros agentes o incluso con otras entidades.
- **Racionalidad:** el agente siempre realiza lo correcto a partir de los datos que percibe del entorno.

- **Reactividad:** un agente actúa como resultado de cambios en su entorno. En este caso, un agente percibe el entorno y esos cambios dirigen el comportamiento del agente.
- **Pro-actividad:** un agente es pro-activo cuando es capaz de controlar sus propios objetivos a pesar de cambios en el entorno. Esta definición no contradice la de reactividad. El comportamiento del agente es resultado de dos tipos de comportamientos, el comportamiento receptivo y el comportamiento de descubrimiento. En un comportamiento receptivo, el agente es guiado por el entorno. El comportamiento de descubrimiento son procesos internos del agente para obtener sus propios objetivos. El agente debe tener un grado de comportamiento receptivo (atributo de reactividad) y un grado de comportamiento de descubrimiento (atributo de pro-actividad).
- **Adaptabilidad:** está relacionado con el aprendizaje que un agente es capaz de realizar y si puede cambiar su comportamiento basándose en ese aprendizaje.
- **Movilidad:** capacidad de un agente de trasladarse a través de una red telemática.
- **Veracidad:** asunción de que un agente no comunica información falsa a propósito.
- **Benevolencia:** asunción de que un agente está dispuesto a ayudar a otros agentes que lo solicitan, si esto no entra en conflicto con sus propios objetivos.

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

2.1.3. Arquitecturas básicas

El área de las arquitecturas de agente considera los aspectos relacionados con la construcción de sistemas computacionales (describiendo la interconexión de los módulos software/hardware) que satisfagan las propiedades de la teoría de agentes. En los agentes nos encontramos con una gran variedad de arquitecturas[134]. Una primera clasificación de las arquitecturas puede ser realizada según si todas las capas tengan acceso al entorno (horizontales) o sólo la capa más baja tenga acceso al entorno (verticales), tal como se muestra en la Figura 2.3. Las arquitecturas horizontales ofrecerán la ventaja del paralelismo entre capas a costa de un

alto conocimiento de control para coordinar las capas, mientras que las verticales reducen este control a costa de una mayor complejidad en la capa que interactúa con el entorno.

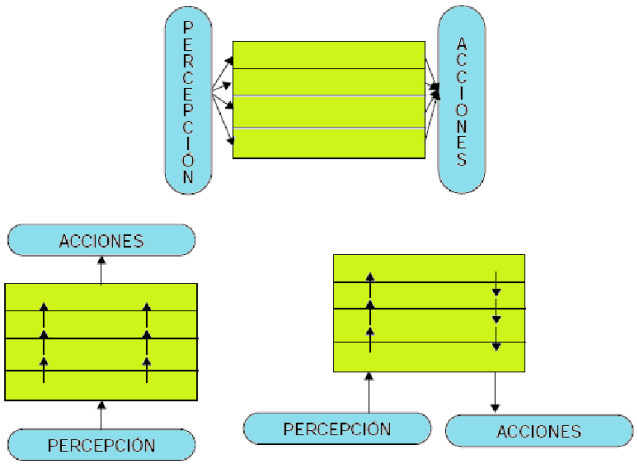

Figura 2.3: Arquitectura horizontal y vertical

Otros tipos de arquitecturas, de acuerdo al paradigma de agente (arquitecturas atendiendo al tipo de procesamiento empleado)[111] son:

Arquitecturas deliberativas: en las cuales el agente es construido según el paradigma de la inteligencia artificial simbólica. Las arquitecturas de agentes deliberativos suelen basarse en la teoría clásica de planificación de inteligencia artificial: dado un estado inicial, un conjunto de operadores/planes y un estado objetivo, la deliberación del agente consiste en determinar qué pasos debe encadenar para lograr su objetivo, siguiendo un enfoque descendente *(top-down)*. Un ejemplo de esta arquitectura es el tipo de *agente BDI*[169] (BDI, de sus siglas en inglés Belief-Desire-Intention), que se muestra en la Figura 2.4.

Arquitecturas reactivas: cuestionan la viabilidad del paradigma simbólico y proponen una arquitectura que actúa siguiendo un enfoque estímulo-respuesta. Las arquitecturas reactivas no tienen un modelo del mundo simbólico como elemento central de razonamiento y no utilizan razonamiento simbólico complejo, sino que siguen un procesamiento ascendente *(bottom-up)*, para lo cual mantienen una serie de patrones que se activan bajo ciertas condiciones de los sensores y tienen un efecto directo en los actuadores. Un ejemplo de este tipo es la arquitectura de *subsunción*[48, 132] que se observa en la Figura 2.5.

Arquitecturas híbridas: mediante esta aproximación el agente es construido por medio de dos o más subsistemas. Uno de ellos es el deliberativo, el cual, contiene un modelo simbólico del mundo, mientras que otro es reactivo. Estas

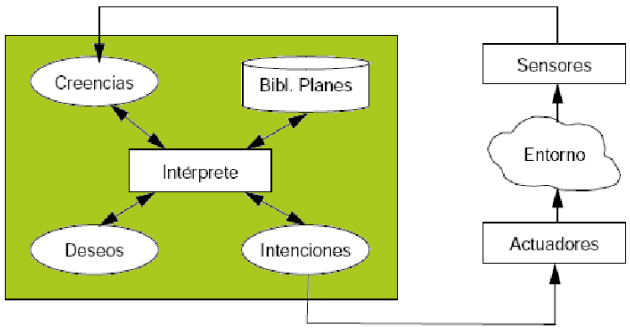

Figura 2.4: Arquitectura deliberativa

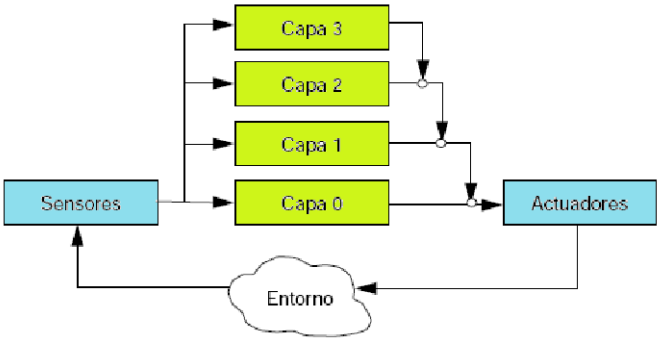

Figura 2.5: Arquitectura de subsunción (reactiva)

arquitecturas combinan módulos reactivos con módulos deliberativos. Los módulos reactivos se encargan de procesar los estímulos que no necesitan deliberación, mientras que los módulos deliberativos determinan qué acciones deben realizarse para satisfacer los objetivos locales y cooperativos de los agentes. Un ejemplo de esta arquitectura es *TouringMachine*[84], la cual se observa en la Figura 2.6.

2.1.4. Sistemas multi-agente

En la mayoría de ocasiones los agentes no son desarrollados de forma independiente sino como entidades que constituyen un sistema denominado multi-agente (MAS, por sus siglas en inglés Multi-Agent Systems). En este caso los agentes deben o pueden interactuar entre ellos. Las interacciones más habituales como son informar o consultar a otros agentes permiten a los agentes hablar entre ellos, tener en cuenta lo que realiza cada uno de ellos y razonar acerca del papel jugado por los distintos agentes[117].

Los MAS deben disponer de una infraestructura asociada que permita a los

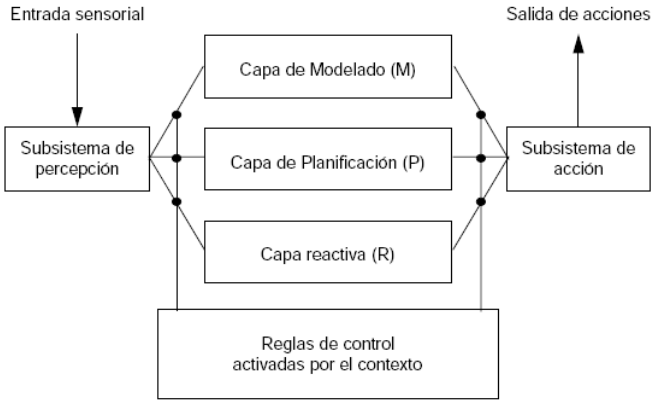

Figura 2.6: Arquitectura híbrida *TouringMachine*

agentes operar de forma efectiva así como interactuar productivamente entre ellos. Esta infraestructura incluirá todos los aspectos relacionados con los procesos de comunicación, esto es fundamentalmente, envío y recepción de mensajes por medio de un lenguaje de comunicación determinado. Así como disponer de protocolos de interacción que permitan conversar a los agentes a un alto nivel de abstracción.

Asumiendo la capacidad de comunicación de los agentes que pertenecen a un MAS, esta capacidad puede verse como un elemento más de percepción (llegada de mensajes), de cognición (interpretación y determinación de las acciones a realizar) y de acción (envío de mensajes). Esta habilidad, considerada por algunos autores como inherente a un agente, tal y como se ha comentado anteriormente, permite a un conjunto de agentes formar sociedades donde pueden existir objetivos de grupo, además de los objetivos propios de cada agente. Para llevar a cabo sus objetivos tanto individuales como grupales, un agente puede necesitar establecer una comunicación con otros agentes, estableciendo en este caso un proceso de coordinación[117].

La coordinación es una propiedad de un sistema de agentes que desarrollan una actividad en un entorno compartido. El grado de coordinación alcanzado por un grupo de agentes depende del compromiso que pretende alcanzar cada uno de ellos.

2.1.5. Plataformas de Agentes

Para desarrollar agentes se necesitan plataformas de ejecución. Existen varias plataformas de agentes, cada una implementa un modelo especifico de agente, crean su propio modelo de interacción y algunas de ellas cumplen con el están-

dar FIPA[1] (Foundation for Intelligent Physical Agents). Entre las populares o relevantes podemos mencionar:

- **JADE**[2][30] es una de dichas plataformas, muy usada porque proporciona conceptos de programación que simplifican la implementación de MAS, además cumple con las especificaciones FIPA para dar soporte a la infraestructura de comunicación entre los distintos agentes. La arquitectura FIPA[87] define en un nivel abstracto, cómo dos agentes pueden localizarse y comunicarse entre sí mediante el registro y el intercambio de mensajes. Concretamente, FIPA propone que una plataforma de agentes debe contener obligatoriamente al menos las siguientes funciones:

  - Un *AMS* (de sus siglas en inglés, Agent Management System): controla el acceso de los agentes y el uso de la plataforma. Mantiene información de todos los agentes dentro de la plataforma, incluyendo sus identificadores y direcciones de transporte. Presta un servicio de páginas blancas a los agentes conectados a la plataforma.
  - Un *DF* (de sus siglas en inglés, Directory Facilitator): proporciona los servicios de páginas amarillas para la plataforma de agentes. Los agentes dentro de la plataforma pueden registrar sus servicios en el facilitador directorio (*Directory Facilitator*) y pueden consultarse con el fin de conocer los servicios ofrecidos por otros agentes.
  - Un *ACC* (de sus siglas en inglés, Agent Communication Channel): es el método de comunicación predeterminado, que ofrece un servicio de transporte de mensajes fiable, preciso y ordenado. El ACC permite la comunicación de los agentes dentro y fuera de la plataforma.

  En la plataforma JADE el modelo de agente fue intencionalmente dejado abierto, para que el diseñador programe la estructura interna del agente y los conceptos que crea necesarios. Otras características importantes que proporciona esta plataforma, que podemos mencionar son:

  - Lenguaje de programación: Java.
  - Programación de las acciones del agente en base a comportamientos (behaviours).
  - Cumple con FIPA ACL[86] para envío y recepción de mensajes.
  - Manejo de información usando ontologías.

[1] http://www.fipa.org/
[2] http://jade.tilab.com/

  - Organizada en contenedores.
  - Soporte GUI.

- **JADE-Leap**[32] (JADE Light Extensible Agent Platform), es una versión de JADE que puede implementar agentes empotrados. JADE-Leap fue seleccionada como plataforma para la ejecución de los agentes empotrados en este trabajo dada su alta popularidad.

- **JACK**[3] en ella se provee un entorno de desarrollo orientado a agentes construido sobre Java y completamente integrado con este lenguaje de programación. Incluye a JACK *Agent*, un lenguaje orientado a agentes, JACK *Agent* no sólo extiende la funcionalidad de Java, sino que además provee un entorno para soportar un nuevo paradigma de programación. JACK ofrece tres extensiones a Java: un lenguaje de agentes, un compilador de agentes y un kernel de agentes. Algunas de las características más importantes que podemos mencionar son:

  - Agentes BDI[121].
  - Interfaz gráfica que soporta a la plataforma GUI.
  - Lenguaje de comunicación propio, pero puede soportar KQML[126].
  - Licencia de pago.
  - Lenguaje de programación propio: JACK.
  - Soporte GUI.

- **MADKit**[4] en esta plataforma los agentes están caracterizados por roles y grupos. La plataforma está construida alrededor del concepto de Agente-Grupo-Rol desarrollado en el contexto del modelo AALAADIN[81], que define una descripción muy simple del esquema de coordinación y de negociación a través del sistema multi-agente. Utiliza a los grupos y los roles como base para construir aplicaciones complejas. No está asociada a ninguna arquitectura de agentes en particular, permitiendo a los usuarios implementar libremente sus propias arquitecturas. Algunas de las características más importantes que podemos mencionar son:

  - Agentes basado en grupos.
  - Lenguaje de agentes KQML.
  - Soporta múltiples lenguajes de programación.

[3] http://www.agent-software.com/shared/products/index.html
[4] http://www.madkit.org

- Soporte GUI.

- **ZEUS**[5] es una herramienta para construir aplicaciones multi-agente colaborativas, además ZEUS define una metodología de diseño de sistemas multi-agente y lo soporta mediante un entorno visual para capturar las especificaciones de los agentes. La idea es proveer una herramienta de propósito general y personalizable, que permita la creación de agentes colaborativos para crear sistemas multi-agente. Algunas de las características más importantes que podemos mencionar son:

  - Lenguaje de agentes KQML.
  - Agentes tipo BDI.
  - Lenguajes de programación Java.
  - Soporte GUI.

- **MAGENTIX2**[6] [188] es una plataforma de agentes para sistemas multi-agente abiertos. Su principal objetivo es soportar protocolos flexibles de interacción y de conversación, para hacer frente al gran dinamismo de la topología del sistema, que es una consecuencia propia de la naturaleza distribuida y autónoma de los componentes. En este sentido, la plataforma proporciona soporte a tres niveles: a nivel de las organizaciones, a nivel de interacción y a nivel de agente. Otro aspecto importante que cubre Magentix2, son los problemas de seguridad, por ello posee un modelo de seguridad que incorpora mecanismos de bajo nivel y medidas de confianza, que complementan los métodos criptográficos clásicos. Algunas de las características más importantes que podemos mencionar son:

  - Soporta el manejo de organizaciones.
  - La infraestructura de comunicación se basa en el sistema AMQP[7] (Advanced Message Queuing Protocol).
  - Módulos de seguridad en las interacciones.
  - Soporte nativo de agentes tipo Jason[42].
  - Lenguajes de programación Java.
  - Mecanismos de rastreo (*tracking*).

[5] http://more.btexact.com/projects/agents/zeus/
[6] http://gti-ia.upv.es/sma/tools/magentix2/index.php
[7] http://www.amqp.org/

- **PANGEA**[205, 206], es una plataforma de agentes para desarrollar sistemas multi-agente abiertos, especialmente los que incluyen aspectos organizativos. La plataforma permite la gestión integral de las organizaciones y ofrece herramientas para el usuario final. Además, incluye un protocolo de comunicación basado en el estándar de IRC (Internet Relay Chat), lo que facilita la aplicación y sigue siendo robusto, incluso con un gran número de conexiones. Con la inclusión de un agente especial de comunicaciones (*Communication Agent*) y un agente "Sniffer" hace posible la oferta de servicios web para el control distribuido de las interacciones. Algunas de las características más importantes que podemos mencionar son:

  - Soporta el manejo de organizaciones.
  - Basado en el uso de servicios Web.
  - Soporte de agentes del tipo BDI y otros modelos.
  - Lenguajes de programación Java.
  - Herramientas para la depuración.

2.2. Organizaciones Virtuales

En esta sección presentamos una descripción de los tópicos y conceptos que son los más relevantes en el área de *Organizaciones Virtuales* y modelos de desarrollado en MAS. También se describe algunas de las contribuciones con respecto al modelado de organización en sistemas basados en agentes y se discuten algunos problemas abiertos. Finalmente, también se explica cómo esos problemas pueden ser resueltos usando un enfoque MDD.

En el área de sistemas multi-agentes, el término *Organización Virtual* (VO, de sus siglas en inglés Virtual Organization) ha sido usado principalmente para describir un conjunto de agentes que a través de patrones de interacción se coordinan unos y otros, con el fin de alcanzar los objetivos generales del sistema[38]. Por lo tanto, en esta sección se describen las principales características de las *Organizaciones Virtuales* (VOs), y además los factores o dimensiones que son necesarias para su análisis y modelado, con el fin de facilitar el desarrollo de MAS Abiertos (Open MAS).

Las primeras metodologías usadas en el diseño de MAS fueron las *orientadas a agentes*[110, 202, 189]. Ellas asumen una perspectiva individualista, donde la principal entidad es el agente, que sigue sus propios objetivos individuales sobre la base de sus propias creencias y habilidades. Además, ellas consideran que los agentes son benevolentes, todos tienen objetivos comunes, y cooperan con el fin

de alcanzar esos objetivos. Por lo tanto, sólo son adecuados para los sistemas cerrados. Por otra parte, las estructuras sociales no son modeladas específicamente, pero se supone que emergen como resultado de la interacción entre los agentes.

En los últimos años, algunos trabajos de sistemas basados en agentes se han centrado en proveer procedimientos y métodos para el diseño de MAS abiertos, donde los agentes pueden tener comportamientos auto-interesados o ser egoístas. Los MAS abiertos también pueden permitir la participación de agentes heterogéneos con diferentes arquitecturas e incluso con lenguajes diferentes[71]. Así, con el fin de soportar los MAS abiertos, existe una nueva tendencia (emergente) entre los desarrolladores en enfocarse en los aspectos organizativos de la sociedad de agentes, para dirigir el proceso de desarrollo del sistema utilizando los conceptos de organización, normas, roles, etc. Esto ha guiado a un nuevo enfoque que es conocido como metodologías *orientadas a la organización*.

En las metodologías basadas en la organización, el diseñador del MAS se enfoca en la organización del sistema, teniendo en cuenta sus principales objetivos, la estructura y las normas sociales. Dos tendencias diferentes se pueden observar al comparar los diferentes enfoques de estas metodologías. Por un lado, los métodos tales como PASSI[66], MOISE[92], TROPOS[125], MESSAGE[50] y INGENIAS[159] detallan los roles del sistema, los grupos y las relaciones, pero no consideran de manera explícita las normas sociales. Por otro lado, métodos y marcos tales como SODA[154], GAIAExOA[204], Electronic Institutions[80], OperA[71], OMNI[72] y GORMAS[22] están enfocados en las normas sociales y en definir explícitamente las políticas de control para crear y reforzar la organización. El principal objetivo de los métodos de este tipo, es el diseño de sistemas multi-agente abiertos, en los cuales los agentes con comportamiento egoísta pueden participar. Estos agentes pueden ser controlados por medio de las normas sociales y una estructura organizativa adecuada.

Las *Organizaciones Virtuales* proporcionan un marco para la actividad y la interacción de los agentes a través de la definición de los roles, de expectativas de comportamiento y de relaciones de autoridad tales como el control[204]. En las VOs existe un nuevo nivel que es independiente de los agentes que lo constituyen, el cual puede ser reemplazado de forma dinámica. Las VOs proporcionan una forma de dividir el sistema, separándolo en grupos o unidades (entidades) que mantienen ciertas relaciones unas con otras (que proporciona el contexto para la interacción entre los agentes y las distintas entidades) y tomando parte de los patrones de interacción con otros roles de una manera institucionalizada y sistemática.

Una VO está representada de manera similar a las organizaciones humanas, basado en la teoría de la Organizaciones Humanas[21, 22]. Esto permite describir los principales aspectos de una organización: su *estructura*, *funcionalidad*, *dinamismo*, *entorno*, y *normas*. Estos cinco elementos describen los miembros

(entidades) que componen la organización, la topología de la organización, los servicios y funciones que ofrece la organización, la evolución de la organización en el tiempo, el entorno donde se encuentra la organización, y las normas sobre la conducta de los miembros, respectivamente.

2.2.1. Metodologías de Organización

El desarrollo de MAS abiertos necesita de metodologías optimizadas para el diseño de software basado en agentes. Sin embargo, como se mencionó anteriormente, las metodologías que surgieron al comienzo son clasificadas como *orientadas a agentes*, donde no se describe la *organización* explícitamente. Estos sistemas son generalmente cerrados, y los agentes externos están prohibidos. En los últimos años, sin embargo, han surgido nuevas metodologías orientadas a la *organización*. Éstas permiten (parcialmente) el diseño de MAS abiertos, lo que permite el desarrollo de sistemas *heterogéneos*. Estas metodologías orientadas a la organización permiten que los agentes externos puedan acceder a la funcionalidad del sistema, pero los agentes están obligados a adherirse a las *normas sociales* del sistema. Entre las metodologías más importantes que permiten el diseño de *Organizaciones* tenemos: PASSI[66], MOISE[92], OperA[71] y GORMAS[22]. Las cuales se resumen a continuación:

- **PASSI**[66] es una metodología que se ha diseñado para el desarrollo de MAS con las siguientes características:

  1. Es altamente distribuida.
  2. Está sujeta a una tasa relativamente baja de los cambios de requisitos.
  3. Es abierta a agentes y sistemas externos, que son desconocidos en el tiempo de diseño, pero que van a interactuar con el sistema en tiempo de ejecución.

  La metodología PASSI se compone de tres modelos que se enfocan en diferentes perspectivas. El proceso de construcción de un modelo cuenta con diecinueve fases. Utiliza UML y lo adapta para representar el MAS a través de sus mecanismos de extensión (restricciones, valores etiquetados y estereotipos). Además, tiene un meta-modelo de MAS que se utiliza para guiar el diseño en términos de escenarios, requisitos, ontologías y recursos. Los escenarios describen una secuencia de interacciones entre los actores y el sistema, y los requisitos están representados con diagramas convencionales de casos de uso.

- **MOISE**[92] es un modelo de organización para sistemas multi-agente basado en nociones tales como roles, grupos y misiones. Permite a un MAS

tener una descripción explícita de su organización. Este modelo se basa en tres conceptos principales: *Roles*, que restringen el comportamiento individual de los agentes; *Organizational links*, el cual regula el intercambio social entre los diferentes agentes; *Groups*, que limita la distribución de los agentes implicados en las interacciones. Este enfoque considera a la organización como un conjunto normativo de reglas que restringe el comportamiento de los agentes. El modelo MOISE (modelo organizacional) está estructurado en tres niveles:

1. Nivel individual (*individual*), donde se define las responsabilidades de cada agente.
2. Nivel de agencia (*agency*), donde se define la incorporación de los agentes en una estructura superior.
3. Nivel de sociedad (*society*), donde se define la estructuración global y la interconexión de los agentes con otras estructuras.

- **OperA**[71] presenta un modelo de interacción entre los agentes en organizaciones o ambientes. OperA ofrece una forma flexible de representar la interacción y la adopción de roles; ya que abstrae de cada agente individual las representación interna específica, y separa el modelado de los requerimientos de la organización y de los objetivos. Sus tres principales hipótesis son las siguientes:

  1. Las entidades autónomas necesitan un entorno social con el fin de alcanzar sus propias metas individuales.
  2. Las organizaciones y/o sociedades tienen sus propias metas y requerimientos globales, que no son necesariamente compartidos con ninguna de las entidades participantes. Sin embargo, estas metas deben alcanzarse mediante la actividad coordinada de estas entidades individuales.
  3. Un proceso de negociación y de ajuste es necesario entre las necesidades individuales y sociales con el fin de coordinar la autonomía individual con las necesidades sociales y metas.

- **GORMAS**[22] es una guía metodológica para el diseño de MAS abiertos desde la perspectiva de las organizaciones humanas. Esta guía se compone de un conjunto de fases de análisis de requerimientos, diseño de la estructura, y el diseño de la dinámica organizacional. En estas fases, los diseñadores principalmente especifican qué servicios ofrece el sistema, su estructura interna, y las normas que regulan su comportamiento, teniendo

en cuenta las características específicas del MAS abierto. GORMAS incorpora un nuevo modelo de Organización. Este modelo se compone de seis meta-modelos que detallan las características específicas de una organización. Al usar estos modelos de la organización, GORMAS especifica el análisis y el diseño de la organización.

De las muchas de las metodologías analizadas, se encontró que éstas no incluyen todas las fases necesarias para desarrollar MAS abiertos. Ellas principalmente excluyen las últimas fases, en la cual la especificación de la *Organización Virtual* debe ser convertida en código ejecutable para una plataforma de agente específica. El problema fundamental para obtener código ejecutable para las VOs es la carencia de plataformas de agentes que den soporte a esta clase de sistemas complejos. Aunque hay actualmente diferentes plataformas que soportan la ejecución de agentes (tales como JADE[8] o JACK[9]), y alguna de estas plataformas tratan con conceptos organizacionales, sin embargo, ellas no pueden soportar directamente los conceptos que aparecen en MAS abiertos, tales como normas, roles y topología organizacional.

Aunque existen los problemas anteriormente descritos, es imprescindible mencionar **E-Institutions**[80] como un trabajo pionero en la organización de agentes. En esa propuesta, los agentes están estructurados en una organización o institución electrónica (*Electronic Institution*) con unos mecanismos de regulación entre sus miembros. Sin embargo en esta propuesta, la creación de nuevos grupos (organizaciones dinámicas) no está permitida, es decir, el trabajo se centra principalmente en el empleo de las estructuras de organización durante el proceso de diseño y en la regulación de sus agentes. La creación de nuevos grupos o sub-organizaciones dinámicamente en tiempo de ejecución es muy complicado.

A pesar de que la propuesta de **E-Institutions** presenta algunas limitaciones será empleada en este trabajo. Sin embargo, existe actualmente una propuesta más adecuada al soporte y ejecución de MAS abiertos, ésta propuesta conocida como THOMAS, es usada en esta Tesis como el principal marco de ejecución de los agentes. A continuación se describe lo que es THOMAS.

2.2.2. Marco de ejecución: THOMAS

THOMAS (MeTHods, Techniques and Tools for Open Multi-Agent Systems)[19], es una nueva arquitectura de un sistema multi-agente abierto, que consiste en un conjunto de módulos interrelacionados que soportan el desarrollo de sistemas aplicados en entornos que funcionan como una "sociedad". Debido a los avances tecnológicos de los últimos años, el término "sociedad", en donde los sistemas

[8]http://jade.tilab.com/
[9]http://aosgrp.com/

multi-agente participan, tiene que cumplir con varios requisitos, tales como: distribución, constante evolución, la flexibilidad para permitir que los miembros pueda entrar o salir de la sociedad, el adecuado manejo de la estructura organizativa que define la sociedad, la ejecución de agentes en múltiples dispositivos (incluidos dispositivos con recursos limitados). Todos estos requisitos definen un conjunto de características que se pueden abordar a través del paradigma de sistemas abiertos y organizaciones virtuales.

Básicamente, la arquitectura de THOMAS consiste en un conjunto de componentes modulares, y sus principales componentes son descritos a continuación (y se pueden observar en la Figura 2.7):

- **Service Facilitator (SF)**, este componente ofrece servicios simples y complejos de los agentes y organizaciones activas. Básicamente, su funcionamiento es como un servicio de páginas amarillas y un descriptor de servicios cuya misión es brindar un servicio de páginas verdes.

- **Organization Management System (OMS)**, es el principal responsable del manejo de las organizaciones y de sus entidades. Por lo tanto, permite la creación y gestión de cualquier organización. El OMS da soporte a las organizaciones virtuales dinámicas y también dispone de las facilidades para la gestión de las normas en las organizaciones virtuales. Por lo tanto, es principalmente responsable de la gestión del ciclo de vida de una organización.

- **Platform Kernel (PK)**, mantiene los servicios básicos de gestión de la plataforma de agentes, como la creación del agente, el descubrimiento del agente y las comunicaciones. El manejo de servicios son considerados en un alto nivel de abstracción, pero ellos están relacionados con otros servicios organizacionales prestados tanto por el OMS y SF. Por lo tanto, este *Platform Kernel* es una plataforma de ejecución adecuada para el desarrollo de las organizaciones virtuales.

El *Service Facilitator* es un mecanismo que permite a las organizaciones y agentes ofrecer y descubrir servicios. El SF actúa como una puerta de entrada para acceder a la plataforma THOMAS. Este acceso se logra de forma transparente por medio de técnicas de seguridad y gestión de derechos de acceso. El SF puede encontrar servicios mediante la búsqueda de un perfil determinado de servicio o la búsqueda de los objetivos que se pueden cumplir tras la ejecución del servicio. Esto se hace usando *matchmaking*[191] y los mecanismos de composición servicio[190] que se proporcionan por el SF. El SF también actúa como un gestor de páginas amarillas y, por lo tanto, permite encontrar que entidades proporcionan un servicio dado.

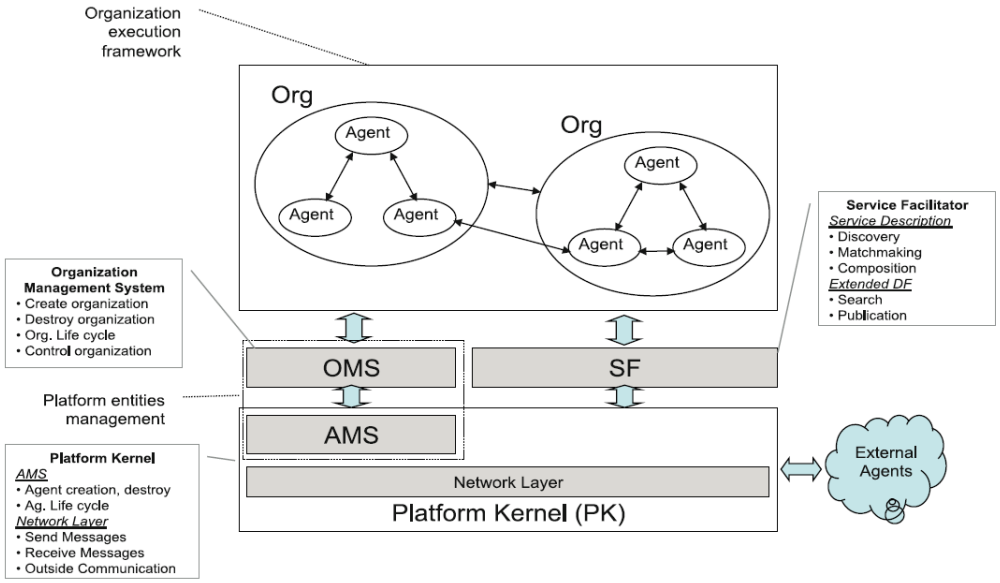

Figura 2.7: Arquitectura de THOMAS

En THOMAS, un servicio se define como una tupla (*sID, goal, prof, proc, ground, ont*) donde:

- *sID*, es un identificador único del servicio.
- *goal*, es el objetivo final del servicio y proporciona el primer nivel de abstracción en la composición de servicios.
- *prof*, es el perfil de servicio que describe el servicio en términos de sus entradas, salidas, pre-requisitos y efectos y los atributos no funcionales de una forma legible para aquellos agentes que buscan información.
- *proc*, describe cómo un cliente tiene que utilizar este servicio, especificando el contenido semántico para utilizar el servicio, las situaciones en las que se obtiene, y los procesos paso a paso para obtener estos resultados. En otras palabras, se especifica cómo llamar a un servicio y lo que ocurre cuando el servicio se ejecuta.
- *ground*, especifica en detalle cómo un agente puede acceder al servicio. Este campo especifica un protocolo de comunicación, los formatos de mensaje, el puerto de contacto, y otros detalles específicos del servicio.
- *ont*, es la ontología que da sentido a todos los elementos del servicio.

En resumen, el SF proporciona un conjunto de servicios estándar (servicios SF) (ver Figura 2.8) para administrar los servicios prestados por las organizaciones o agentes individuales. Estos se clasifican en:

1. Registro (*Registration*), para añadir, modificar o eliminar los servicios del directorio de SF;

2. Asequibilidad (*Affordability*), para la gestión de la asociación entre prestadores y sus servicios,

3. Descubrimiento (*Discovery*), para la búsqueda y componer servicios en respuesta a las necesidades de los usuarios.

Estos servicios SF también puede ser empleado por el OMS y PK de THOMAS para anunciar sus propios servicios.

SF services

Type	SF service	Description
Registration	RegisterProfile	Creates a new service description (profile)
	RegisterProcess	Creates a specific implementation (process) for a service
	ModifyProfile	Modifies an existing service profile
	ModifyProcess	Modifies an existing service process
	DeregisterProfile	Removes a service description
	BeforehandComposer	Composes services in background (internal)
Affordability	AddProvider	Adds a new provider to an existing service process
	RemoveProvider	Removes a provider from a service process
Discovery	SearchService	Searches for a service (or a composition of services) that satisfies the user requirements
	GetProfile	Gets the description (profile) of a specific service
	GetProcess	Gets the implementation (process) of a specific service

Figura 2.8: Servicios del SF

El *Organization Management System* se encarga de la gestión del ciclo de vida de la organización, incluyendo la especificación y la administración de los componentes estructurales (roles, unidades y normas) y los componentes de ejecución (agentes participantes, los roles que desempeñan, y las unidades organizativas activas).

Las organizaciones están estructuradas por medio de *Unidades Organizativas* (OUs), que representan grupos de entidades (agentes u otras OUs), que se relacionan para lograr un objetivo común. Estas unidades organizativas tienen una topología interna (es decir, jerárquica, equipo, plana, ...), que impone restricciones sobre las relaciones de los agentes y su control (es decir, supervisión o

relaciones de información). Además, una unidad organizativa indica qué posiciones estructurales del sistema son (miembro, supervisor, subordinado), así como las relaciones entre estas posiciones impuestas por la estructura. En THOMAS, por defecto existe una primera unidad organizativa "virtual" que representa el "mundo" del sistema, donde los agentes participan de la forma predeterminada en que se han definido. El OMS crea unidades organizativas dentro de esta unidad "virtual", que también puede estar compuesta a su vez de varias unidades organizativas.

Un *rol* representa una posición dentro de la OU en el que está definido. También está relacionado con las restricciones de comportamiento, que especifican su funcionalidad (servicios que necesita y ofrece), y con acciones deónticas (prohibición, las obligaciones y permisos), así como las consecuencias de estas medidas (sanciones y recompensas). Por lo tanto, los roles podrían tener asociada las normas para el control de las acciones del rol (que servicios a los agentes se les permite solicitar, ofrecer o prestar cuando juegan un rol especifico, es decir, permisos de acceso a los recursos). Dado que los agentes dinámicamente pueden adoptar roles dentro de las unidades organizativas, el OMS controla este proceso de adopción de roles y los controles de las entidades que desempeñan cada rol a través del tiempo.

El OMS ofrece varios servicios necesarios para el desempeño adecuado de una organización (ver Figura 2.9). Se clasifican de la siguiente manera:

- Servicios estructurales (*Structural services:*): para registrar o dar de baja los componentes estructurales, funciones específicas, normas y unidades. Los servicios estructurales incluyen el registro y servicios de información. Los servicios de registro permitirán la creación o supresión de los elementos básicos de la organización: las unidades, funciones y normas. Los *servicios de información* proporcionan información específica de todos los componentes de la organización.

- Servicios Dinámicos (*Dynamic services*): para la adopción de roles y la aplicación de sanciones (por ejemplo, obligando a un agente para que deje un rol específico).

El último componente es el *Platform Kernel*, encargado de prestar los servicios habituales requeridos en una típica plataforma multi-agente. Por lo tanto, es responsable de la gestión del ciclo de vida de los agentes que forman parte de las diferentes organizaciones, y también hace que sea posible tener un canal de comunicación (que incorpora varios mecanismos de transporte de mensajes) para facilitar la interacción entre las entidades.

Los servicios necesarios del PK en una infraestructura THOMAS se clasifican en cuatro tipos (ver Figura 2.10):

OMS services

Type	OMS service	Description
Structural	Registration	
	RegisterRole	Creates a new role within an OU
	RegisterNorm	Includes a new norm within an OU
	RegisterUnit	Creates a new OU within a specific organization
	DeregisterRole	Removes a specific role description from an OU
	DeregisterNorm	Removes a specific norm description
	DeregisterUnit	Removes an OU from an organization
	Information	
	InformAgentRole	Indicates roles adopted by an agent
	InformMembers	Indicates entities that are members of a specific OU
	QuantityMembers	Provides the number of current members of a specific OU
	InformUnit	Provides OU description
	InformUnitRoles	Indicates which roles are the ones defined within a specific OU
	InformRoleProfiles	Indicates all profiles associated to a specific role
	InformRoleNorms	Provides all norms addressed to a specific role
Dynamic	AcquireRole	Requests the adoption of a specific role within an OU
	LeaveRole	Requests to leave a role
	Expel	Forces an agent to leave a specific role

Figura 2.9: Servicios del OMS

1. Registro (*Registration*), para añadir, modificar o eliminar los agentes nativos de la plataforma;

2. Descubrimiento (*Discovery*), para obtener información acerca de los agentes nativos que están activos en la plataforma;

3. Administración (*Management*), para controlar el estado de activación del agente nativo en la plataforma;

4. Comunicación (*Communication*), para comunicarse con los agentes dentro y fuera de la plataforma

Como comentario final a este punto, una vez que se conoce las organizaciones de agentes, metodologías y plataformas analizadas, en este trabajo se propone la utilización del enfoque conocido como desarrollo dirigido por modelos para crear un modelo unificado organizaciones de agentes, que esté dirigido a modelar las sociedades abiertas en las que agentes heterogéneos y autónomos, enfocado en la integración de servicios. En este modelo, las funcionalidades de las entidades

PK services

Type	Service	Description
Registration	Register	Registers a new agent in the platform
	Deregister	Eliminates an agent registration
	Update register	Modifies the information appearing in an agent register (except the agent name).
Discovery	Agent Search	Requests information from a registered agent in the platform.
	Get Description	Obtains the platform description.
Management	Suspend	Suspends the execution of a specific agent.
	Activation	Activates the execution of an agent that is currently suspended.
Communication	Send	Sends a message to any agent in the platform or outside of it.

Figura 2.10: Servicios del PK

son descritas de forma abstracta con el objetivo de facilitar el proceso de diseño de organizaciones virtuales. Por ello, a continuación se resumen los principales aspectos del Desarrollo Dirigido por Modelos.

2.3. Desarrollo Dirigido por Modelos

En esta sección se hace una breve introducción sobre la metodología de Desarrollo Dirigido por Modelos (MDD, por sus siglas en inglés Model Driven Development) que se utiliza ampliamente para el desarrollo de software orientado a objetos. Para ello, se realiza una revisión de algunos conceptos fundamentales y formas de aplicación en el desarrollo de software. Además se explica cómo utilizar esta metodología para desarrollo de sistemas basados paradigma de agente y sistemas multi-agente.

2.3.1. Los motivos de usar MDD

La reutilización de software es una de las estrategias que se considera primordiales en la industria de software, debido a que permite enfrentar el reto de desarrollar productos informáticos con niveles de calidad y productividad adecuados en un contexto de negocio altamente complejo, dinámico y con acelerados cambios tecnológicos[167, 35]. Además la industria del software siempre ha tenido como norte el mejorar la portabilidad y la interoperabilidad de sus sistemas informáticos.

Estos desafíos mueven a los investigadores a proponer un proceso de desarrollo en los cuales los modelos (antes que el código) sea el factor fundamental

de la estrategia de desarrollo. En donde se proporcionen mecanismos y herramientas de trabajo integradas que asisten al desarrollador en la construcción y transformación progresivas de modelos hasta llegar al código de implementación. Así surge la iniciativa conocida como el desarrollo dirigido por modelos que resalta la importancia y potencia el uso de los modelos como estrategia clave para especificar y comprender un sistema de software, luego con la aplicación progresiva de conversiones de los modelos (pasos de transformación) llegar al sistema informático a implementar, es decir, al código que es necesario ejecutar.

De esta forma se presenta el MDD como una metodología para el desarrollo del software que tiene como objetivos mejorar la *productividad*, la *interoperabilidad*, la *portabilidad* y la *reutilización* de los sistemas informáticos.

2.3.2. ¿Qué es un modelo?

Obtener una definición clara de lo que es un modelo es importante ya que éste es el núcleo del proceso del desarrollo dirigido por modelos. Si buscamos su definición en el diccionario de la Real Academia de la Lengua Española, encontraremos diferentes y variadas dependiendo del contexto donde se use el concepto *modelo*, por ejemplo un modelo es:

- Una persona que posa para un artista.
- Persona, por lo común desnuda, que sirve para el estudio en el dibujo.
- Una réplica de un objeto en escala reducida, es decir, una miniatura.
- Una idea usada en un ejemplo.
- Arquetipo o punto de referencia para imitarlo o reproducirlo.
- En las obras de ingenio y en las acciones morales, ejemplar que por su perfección se debe seguir e imitar.
- Un formulismo matemático para expresar relaciones, entidades y operaciones.

Pero de todas las definiciones anteriores se obtiene algunas características comunes:

- Un modelo es siempre una abstracción de algo que existe en la realidad.
- Un modelo puede ser usado como un ejemplo para algo que existe en la realidad.
- Una idea usada en un ejemplo.

Así de forma general podemos decir que un modelo es la representación esquemática o conceptual de un fenómeno o de *algo* (un objeto) que existe en la realidad. Los modelos normalmente describen, explican y predicen el comportamiento de un fenómeno o los componentes del mismo. Estos también pueden representar una teoría o hipótesis de cómo funciona *algo* que existe en la realidad.

Pero en el contexto del MDD debemos describir que representa ese "*algo* que existe en la realidad", ya que estos modelos deben ser relevantes para el contexto del desarrollo de software. En el mundo del desarrollo de software se usa la palabra *sistema* para describir ese *algo* (ese objeto) que generalmente es motivo de estudio. Con esto podemos ahora tratar de definir lo que es un modelo para el contexto de MDD.

Un **modelo** es una descripción de todo o parte de un sistema escrito en un *lenguaje bien definido*. En tanto que un lenguaje bien definido es aquel lenguaje que posee una estructura sintáctica (de forma) y semántica (de significado) concreta, exacta y que permite la interpretación automática por parte de los computadores[123].

Un lenguaje *bien definido* que permite el modelado para el MDD, es el Lenguaje de Modelado Unificado (UML, por sus siglas en inglés Unified Modeling Language). El MDD no está restringido a usar UML, pero éste se ha convertido en el estándar de facto para el modelado de software. El UML es un lenguaje gráfico para visualizar, especificar, construir y documentar un sistema de software.

La utilidad del UML en el mundo de MDD se puede observar en la Figura 2.11, y la podemos resumir como, por medio de utilizar diagramas visuales que son interconectados por el usuario (y que el computador es capaz de entender), el desarrollador es capaz de abstraer un sistema bajo estudio y crear un ejemplo (una copia) del sistema sobre el computador (un modelo) que está especificada por el lenguaje UML. La popularidad del lenguaje UML (o similares) se debe a que en general *un modelo* se presenta con frecuencia como una combinación de dibujos y de texto[151].

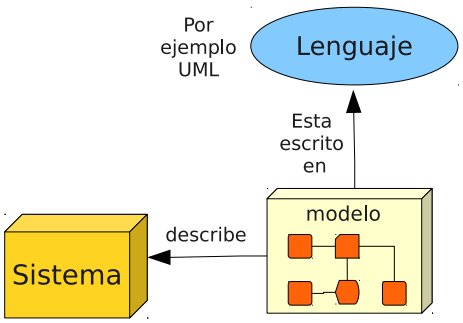

Figura 2.11: Relación entre los modelos y los lenguajes

Finalmente para este enfoque, un modelo lo componen un conjunto de elementos que describen algo abstracto o hipotético de la realidad (el sistema), y así nos ayuda a comprender la complejidad inherente del sujeto o tema bajo estudio.

2.3.3. ¿Qué es MDD?

El desarrollo dirigido por modelos es un recurso bastante nuevo en el campo de la ingeniería del software. Lo fundamental del MDD es definir un proceso que sea guiado por el uso de modelos (*model-driven*), en el que los lenguajes de modelización (visuales) se usen para integrar una enorme diversidad de tecnologías utilizadas en el diseño de sistemas informáticos [123]. Se trata de un enfoque para crear buenos diseños que puedan hacer frente a los múltiples y variados desarrollos de un sistema de software

La intención del MDD es crear modelos legibles por computadores, que puedan ser entendidos por herramientas automáticas para generar plantillas, código, modelos de pruebas y para la integración del código en múltiples plataformas y tecnologías. Para conseguir este objetivo el MDD propone que usando modelos con un alto nivel de abstracción ir transformándolo paulatinamente (de forma automática) en modelos con menos nivel de abstracción, hasta llegar al código a implementar.

Actualmente, la arquitectura dirigida por modelos (MDA[10], por sus siglas en inglés Model Driven Architecture) es una aproximación para el desarrollo de software basada en el MDD que ha sido definida y estandarizada por la OMG (de sus siglas en inglés, Object Management Group). El enfoque MDA permite realizar e integrar un modelo en múltiples modelos específicos de la plataforma de destino[151]. Este enfoque persigue como objetivo final la generación completa del software a partir de los modelos con la menor intromisión humana como sea posible en el proceso de generación del software.

2.3.4. Utilizando MDA

MDA es un "*framework*" para el desarrollo de software que define una nueva forma de construir software en la que usan modelos a distintos niveles de abstracción, para guiar todo el proceso de desarrollo, desde el análisis y el diseño hasta el mantenimiento del sistema. Para lograr este objetivo pretende separar, por un lado, la especificación de las operaciones y datos del sistema, y por el otro lado los detalles de la plataforma de implementación. MDA propone mover la filosofía de programación basada en objetos a una basada en modelos, cambiando los artefactos de software de la tecnología de programación: de "todo es

[10]http://www.omg.org/mda/

un Objeto" a "todo es un Modelo"[35]. Para ello, MDA propone:

- Definir una especificación (un modelo) independiente de las plataformas sobre la cual se construye el sistema.
- Definir un conjunto de especificaciones (modelos) para cada una de las plataformas particulares donde se construirá el sistema.
- Elegir la plataforma específica para el sistema.
- Transformar las especificaciones iníciales del sistema a la plataforma específica.

Resumiendo, la aproximación MDA utiliza y crea diferentes modelos a distintos niveles de abstracción, para luego enlazarlos o combinarlos cuando se necesite la implementación de la aplicación. Cuando los niveles de abstracción son muy altos, esos modelos son conocidos como meta-modelos (el término "meta" significa un nivel más alto de abstracción).

2.3.5. Modelos en MDA

El MDA considera diferentes tipos de modelos, de acuerdo con el nivel de abstracción[151], estos modelos reciben diferentes nombres como:

- El modelo independiente de computación (CIM, por sus siglas en inglés Computation Independent Model) donde los requerimientos del sistema se detallan en un modelo independiente de computación. Se centra en el entorno del sistema y los requisitos para el mismo. Los detalles de la estructura y el procesamiento del sistema no se muestran. En algunos casos, el CIM se refiere a un modelo del dominio, y se usa vocabulario propio de los expertos en el dominio para la especificación.
- El modelo independiente de plataforma (PIM, por sus siglas en inglés Platform Independent Model) que representa las funcionalidades del sistema sin considerar la plataforma final donde será implementado. Representa los modelos que describen una solución de software que no contiene detalles de la plataforma. Se centra en las especificaciones de aquellas partes del sistema que no cambian de una plataforma a otra. En este punto de vista debe emplearse lenguaje de modelado de propósito general (como por ejemplo UML) y en ningún caso se emplearán lenguajes específicos de plataformas.

  Al no incluir detalles específicos de una tecnología determinada este modelo es útil en los siguientes aspectos:

  - Usa modelos que son fácilmente comprensibles y manejables por los usuarios del sistema.
  - Permite la creación de una estructura y funcionalidad básica del sistema que será única, aunque el sistema vaya ser implementado sobre diferentes plataformas.

- El modelo específico de plataforma (PSM, por sus siglas en inglés Platform Specific Model) que se obtiene de combinar el modelo PIM con los detalles específicos de la plataforma seleccionada. Son modelos que contienen los detalles de la plataforma o tecnología con que se implementará la solución. El PSM es un modelo del sistema de un nivel muy bajo, muy cercano al código de la plataforma, este modelo puede incluir más o menos detalles dependiendo de su propósito.

- El modelo de implementación o mejor dicho el código, también se considera un modelo ya que el uso de lenguajes de programación abstrae de los detalles propios del hardware de computación donde éste se ejecuta.

2.3.6. Las transformaciones

Un aspecto fundamental de MDA es la definición del modelo de transformación, el cual permite automáticamente convertir los modelos. La transformación de modelos es el proceso de convertir un modelo en otro modelo del mismo sistema. Las transformaciones permiten pasar de un modelo con un nivel de abstracción dado a otro modelo con un nivel diferente, y ayudan a mantener sincronizados los diferentes modelos, si se requiere. Las transformaciones son entidades relacionales que describen reglas de mapeo, de cómo los conceptos en un meta-modelo son mapeados a los conceptos de otro meta-modelo. En otras palabras las instancias de un meta-modelo son convertidas en la instancias de otro meta-modelo, el proceso de transformación describe completamente la traslación del modelo fuente al modelo destino, esto se ilustra en la Figura 2.12.

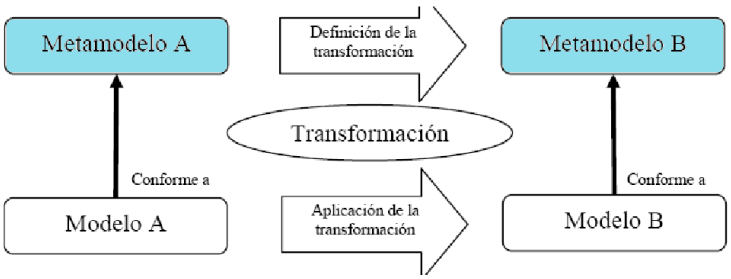

Figura 2.12: Definir una transformación entre modelos

Con el fin de definir las transformaciones entre los diferentes modelos, existen varias iniciativas relacionadas con el enfoque MDA, entre las cuales merece la pena mencionar el lenguaje QVT (Query/Views/Transformations). El QVT es un estándar de la OMG que especifica cómo puede ser escrita una transformación[152]. Este lenguaje permite especificar desde transformaciones muy simples (Query y Views), por ejemplo donde un modelo de salida está compuesto sólo por parte del modelo de entrada, hasta complejas traslaciones entre modelos (transformaciones).

Las transformaciones descritas anteriormente pueden ser aplicadas a diferentes niveles de abstracción, a los tipos de modelos mencionados. Es posible aplicar transformaciones automáticas que conviertan una especificación de PIM a PSM, conocida como transformación vertical, ya que los modelos tienen diferentes niveles de abstracción. El PSM tiene un nivel más bajo de abstracción que el PIM. También se pueden aplicar transformaciones PIM-PIM o PSM-PSM conocidas como transformaciones horizontales (donde los modelos tienen el mismo nivel de abstracción). En general todas las transformaciones anteriores son conocidas como transformaciones modelo-a-modelo, pero como a partir de los modelos PSM se puede generar automáticamente código ejecutable para distintas plataformas, este tipo de transformación se conoce como transformación modelo-a-código o modelo-a-texto. Para ilustrar lo explicado se muestra en la Figura 2.13 algunas transformaciones entre distintos modelos.

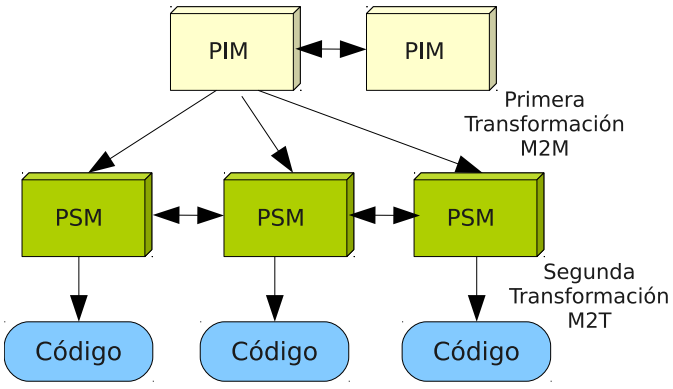

Figura 2.13: Transformación entre modelos

A la luz de este enfoque, surgen nuevos esquemas de trabajo en los que se distinguen los diferentes roles de los participantes del proceso[167] (Figura 2.14). El analista de negocio, experto en el dominio del problema, que modela una realidad por medio de CIM; el arquitecto y el diseñador que definen una propuesta de solución (transformación del CIM en PIM) y progresivamente la concretan (transformación de PIM a PIM), hasta llegar a un diseño detallado; el desarrollador o el diseñador que tiene amplio conocimiento de la tecnología y decide la

manera más adecuada como el diseño detallado se implementará en una plataforma particular (transformación del PIM a PSM).

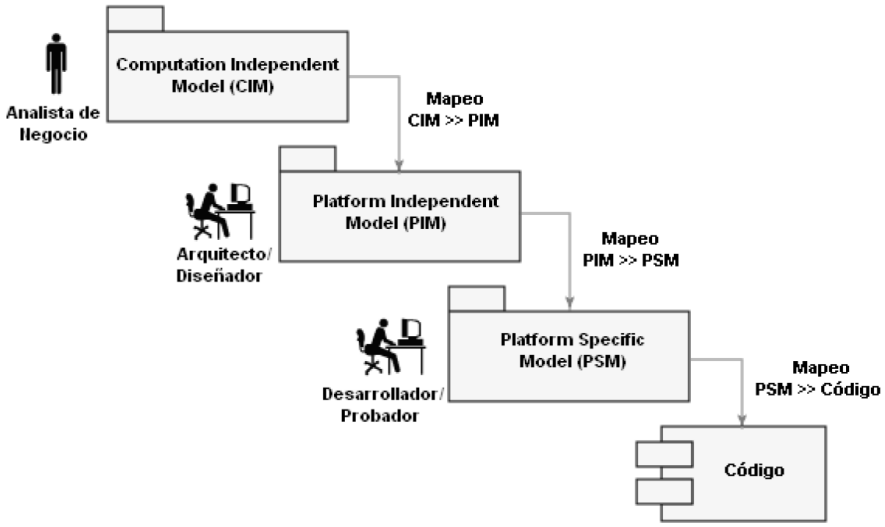

Figura 2.14: Roles y transformación entre modelos

2.3.7. Proceso de desarrollo basado en modelos

El esquema que se muestra en la Figura 2.15 toma como referencia la propuesta de desarrollo basada en modelos planteada por Kleppe y otros[123], con el propósito de compararla con el desarrollo tradicional. En los dos enfoques del proceso, cada etapa del desarrollo produce artefactos que sirven como insumo para la siguiente etapa. Un proceso típico de desarrollo de software incluye las siguientes fases:

- Recogida de requisitos
- Análisis
- Diseño
- Codificación
- Prueba
- Despliegue

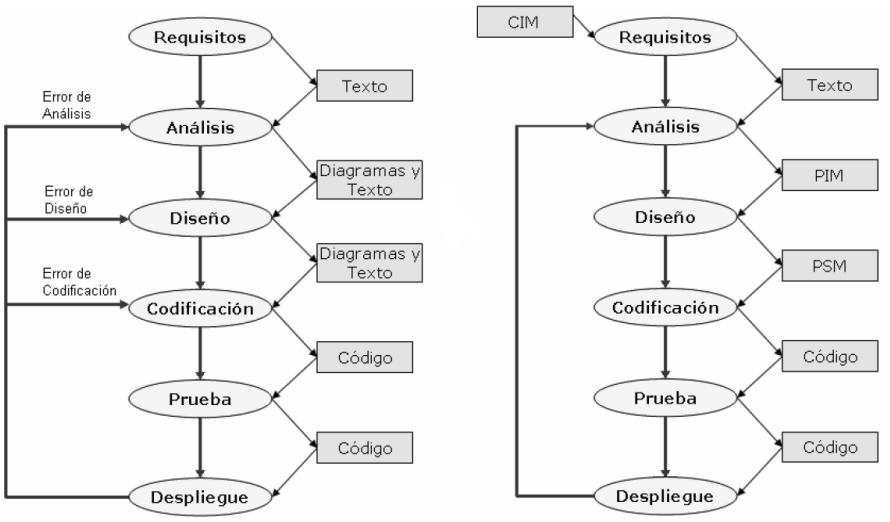

Figura 2.15: Desarrollo de software con el enfoque clásico y con MDA

La Figura 2.15 esquematiza el proceso de desarrollo de software tradicional, en este proceso toda la carga es soportada por los desarrolladores del sistema, es decir, que dependiendo del tipo de error el desarrollador debe ir atrás en el esquema y corregir el fallo, para luego hacer un serie de cambios "adicionales" en la fases siguientes que permitan sincronizar las correcciones hechas.

En cambio si observamos la Figura 2.15 en el caso de MDA, la carga para el desarrollador puede ser mucho menor (en el caso de corregir fallos e implementar algunas actualizaciones) ya que se debe ir hacia atrás para reparar el fallo (a los modelos PIM, principalmente), pero como la mayor parte de las fases se realizan con ayuda de las transformaciones automáticas, la sincronización de los cambios se traslada a la mayor parte (o al todo) el sistema.

2.3.8. Meta-modelado y MOF

El uso de MOF[11] (de sus siglas en inglés, Meta-Object Facility) en MDA es fundamental ya que tiene el propósito de lograr la interoperabilidad de las herramientas y plataformas, posibilitando evadir los problemas por la diversidad de plataformas y la evolución tecnológica. Además de MOF existen otros estándares en lo que se apoya MDA para llevar a cabo su función, por ejemplo:

- UML: empleado para la definición de los modelos independientes de la plataforma y los modelos específicos de las plataformas de destino. Es un

[11] http://www.omg.org/technology/documents/formal/mof.htm

estándar para el modelado introducido por el OMG.

- MOF[12]: establece un marco común de trabajo para las especificaciones del OMG, a la vez que provee de un repositorio de modelos y meta-modelos.
- XMI: define una traza que permite transformar modelos UML en XML para poder ser tratados automáticamente por otras aplicaciones.

MDA tienen el propósito de lograr la interoperabilidad de las herramientas y plataformas, posibilitando evadir los problemas por la diversidad de plataformas y la evolución tecnológica. El OMG plantea una arquitectura de cuatro niveles para la definición de sus estándares, en donde cada capa se define como instancia de la anterior. Esta arquitectura de modelos denominada MOF representa el nivel meta más general (M3) y tiene el objetivo de permitir la incorporación de nuevos lenguajes de modelado (meta-modelos), ver Figura 2.16.

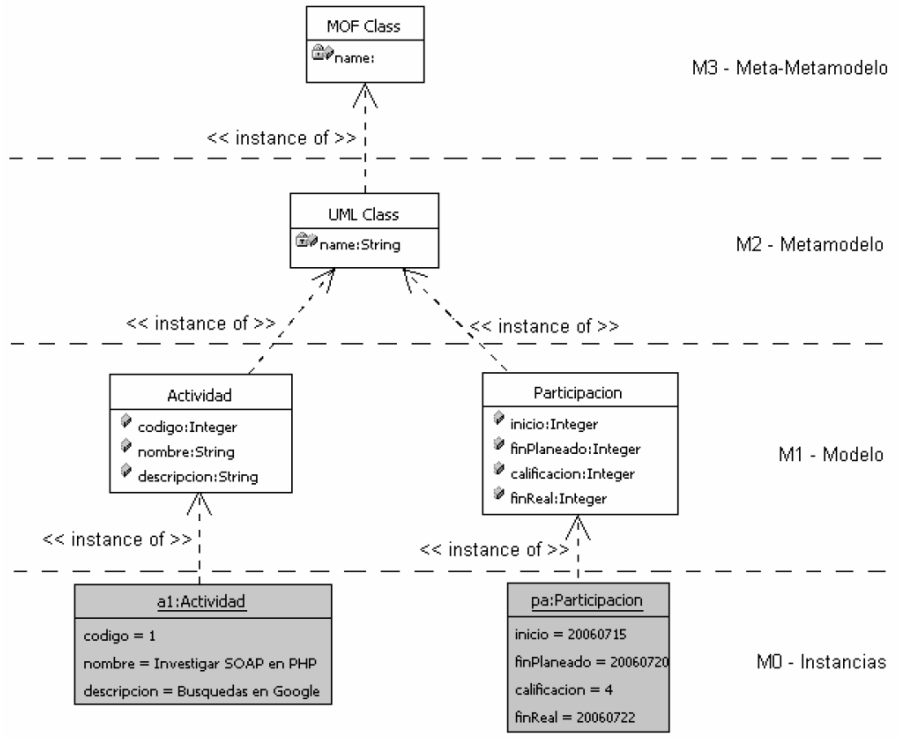

Figura 2.16: Las capas del MOF

Como comentamos anteriormente el término 'meta' significa un nivel más alto de abstracción. Un meta-modelo es un modelo que define el lenguaje para

[12]MOF Core Specification, http://www.omg.org/docs/ptc/04-10-15.pdf

expresar un modelo. Básicamente se trata de usar modelos para describir otros modelos.

Un meta-modelo especifica los conceptos y sus relaciones con el fin de construir y de interpretar los modelos. Un meta-modelo es simplemente un modelo de un lenguaje de modelado, que define la estructura, la semántica y las restricciones para una familia de modelos. En MDA, el MOF es el lenguaje que facilita la creación de los meta-modelos[153]. MOF es un estándar que describe cómo los meta-modelos deben ser realizados, por ejemplo, el UML es uno de los meta-modelos descritos utilizando MOF. A continuación se definen las diferentes capas especificadas en la arquitectura del OMG que sustenta MOF[167].

- **Capa M3 (Metameta-modelo)**. Corresponde a MOF, es una especificación que define un lenguaje abstracto para especificar, construir y manejar elementos comunes a cualquier meta-modelo.
- **Capa M2 (Meta-modelos)**. Especifica las entidades de un lenguaje de modelado. Los lenguajes que se han definido como instancias de MOF son: por ejemplo UML, CWM[13] (Common Warehouse Metamodel) y MOF en sí mismo.
- **Capa M1 (Modelos)**. Se refiere a los modelos de usuario que suelen desarrollarse en el momento de construir un sistema de información.
- **Capa M0 (Instancias)**. Describe instancias de las entidades propuestas en un modelo de un sistema de información. Es en este nivel donde pueden usarse los diagramas de objetos como instancias de las clases para verificar que se cumplen las restricciones definidas en el nivel de los modelos (M1).

Aunque parezca que MOF sólo se utiliza en los modelos, su aplicación cubre todos los elementos de MDA, como son las transformaciones. Como se comentó anteriormente, las transformaciones son entidades relacionales que describen reglas de mapeo, de cómo los conceptos en un meta-modelo son mapeados a los conceptos de otro meta-modelo. La especificación de más alto nivel de cómo es el lenguaje, donde se escriben las reglas de transformación está también basado en MOF. La Figura 2.17 ilustra este hecho.

2.3.9. Herramientas usadas en MDD

Los lenguajes son esencialmente parte del desarrollo de los sistemas de computación, los lenguajes de modelado son usados para crear modelos. Los modelos tienen un alto nivel de abstracción que pueden ocultar o enmascarar detalles de

[13] http://www.omg.org/cwm/

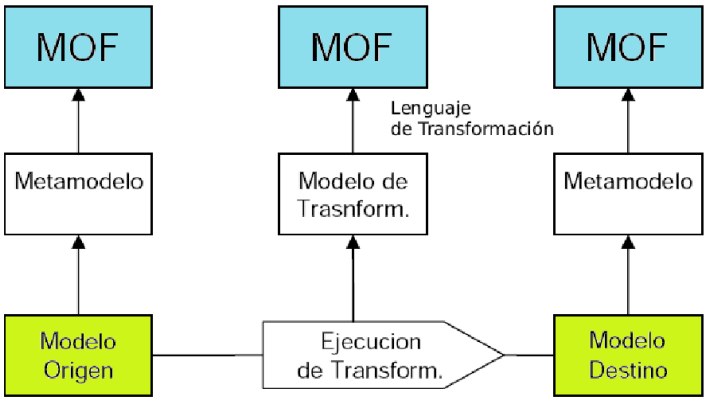

Figura 2.17: Las transformaciones y las capas MOF

la implementación y detalles específicos de la tecnología. Hay lenguajes de propósito general que ofrecen abstracciones que son aplicables a una gran variedad de dominios, como por ejemplo el Lenguaje de Modelado Unificado. También, existen lenguajes específicos de dominio DSL (de sus siglas en inglés, Domain Specific Language), que brindan un conjunto de conceptos altamente especializados para un grupo reducido de dominios, como por ejemplo el Agent UML[29].

Una herramienta para la edición de modelos visuales (al estilo UML) que debemos mencionar es el "IDE de Eclipse"[14] como uno de los entornos favoritos para implementar modelos y meta-modelos. Esto se debe a que su tecnología permite crear e instalar los *plugins* necesarios para un problema en particular. Entre los más utilizados podemos mencionar el proyecto *EMF*[49] (de sus siglas en inglés, Eclipse Modeling Framework) que es una infraestructura de modelado y generación de código que facilita la creación de herramientas y aplicaciones basadas en una estructura de modelos de datos. Desde una especificación de un modelo descrito en *XMI* (de sus siglas en inglés, XML Metadata Interchange), el *EMF* proporciona herramientas y rutinas de soporte para producir un conjunto de clases Java para el modelo. El *EMF* permite utilizar un lenguaje de meta-modelo llamado *Ecore* (usado en la descripción de los meta-modelos) que está basado en el lenguaje *MOF*.

También podemos mencionar el proyecto *GMF*[15] (de sus siglas en inglés, Graphical Modeling Framework) ofrece los componentes necesarios para el desarrollo de editores gráficos basados en *EMF*. Otra herramienta importante para crear meta-modelos es *Topcased*[15], que es un ambiente de desarrollo de software dedicado principalmente al diseño de sistemas empotrados críticos, incluyendo el

[14]http://www.eclipse.org/
[15]http://www.eclipse.org/modeling/gmf/

hardware y/o software. El modelado con *Topcased* promueve la ingeniería dirigida por modelos y sus métodos como clave de su tecnología. Los modelos creados por esta herramienta son compatibles con el *Ecore* de *EMF*.

Actualmente, las herramientas de transformación que son usadas de forma más amplia para la conversión de modelo-a-modelo o para la transformación de modelo-a-código son: *ATL*[15] (de sus siglas en inglés, Atlas Transformation Language), *Kermeta*[192] y *MOFScript*[15], las cuales son compatibles con *EMF* y pueden ser utilizadas como un *plugin* de Eclipse. Como mencionamos anteriormente, QVT es el lenguaje de transformación estándar de OMG que define como las transformaciones pueden ser escritas, pero *ATL*, *MOFScript* y *Kermeta* son herramientas que realizan un trabajo similar y son comúnmente usadas en el mundo del MDA. Específicamente, *ATL* es un lenguaje de transformación de modelos y una herramienta del campo de la Ingeniería Dirigida por Modelos (MDE, de sus siglas en inglés Model-Driven Engineering), que permite realizar transformaciones de modelo-a-modelo. De similar forma *Kermeta* es una herramienta que permite la conversión entre modelos. En cambio *MOFScript* es un lenguaje/herramienta que permite implementar la transformación modelo-a-código.

Finalmente, para el desarrollo de este trabajo se utilizaron las herramientas *ATL* y *MOFScript* para implementar los distintos mecanismos de transformación. La selección de la misma se debe a facilidad de integración con la herramienta Eclipse y a la existente comunidad que brinda soporte y ayuda a estas herramientas. Otro motivo en seleccionar este conjunto de herramientas, es que son las más utilizadas por los investigadores y desarrolladores de la comunidad del "model-driven".

2.4. Aplicando MDD al diseño de agentes

El objetivo final de MDD es mejorar la calidad de los productos y el proceso de desarrollo de software, permitiendo la reutilización de modelos y la transformación automática de la especificación de un modelo a otro, de una manera simple y transparente[123]. La arquitectura dirigida por modelos es una forma para desarrollar aplicaciones que nos permite separar la lógica de la aplicación de la plataforma utilizada para su implementación. Como se explicó en la sección 2.3.4, una especificación MDD completa divide los modelos generados en las fases de análisis y de diseño en dos conjuntos: un modelo independiente de plataforma (PIM) y en uno o más modelos específicos de plataformas (PSM), que describen cómo el modelo base es implementado en cada una de las plataformas seleccionadas. Este enfoque usado para construir 'software clásico' es perfectamente válido para desarrollar agentes.

Desde nuestro punto de vista, los estándares y la infraestructura tecnoló-

gica MDD son importantes para crear metodologías de agentes. En particular, adoptando los estándares MDD para el modelo de interoperabilidad y para la transformación automática de modelo-a-modelo se podría: (i) apoyar un proceso de desarrollo de software flexible y personalizable, y (ii) ofrecer un enfoque complementario a la definición de un meta-modelo común. Por lo tanto, su aplicación puede contribuir a reducir la brecha entre el diseño de agentes y su implementación.

Desde el punto de vista del diseño de un MAS, diferentes metodologías han identificado un conjunto de modelos para especificar las diferentes características del sistema. Estos modelos pueden ser ajustados y reflejados en diferentes meta-modelos MDD que especifican los conceptos que describen el MAS, por ejemplo los conceptos: roles, comportamientos, tareas, interacciones, protocolos, etc. Esos meta-modelos pueden ser usados para modelar sistemas multi-agente de una manera abstracta sin enfocarse en los detalles o requisitos específicos de la plataforma, como un modelo independiente de plataforma. Los meta-modelos pueden ser especificados desde diferentes puntos de vista y pueden ser representados a diferentes niveles de abstracción. Además, pueden ser parcialmente o totalmente reutilizados o combinados con otros modelos para generar un nuevo sistema.

El enfoque MDD permite la realización y la integración de un meta-modelo en múltiples plataformas específicas del modelo destino (sobre el nivel PSM). Es posible desarrollar modelos de transformaciones desde meta-modelos de agentes o de sistemas multi-agente (PIM) a modelos específicos de plataforma (PSM). Este modelo incorpora aspectos concretos de la plataforma de agentes donde serán ejecutados. Éste es un modelo de transformación vertical, ya que permite transformar un modelo más general a uno más concreto (o a código) que corresponde a la plataforma de agente en cuestión. La Figura 2.18 ilustra las posibles relaciones entre los conceptos de diferentes modelos MDD y las posibles transformaciones entre ellos. Las transformaciones a código en este ejemplo supone que existen tres plataformas para la ejecución de los agentes: Andromeda[1], JADE[30] y SPADE[78].

Con esta filosofía, un modelo de agente sirve como un puente entre los conceptos utilizados en la metodologías de diseño de un sistemas multi-agente (conocimientos, comportamientos, capacidades y así sucesivamente) y los detalles de implementación usados para implementar un agente (componentes, clases o marcos). Finalmente en los siguientes capítulos se presentará un nuevo meta-modelo de agente, así como las transformaciones necesarias para obtener el código para ejecutar el agente en dos plataformas: Andromeda y JADE-Leap. La selección de estas plataformas de ejecución (PSM) se debe: (i) estas plataformas permiten la ejecución de agentes empotrados y algunos de los agentes del sistemas son de este tipo; (ii) la plataforma Andromeda fue implementada en el marco de esta

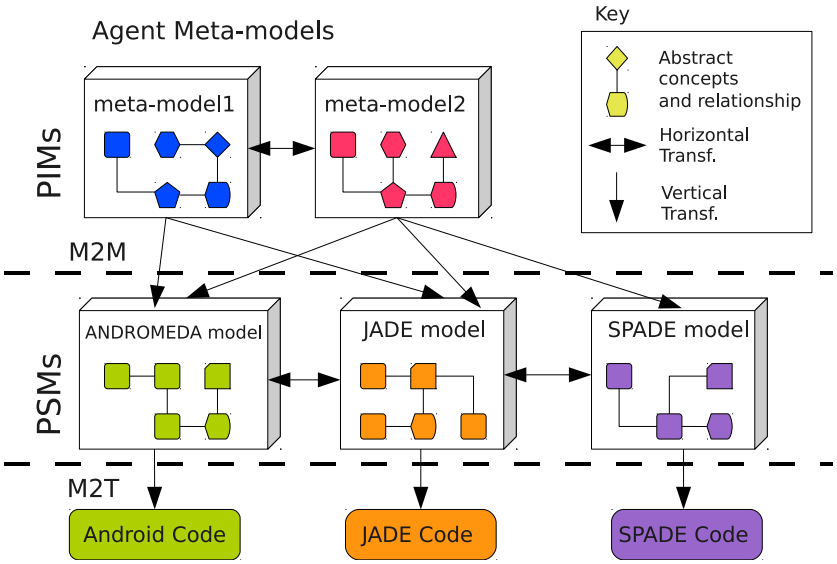

Figura 2.18: Meta-modelos MDA y sus esquemas de transformación

tesis; y (iii) la plataforma JADE-Leap (como Jade) es una de las más populares en la comunidad de agentes.

2.4.1. Metodologías MAS que poseen meta-modelos

En este punto mencionamos las tecnologías de MAS que usan relevantes abstracciones conceptuales en sus metodologías de análisis y diseño. En otras palabras, los modelos conceptuales son usados para guiar el desarrollo del MAS. Esos modelos conceptuales pueden añadirse a la estructura del MAS como unos elementos interconectables. Cada vez que se adiciona un concepto al modelo del MAS, la funcionalidad de la plataforma se incrementa. Por esta razón estos conceptos son casi siempre del tipo complementario y su inclusión permite incrementar el desempeño del MAS. Entre las metodologías más importantes destacamos[117]:

- **FAML**[34], este trabajo introduce un meta-modelo relativamente genérico orientado a agentes, cuya idoneidad permite apoyar el desarrollo MAS. Los componentes del meta-modelo se obtienen a partir de cinco propuestas existentes, orientadas a agentes (que son usadas frecuentemente): Adelfe, PASSI, Gaia, INGENIAS y Tropos. Los conceptos y sus relaciones usados en el meta-modelo son evaluados por un mapeo a dos meta-modelos orientados a agentes: TAO y ISLANDER. Los autores proponen que el meta-modelo FAML resultante, es un candidato potencial e importante para la

futura estandarización del modelado de agentes.

- **Gaia**[204] presenta un modelo que contempla los aspectos sociales en un sistema de agentes abierto, con atención especial en los objetivos sociales, tareas sociales y reglas organizacionales. Se explota las abstracciones organizacionales para brindar una guía muy clara para el análisis y diseño de sistemas complejos y abiertos. Se usan los conceptos de: rol, agente, comunicación, organización y entorno. Los principales conceptos que aparecen en la metodología se dividen en dos: abstractos y concretos. Las entidades abstractas son aquellas que son empleadas durante el análisis para la conceptualización del sistema. Las entidades concretas son empleadas en el proceso de diseño.

  El proceso de análisis de análisis se compone de:

  - Identificar los roles del sistema
  - Para cada papel identificar y documentar los protocolos asociados
  - Empleando el modelo de protocolos como base, elaborar con más detalle el modelo de roles.

  El proceso de diseño se compone de las siguientes fases:

  - Creación de un modelo de agente, agregando roles a los tipos de agentes y documentado las instancias de cada tipo de agente.
  - Desarrollo de un modelo de servicios, examinando protocolos y propiedades de viveza y seguridad.
  - Desarrollo de un modelo de conocimiento, por medio del modelo de interacción y el modelo de agente.

  El proceso de análisis y diseño se ilustra en la Figura 2.19, usando los meta-modelos correspondientes.

- **Tropos**[55] propone una metodología para el desarrollo de software la cual se basa en modelar los requerimientos iníciales, los requerimientos finales, la arquitectura de diseño, los detalles de diseño y de implementación. Utiliza los conceptos: actor, metas, planes, recursos, relaciones, descomposición de los objetivos y descomposición de planes.

  Según Castro[55], en Tropos se proponen cinco fases en el proceso de desarrollo de software:

  - Requerimientos iníciales: consiste en entender el problema desde un punto de vista organizacional, el resultado de esta fase sería un modelo

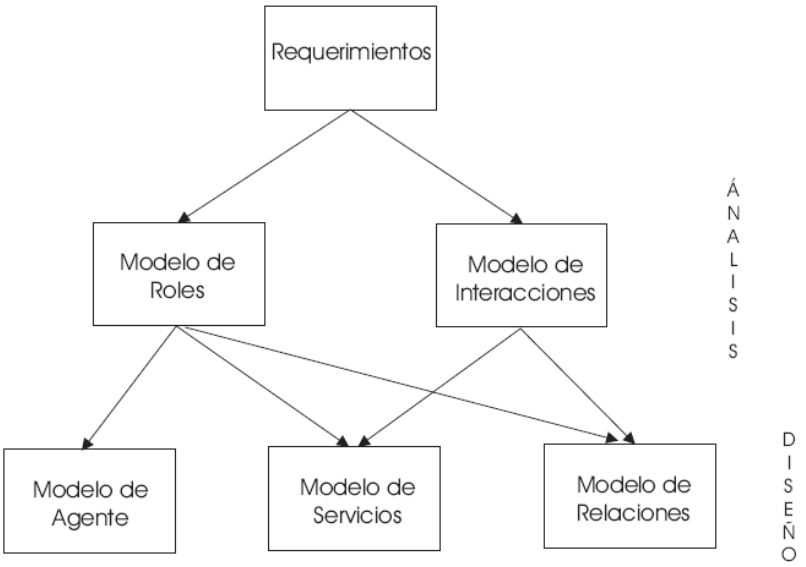

Figura 2.19: Meta-modelos de GAIA

de organización (o de actores) que incluya los actores que aparecen en el entorno, sus objetivos, así como las dependencias entre actores para la obtención de objetivos, representadas éstas últimas mediante conexiones dirigidas.

- Requerimientos finales: se trata de una especificación de requerimientos donde se describen todos los requerimientos funcionales y no funcionales. El sistema viene a ser representado por un conjunto de actores que contribuyen al cumplimiento de unos objetivos.
- Diseño arquitectónico: se define la arquitectura global del sistema, mediante conceptos como subsistema, interconexiones, dependencias y control. La arquitectura estará formada por un conjunto de componentes, expresando como trabajan de forma conjunta.
- Diseño detallado: en esta fase se definen con detalle cómo se comportan los diferentes componentes (capacidades e interacciones) identificados en la arquitectura de la fase anterior.
- Implementación: donde se sigue paso por paso la especificación del diseño detallado. Para la implementación se ha empleado JACK.

- **Ingenias**[93] proporciona una metodología y un conjunto de herramientas para desarrollar sistemas multi-agente. La estructura usa cinco puntos de

vista complementarios en el diseño de un MAS: organización, agente, tareas/metas, interacciones y entorno. El principal concepto de INGENIAS es la organización que contiene grupos y flujos de trabajo. Un flujo contiene tareas que producen interacciones, un grupo contiene agentes y roles.

- **Opera**[71] presenta un modelo de interacción entre agentes en la organización y el entorno. Opera proporciona una forma flexible para representar las interacciones y los roles jugados, porque abstrae las representaciones internas de los agentes y separa los modelos de requisitos de la organización y objetivos. Los tres principales supuestos son: (i) las entidades autónomas necesitan interacción social para alcanzar sus metas individuales, (ii) las organizaciones y/o las sociedades tienen sus propias metas y requerimientos, los cuales no son necesariamente compartidos por los participante de las entidades pero debe ser alcanzados por la actividad (coordinada) de los participantes, (iii) asume un proceso de negociación y adaptación necesario entre los requerimientos individuales, requerimientos sociales y las características con el fin de conjugar la autonomía individual con los requerimientos sociales y metas.

- **Meta-modelo Unificado**[33] este trabajo trata de unificar tres metodologías existentes: ADELFE[162], Gaia[201] y PASSI[65] por medio el estudio de sus meta-modelos y los conceptos comunes o similares en la metodologías. Compara un cierto número de características en el agente y en el sistema, tales como la estructura del agente, su organización, las capacidades de interacción o como el agente puede ser implementado. Pero los meta-modelos obtenidos fueron vistos como muy complejos de tal forma que los autores decidieron no continuar con el trabajo.

2.4.2. Metodologías MAS que aplican MDD en su diseño

Pocos trabajos han integrado el enfoque que brinda el MDD (o la MDA) en el diseño de MAS. Las más relevante son (algunas de ellas comentadas anteriormente): INGENIAS[160], MetaDIMA[96], TROPOS[161] y PIM4AGENT[99]. También mencionaremos a AML y AUML como propuesta que usan el enfoque MDD al modelado de agentes. Pero la propuesta de Perini-Susi[161] y Pavón[160] se basan en utilizar meta-modelos específicos de una metodología; en Guessoum y Jarraya[96] no se propone en obtener código ejecutable del agente. En cambio Hans y otros[99] no contempla que los modelos de agentes específicos de plataforma se ejecuten en sistemas con pocos recursos computacionales, plataformas para agente empotrados. En cambio en AML y AUML ninguna de estas herramientas genera código que pueda ser ejecutado en alguna plataforma.

- **INGENIAS**[160] muestra que el desarrollo dirigido por modelos con sistemas basados en agentes puede facilitar la implementación de métodos y herramientas para el desarrollo de MAS. Esto se ilustra con la metodología INGENIAS y sus herramientas, con el kit de desarrollo de INGENIAS (IDK), el cual se basa en la definición del meta-modelo INGENIAS para MAS, e implementa muchas de las facilidades que se requieren en el enfoque MDD.

- **MetaDIMA**[96] ayuda al diseñador a implementar un MAS en la plataforma DIMA usando MetaGen el cual es una herramienta con el enfoque MDD, dedicada a la definición de meta-modelos y modelos. MetaDIMA proporciona un conjunto de meta-modelos para que en el proceso de desarrollo del MAS se definan los PIMs y PSMs correspondientes, mediante el análisis de la aplicación. También define una librería de meta-modelos para la identificación de los conceptos utilizados y diseña las reglas de transformación para implementar el modelo.

- **TROPOS**[161] este trabajo abordó la transformación de modelos en MAS por la discusión de un ejemplo práctico utilizando la metodología de *Tropos*. El estudio se centra en las transformaciones automáticas para descomponer los planes de *Tropos* en unos diagramas de actividades de UML 2.0. Adicionalmente muestra cómo usar las técnicas de transformación para automatizar el mapeo de modelos y además se describe una herramienta CASE basada en un arquitectura modular.

- **PIM4AGENT**[99] este trabajo presenta un meta-modelo para un sistema de agentes que abstrae de las existentes metodologías orientadas a agente, lenguajes de programación y plataformas y que podría ser considerado como un modelo independiente de plataforma. Este meta-modelo define una sintaxis abstracta para un lenguaje de modelado de dominio-especifico para MAS que está actualmente en desarrollo y que proporciona la base para generar código a partir de los agentes diseñados. Esto se realiza aplicando los principios de MDD y transformaciones de modelos.

- **AML**[56] es un lenguaje visual semi-formal de modelado que permite especificar, modelar y documentar sistemas que incorporan características de MAS. El AML es especificado como extensión de UML 2.0 que cumple con la infraestructura de modelado de OMG: MDA, MOF, UML y OCL[16] (de sus siglas en inglés, Object Constraint Language). La meta de AML es de proporcionar las herramientas de ingeniería de software lista para

[16]http://www.omg.org/technology/documents/formal/ocl.htm

el uso, es decir, herramientas completas y altamente expresivas para modelar de forma conveniente el desarrollo de software basado en sistemas multi-agente.

- **Agent UML** (AUML)[28] permite la representación de los comportamientos internos a un agente y luego relaciona la descripción interna con los comportamientos externos del agente a través del uso y la extensión de los diagramas de clase UML y por la descripción de los protocolos de interacción en una nueva forma. Este trabajo extiende el estándar establecido para el modelado de sistemas orientados a objetos -UML- para que sea utilizado en el software orientado a agentes. Introduce los artefactos de software apropiados y extiende los diagramas UML para especificar y soportar el diseño de agentes.

2.4.3. MDD en la implementación de MAS

Desde el punto de vista del diseño de sistemas orientados a agentes, el desarrollo de aplicaciones consiste en cómo obtener el código de agente que podrá ser ejecutado en diferentes plataformas. Es decir, concentrar el desarrollo de la aplicación desde un meta-modelo unificado de agente, y después de esto, aplicar diferentes transformaciones para obtener implementaciones para diferentes plataformas.

En la literatura de MAS, los investigadores han formulado un conjunto de típicos meta-modelos que guían el proceso de desarrollo de MAS usando un enfoque basado en modelos. Algunos trabajos han concentrado sus esfuerzos en la creación de un modelo genérico unificado para analizar y modelar el sistema utilizando diferentes metodologías. Algunas de las propuestas más significativas son: INGENIAS[91], PIM4AGENT[99] FAML[34], Agent UML(AUML)[28] y AML[56]. Aunque estas propuestas usan componentes similares en sus meta-modelos, ninguno de ellos se enfoca en el desarrollo de un MAS como un *Sistema Ubícuo*. Estos enfoques tienen un alcance limitado en la forma de diseñar como los agentes perciben, actúan y controlan el mundo físico.

Sin embargo, muy recientemente se ha propuesto un trabajo[26] que aplica el enfoque *model-driven* para el diseño e implementación de MAS. En el trabajo se propone en implementar MAS para soportar ambientes inteligentes (AmI). En el trabajo se utilizan los meta-modelos de PIM4AGENT para el diseño del MAS, y posteriormente usan algunas transformaciones para implementar el MAS en la plataforma Malaca[16].

La diferencia de nuestra propuesta con respecto a las propuestas existentes es que nosotros proponemos un específico meta-modelo de *Entorno*, que proporciona acceso a dispositivos físicos situados en el mundo real. Por otra parte, algunas de

las propuestas (FAML, AUML y AML) definen meta-modelos de alto nivel, pero no llegan a la fase de implementación, y esto dificulta enormemente el trabajo de los desarrolladores cuando se trata de obtener el código ejecutable. Por otra parte, incluso en las pocas metodologías que incluyen la fase de implementación, en la mayoría existe una brecha entre el diseño y los modelos de implementación. Dado que hay notables diferencias entre la definición de agente en alto nivel y la implementación de la clase agente.

2.5. Sistemas Ubícuos y Ambientes Inteligentes

Esta sección presenta una introducción de la *computación ubícua*. En particular se realiza una revisión de cómo esta tecnología es utilizada para crear entornos que se conocen como *ambientes inteligentes*, que con la ayuda de los sistemas empotrados inteligentes "*inteligencia empotrada*" (*agentes*) permiten brindar la posibilidad a las personas de utilizar la tecnología de forma de que no sean consciente de ello (de forma no intrusiva) con el objetivo de mejorar en forma significativa la calidad de vida de los involucrados en tales entornos.

2.5.1. Introducción

La *Computación Ubícua* ("*Ubiquitous Computing*") también llamada *Computación Pervasiva* ("*Pervasive Computing*") es un paradigma en el cual la tecnología está virtualmente invisible en nuestro ambiente de trabajo y hogar debido a que ha sido insertada en el entorno, con el objetivo de que la misma –la tecnología– sea capaz de mejorar nuestra calidad de vida creando un *ambiente inteligente* (AmI, por sus siglas en inglés, *Ambient Intelligent*).

En la computación ubícua, la omnipresencia, que era hasta hace muy poco un atributo propio de la divinidad, está pasando a convertirse en una característica habitual de nuestra sociedad[198], con la aparición de dispositivos electrónicos de todo tipo incorporados en toda clase de objetos fijos o móviles "embebidos", "incrustados" o "empotrados" (según la terminología en uso) conectados entre sí por medio de redes, como Internet. Estos dispositivos son capaces de ofrecernos una gran cantidad de información más o menos útil; así como de proporcionar esa misma avalancha de datos sobre nosotros mismos a los demás, incluso sin que seamos completamente *conscientes* de ello[199, 139].

Pero el diseño de *ambientes inteligentes* no sólo es un objetivo de la computación ubícua, también aparece en las investigaciones de varias disciplinas como: computación móvil, redes de sensores, inteligencia artificial, robótica, computación multimedia, reconocimientos de formas, procesamiento de señales, agentes de software e inteligencia embebida[168]. Está inteligencia debe comprender el

razonamiento autónomo y actuar de manera cooperativa. Por ello un *ambiente inteligente* se refiere al paradigma en el cual las personas esta potenciadas o fortalecidas ("empowered") por el uso de entornos digitales que son *consciente* ("aware") de su presencia y su contexto, que son sensibles, adaptativos y responden a sus necesidades, hábitos, gestos y emociones[25].

Actualmente, el término computación ubícua significa la omnipresencia de computadores muy pequeños interconectados (en forma inalámbrica generalmente) que se incrustan de forma casi invisible en cualquier tipo de objeto cotidiano. Usando pequeños sensores, estos procesadores embebidos pueden detectar el entorno que les rodea y equipar a su objeto con capacidades tanto de procesar información como de comunicación[139]. Esto añade otra dimensión completamente nueva, dichos objetos podrán descubrir, por ejemplo, donde se encuentran, qué otros objetos se encuentran junto a ellos y lo que les ha ocurrido anteriormente. Se podrán comunicar también y cooperar con otros objetos "inteligentes", y acceder a toda clase de recursos en Internet. De esta forma los objetos y los aparatos podrán reaccionar y funcionar de manera sensible al contexto y parecer (máquinas) "inteligentes", sin ser realmente (seres) "inteligentes"[62].

Debido a los continuos avances en los campos de la computación, la microelectrónica, la tecnología de la comunicación y la ciencia de los materiales, esta visión de informatización completa e interconexión de objetos cotidianos podrá convertirse en una realidad palpable en un futuro no demasiado lejano[139]. Como la computación ubícua podrá originar la aparición de una serie de aplicaciones totalmente nuevas donde, por ejemplo, los objetos funcionando de forma cooperativa crearán nuevas utilidades emergentes, esta visión con el tiempo también podrá tener éxito desde un punto de vista comercial y tendrá indudablemente enormes repercusiones económicas y sociales[145]. También sacará a debate cuestiones acerca de la aceptación de la tecnología y de la creación de un mundo donde la realidad quedara estrechamente ligada a (y en algunos casos incluso mezclada con) nuestro ciberespacio basado en la información.

2.5.2. Computación Ubícua, Computación Pervasiva, Ambiente Inteligente: ¿conceptos diferentes?

Durante las últimas dos décadas los conceptos de computación ubícua/pervasiva y ambientes inteligentes, se han utilizado para describir enfoques similares. Un entorno donde existe tecnología inalámbrica no intrusiva se describe a menudo como computación ubícua, computación pervasiva o ambiente inteligente. La empresa Xerox introdujo el concepto de la computación ubícua, IBM acuñó el concepto computación pervasiva y la empresa Philips seleccionó la expresión ambientes inteligentes. Parece difícil distinguir los ambientes inteligentes de los antiguos

conceptos de computación pervasiva y la computación ubícua, especialmente si hasta los creadores y patrocinadores de estos términos parecen usarse indistintamente. Sin embargo, todo indica que las principales diferencias dependen del contexto de uso: EEUU vs Europa y academia vs industria[144, 174].

La visión de Mark Weiser en Xerox PARC con su *computación ubícua* (el término aparece en 1987)[198, 174], tiene un enfoque más académico. Weiser imaginó en sus pioneras investigaciones a las computadoras no como computadores personales, sino como sistemas no intrusivos omnipresentes de la vida cotidiana [198], y se preguntó si el agente inteligente era la metáfora de la computadora del futuro.

En cambio Uwe Hansmann, et al.[139] de IBM usó el termino *computación pervasiva* (alrededor de 1994), con una visión más industrial. El concepto computación pervasiva se utilizó en artículos de investigación como un marco donde se permite a las personas conectarse en cualquier lugar en cualquier momento (*anyplace at anytime*), es decir, a entregar información a los usuarios de computadoras donde y cuando lo puedan necesitar. Esta visión usa el eslogan: "en todas partes en cualquier momento" (*everywhere at anytime*), como el objetivo tanto de la computación pervasiva o ubícua[174], por ello, no se ven grandes diferencias entre ambos conceptos.

Ahora, el concepto ambiente inteligente (que aparece hasta 2003), es un término más Europeo, que apenas se usaba en investigaciones estadounidenses a principios de siglo[174, 25]. Sin embargo, la empresa Philips elaboró el hogar digital (*digital home*) y describió los ambientes inteligentes, cómo los entornos que son controlados por el habla y los gestos, que soportan proporcionar (algo) cualquier cosa, en cualquier momento y en cualquier lugar (*anything, anytime, anywhere*)[62, 25]. El ambiente inteligente se utiliza por ejemplo para describir la tecnología que desaparece en su entorno, y que sirve como un puente entre el mundo real y el digital[145]. Un enfoque que es claramente similar y que incluye a la computación ubícua o pervasiva.

Sin embargo, existen investigadores que describen leves diferencias entre estos términos[24], principalmente entre el ambiente inteligente y la computación ubícua/pervasiva (por ejemplo, la ubiquidad no implica inteligencia, la inteligencia no implica omnipresencia). En otros estudios se trata de diferenciar estos términos en función a las tecnologías predominantes en los diferentes escenarios de uso[174]: la computación ubícua se relaciona más con el trabajo en el entorno (*anywhere, anytime*); La computación pervasiva se refiere más a la implementación de redes de dispositivos; y el ambiente inteligente se relaciona más con la aplicación de sensores inteligentes. En general, se puede concluir que diferenciar los conceptos resulta poco importante para este trabajo, ya que de forma práctica estos términos tratan enfoques o visiones muy similares.

2.5.3. Evolución de los sistemas ubícuos

Mark Weiser, en 1991 describió su visión de lo que él llamaba computación ubícua, hoy llamada computación pervasiva (ambos términos se usaran de igual forma en está memoria). La esencia de su visión era la creación de entornos repletos de computación y de capacidad de comunicación, todo integrado de forma inapreciable junto a los personas. La visión de Weiser estaba bastante alejada de su época, entre otras razones porque no existía la tecnología necesaria para llevarlo a cabo[198, 199]. Pero después de más de una década de progreso en el campo de los dispositivos hardware, las criticadas ideas de Weiser en el 1991 ahora son productos comercialmente viables:

- Computadores de bolsillo
- Redes inalámbricas
- Sensores muy avanzados

Todo esto es posible a los constantes avances en microelectrónica que se han convertido en algo común: la *ley de Moore*, formulada en los años sesenta por Gordon Moore, afirma que la capacidad de computación disponible en un microchip se multiplica por dos aproximadamente cada 18 meses y, de hecho, esto ha resultado ser un pronóstico extraordinariamente exacto del desarrollo del chip desde entonces[141]. Se puede observar también un crecimiento exponencial comparable en otras áreas, como por ejemplo en la capacidad de almacenamiento y el ancho de banda para la comunicación. Visto de otra forma, los precios para la funcionalidad microelectrónica con la misma capacidad de computación están bajando gradualmente de forma radical.

Esta tendencia que no cesa producir computadores cada vez más pequeños, lo que anuncia un cambio de paradigma en las aplicaciones informáticas[178]: se montarán procesadores, dispositivos de memoria y sensores para formar una amplia gama de aparatos electrónicos de información baratos, que estarán conectados sin cables entre sí y a Internet, serán construidos de forma personalizada para realizar tareas especificas. Estos componentes microelectrónicos se podrán incrustar además en casi cualquier tipo de objeto cotidiano, lo que le añadirá sensibilidad "*smartness*"; por ejemplo modificando su comportamiento dependiendo del contexto en que se encuentre del objeto.

Al final, el procesamiento de la información y las capacidades de comunicación quedarán integrados en objetos que, por lo menos a primera vista, no parecerán de ningún modo aparatos eléctricos, de esta forma las capacidades de la computación se volverán *ubícuas*[198]. Weiser vio la tecnología solamente como un medio para un fin y como algo que deberá quedar en segundo plano para permitir al usuario

concentrarse completamente en la tarea que está realizando. En este sentido, considerar el computador personal como herramienta universal para la tecnología de la información sería un enfoque equivocado, ya que su complejidad absorberá demasiado la atención del usuario.

Según Weiser, el computador como dispositivo dedicado deberá desaparecer, mientras que al mismo tiempo deberá poner a disposición de todo lo que nos rodea sus capacidades de procesamiento de la información. Weiser ve el termino Computación ubícua en un sentido más académico e idealista como una visión de tecnología centrada en la persona, mientras que la industria ha acuñado por eso el término Computación Pervasiva, con un enfoque ligeramente diferente (podemos decir en forma general que es el enfoque práctico de Computación ubícua actualmente). Aunque su visión siga siendo todavía integrar el procesamiento de la información en objetos cotidianos de forma casi invisible, su objetivo principal es utilizar tales objetos en un futuro próximo en el ámbito del comercio electrónico y para técnicas de negocios basados en la Web. Está variante pragmática de computación ubícua está empezando ya a tomar forma, y para algunos autores se describe como la era "post-PC"[142].

2.5.4. Retos y taxonomía de los sistemas ubícuos

Como hemos descrito uno de los principales objetivos de la computación ubícua es hacer desaparecer a los dispositivos computacionales haciéndolos situarse en un segundo plano. Este objetivo de crear dispositivos que se mezclen en la vida cotidiana hasta que lleguen a ser indistinguibles supone una potencial revolución que puede hacer cambiar el modo de vida diario. Las personas se centraran en las tareas que deben hacer, no en las herramientas que utilizan, porque se pretende que esas herramientas pasen desapercibidas.

El significado de enviar la computación a un segundo plano, está referido a dos conceptos diferentes[139]: El primero es el significado literal de que la tecnología de la computación se debe integrar en los objetos, cosas, tareas y entornos cotidianos. Y la segunda es que esta integración se debe realizar de forma que la introducción de la computación en estas cosas u objetos no interfieran con las actividades para las que son usadas, y que siempre proporcionen un uso más cómodo, sencillo y útil de esos objetos.

Estos objetos cotidianos en los que se integra la tecnología de la computación pasan a tener una serie de propiedades que permiten la creación del entorno ubícuo buscado. Estas son algunas de esas propiedades[142]:

- **Comunicación entre dispositivos**: todos estos objetos dotados de capacidad de computación también tienen capacidad de comunicación, y no

solo con el usuario, sino con los demás objetos integrados que haya a su alrededor.

- **Memoria de eventos**: además de poder comunicarse entre ellos e interactuar con los usuarios, estos dispositivos tienen capacidad de memoria y pueden utilizar esta memoria para una mejor interacción con el resto de dispositivos.

- **Sensible al contexto**: estos objetos son sensibles al contexto, es decir, se adaptan a las posibles situaciones, como la situación geográfica, los dispositivos que hay a su alrededor, las preferencias de los usuarios, y actúan dependiendo de ese entorno que los rodea.

- **Reactivos**: estos objetos reaccionan a ocurrir determinados eventos, que pueden percibir en su entorno mediante sensores o a través de la interacción con otros dispositivos.

Ahora desde punto de vista de la evolución histórica de la computación, algunos investigadores plantean que la computación ubícua representa el próximo paso (y un gran paso) de dos ramas de la computación[139], como lo son *los sistemas distribuidos* y la *computación móvil*, ya que los problemas encontrados en la computación pervasiva, fueron problemas tratados en estas dos ramas de la computación y que ya algunos fueron resueltos y por ello su solución podría aplicarse directamente. Pero también se plantean nuevos retos y problemas que no tiene analogía con los problemas de las anteriores ramas.

- **Computación distribuida**: de manera muy general esta rama fue creada de la intersección de la computación personal y de las redes de área local. Sus investigaciones se inician en los años 70 y se fundamentaron en encontrar herramientas que permitiesen trabajar con dos o más computadores conectados a una red ya sean estáticos, dinámicos, inalámbricos o alambrados.

- **Computación móvil**: esta nace a partir de los años 90, con el auge de los computadores portátiles y los investigadores ahora se enfrentan al problema de investigar/crear herramientas que permitan operar con sistemas de computación distribuida pero ahora con clientes móviles.

Para ilustrar los problemas que se encuentra en la computación ubícua[178], y como esta proviene de la evolución de los sistemas distribuidos y de la computación móvil, se muestra la Figura 2.20.

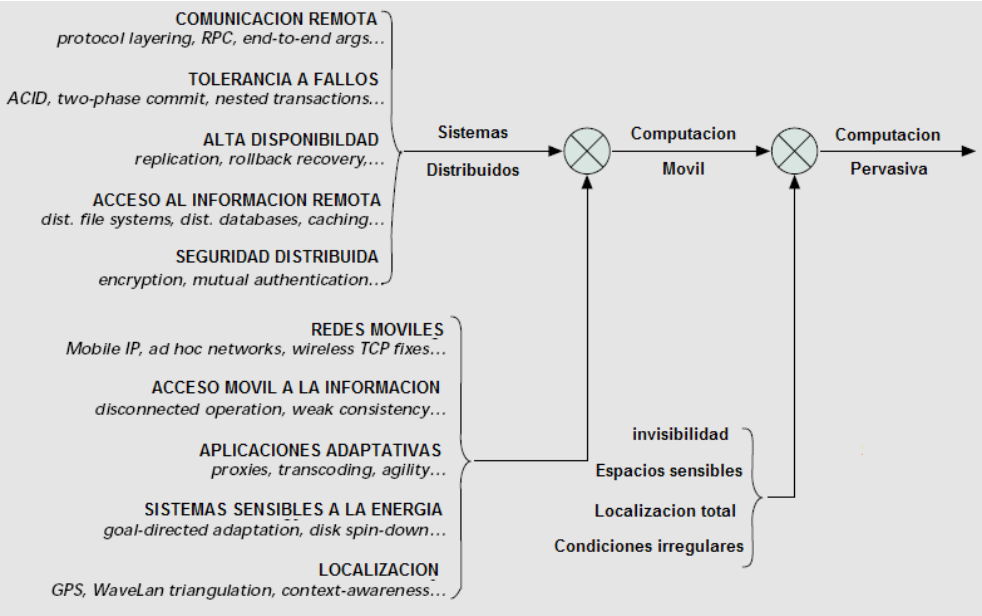

Figura 2.20: Taxonomía de la Computación Ubícua

Esta corresponde a una taxonomía de los problemas investigados por la computación ubícua. Si observamos la Figura 2.20 de izquierda a derecha veremos como a la computación ubícua se le incrementan los problemas encontrados en la otras dos áreas, lo que quiere decir, que es mucho más complejo implementar un sistema con computación ubícua que un simple sistema distribuido con una robustez comparable, o que un sistemas de computación móvil, aunque use técnicas o herramientas análogas.

2.5.5. Inteligencia empotrada

Los computadores llevan ya un largo camino recorrido. Originalmente se concibieron como máquinas para reemplazar a computadoras humanas: personas, principalmente mujeres, que manejaban calculadoras mecánicas de escritorio para computar tablas matemáticas. Más adelante, los computadores lentamente evolucionaron hasta llegar a convertirse *en computadores como herramientas* para el tratamiento de la información, para ayudar a las personas, más que para reemplazarlas. La simbiosis persona-computador era una idea radical en su tiempo y sólo llegó a ser tendencia dominante con la aparición de los computadores personales años más tarde.

Durante las dos últimas décadas y más notablemente con la aparición de la *World Wide Web*, las máquinas de computación han evolucionado más hacia *el*

computador como medio, integrando excelentes medios de comunicación con un uso generalizado rápidamente para la comunicación y la información. Ahora, con la aparición de la computación ubícua y la amplia integración de las tecnologías de la información, podemos comenzar a ver el computador como material de diseño para la creación de productos y dispositivos cotidianos -un ingrediente- en vez de un producto final[145].

De la misma manera que nuestro concepto del computador ha cambiado a lo largo del tiempo, nuestro concepto de las personas que utilizan tecnología informática ha evolucionado. Esto es evidente en nociones tales como operador, usuario y, más recientemente, consumidor. La noción de operador nos lleva a una época inicial en el uso del computador, situando a las personas fuera del sistema, lo que llevó a ver el estudio de sistemas de computación como una Ciencia dura. La noción de simbiosis persona-computador dio comienzo a una segunda época en el uso del computador en la que las personas llegaron a ser consideradas como parte de los sistemas interactivos, si bien una parte periférica, como queda patente en la noción todavía vigente de interfaz de usuario (originalmente concebida sólo como una interfaz más que el computador puede soportar). En esta perspectiva más amplia, el estudio de sistemas, y en particular el factor humano, era difícil de acometer en la tradición de la Ciencia de la Computación, finalmente dando lugar a la Interacción Persona- Computador como una disciplina por derecho propio. En la actual perspectiva de la Interacción persona-computador, las personas se caracterizan por tareas de trabajo para cuyo desarrollo aplican computadores de forma *explícita*.

La computación ubícua desafía ahora esta perspectiva, propone una nueva época en el uso del computador en la que las personas se rodean de un gran número de dispositivos informáticos, utilizados de forma implícita en la realización de actividades cotidianas. Ahora estos dispositivos informativos son artefactos de la vida cotidiana que tienen incrustados o embebidos capacidades de computación, para realizar tareas propias de las ciencias de la computación: de interfaz y cálculo[25].

Estas capacidades de computación introducidas en los dispositivos de uso común es lo que se conoce como *sistemas empotrados*[179, 145]. Si ahora estos sistemas empotrados están dotados de facultades como aprendizaje, razonamiento, autonomía, cooperación y planificación (soportadas por técnicas de *inteligencia artificial*), estos sistemas serán conocidos como de *inteligencia empotrada*[64].

Pero no sólo debemos pensar que la inteligencia se deba a que los algoritmos de planificación o razonamiento, etc., puedan ser ejecutados en estos dispositivos informáticos (en vez de un computador personal por ejemplo). Sino a nuevas interfaces de interacción con estos dispositivos (relación persona-computador), que la relación se realice en términos más naturales con las personas: reconociendo el habla, los gestos, las emociones, como también sus posibilidades de

comunicación con otros artefactos del entorno[168]. Para así poder ser capaces de sustentar la frase "*el comenzar a ver el computador como material de diseño para la creación de productos y dispositivos cotidianos -un ingrediente- en vez de un producto final*".

La fuerza motora que hay detrás de estos desarrollos tecnológicos dinámicos es la microelectrónica, donde la progresión ha cumplido verdaderamente la Ley de Moore durante estas décadas. Los recientes logros en los campos de los micro-sistemas y de la nano-tecnología están adquiriendo también cada vez más importancia[199].

Un ejemplo son los micro-sensores incrustados adecuados para detectar una gran cantidad de parámetros del entorno. Otro desarrollo interesante es el de los sensores de radio que pueden transmitir los cambios de presión o de temperatura a varios metros de distancia sin ninguna fuente explícita de energía. La energía necesaria para transmitir los datos capturados, y también para añadir un código de identificación individual a los datos, se obtiene durante el propio proceso de medición que utiliza materiales piezoeléctricos.

Las denominadas etiquetas inteligentes (*smart labels*) o etiquetas de radio (*radio tags*) también funcionan sin una fuente incorporada de energía. Estas etiquetas contienen un transmisor/receptor (*transponders*) que reciben una señal de alta frecuencia desde una distancia de hasta dos metros. La energía de la señal es utilizada por el receptor para descodificar el mensaje, potenciar sus capacidades internas de procesamiento de la información y devolver una respuesta. Esto permite transmitir y recibir hasta varios cientos de bytes sin cables en un espacio de unos cuantos milisegundos. Los transmisores tienen unos tamaños de unos cuantos milímetros cuadrados, son tan delgados como una hoja de papel y están disponibles como etiquetas de dirección flexible y un costo muy bajo.

Por muy poco dinero ya se pueden fabricar completos sistemas computarizados en un solo chip, de solamente unos pocos milímetros cuadrados, y con cientos de megabytes de memoria (suficientes para un sistema operativo sencillo). Esta tecnología se utiliza principalmente para tarjetas inteligentes, pero se puede encontrar también en sistemas incrustados, con procesadores integrados en todo tipo de aparatos para realizar tareas de control. Estos procesadores, junto con sensores apropiados, interfaces de entrada-salida y capacidades de comunicación, son los componentes básicos que podrán llegar a *transformar en "inteligentes"* a los objetos del mundo real.

2.5.6. Inteligencia empotrada usando agentes

En esta sección supondremos que la teoría de agentes es conocida, ya que fue descrita anteriormente en este capítulo, específicamente en la sección 2.1.

Para entender como la computación ubícua puede crear un entorno dotado

con inteligencia (un ambiente inteligente), comienza por comprender el proceso de automatización del ambiente, éste puede ser visto como un ciclo de percepción del estado del entorno, luego razonar sobre ese estado junto con los objetivos/tareas previstas, para así proponer un conjunto de acciones y actuar sobre el entorno para logra cambiar el estado[139, 178]. La percepción está en el fondo del proceso con los sensores monitorizando el entorno usando componentes físicos, y permitiendo a la información a estar disponible por medio de la capa de comunicación. Así las bases de datos guardarán está información, de la cual se extraerá conocimiento por medio de algoritmos de minería de datos y modelos de predicción. Este nuevo conocimiento será utilizado por los algoritmos de decisión para así poder realizar sus acciones correspondientes, que está en el tope del proceso. Estas acciones son comunicadas por las capas de comunicación a los actuadores físicos y dispositivos de control que tienen como objetivo cambiar el estado del ambiente.

Dado que la investigación de ambientes inteligentes está gobernada por el mundo real, los sensores y actuadores juegan un papel muy importante, ya que nos permiten monitorizar, observar e interactuar con el mundo físico en tiempo real[62]. Además, nos permite tomar las acciones apropiadas de control sobre el entorno. Pero el manejo del conocimiento (la inteligencia) del ambiente puede ser implementada y modelada con el paradigma de *agente software*, como el que se mostró anteriormente en la Figura 2.1 (el enfoque de Russell[176]), donde los componentes físicos son los que le permiten al agente *sentir y actuar* sobre el ambiente.

Desde la perspectiva de la computación ubícua la inteligencia del entorno viene dada entre otras cosas, por una red dispositivos empotradas (con capacidades de computación empotradas) que pueden interactuar con el usuario que vive o trabaja en ese entorno[168, 24]. El reto es cómo manejar y configurar ese conjunto de dispositivos o el sistema de artefactos en el ambiente ubícuo de una manera transparente y no intrusiva, sin que el usuario sea consciente de esta configuración y sin que el usuario tenga que manualmente configurar estos dispositivos para que realicen las funciones deseadas. Como comentamos en la introducción en los ambientes inteligentes las personas están potenciadas a través de ambientes tecnológicos que son capaces de "percibir" su presencia y la del contexto para sentir, adaptarse y responder según sus necesidades. Así, que resumiendo un entorno para este paradigma se caracterizan por ser *ubícuo, transparente e inteligente*[179, 182]:

- **Ubícuo:** debido a que el usuario está inmerso y rodeado de un conjunto interconectado de artefactos (o de sistemas) empotrados.
- **Transparente**: dado que los equipos de computación son invisibles al usuario como si no existieran y están completamente integrados al entorno.

- **Inteligente:** debido a que el sistemas puede reconocer a las personas que viven o trabajan en el ambiente y es capaz de programarse a si mismo según las necesidades aprendidas de cada uno de ellos (las personas).

Por todo lo anterior, muchas investigaciones usan el paradigma de **agente software** para implementar los ambientes inteligentes[9, 69, 73, 127, 177, 186], dado que existe mucho de aprendizaje y adaptación para crear la inteligencia del entorno. Esta visión de crear ambientes inteligentes utilizando agente inteligentes empotrados en el entorno facilita que ellos puedan controlar el ambiente de acuerdo con las necesidades y preferencias del usuario. Los agentes inteligentes son artefactos computacionales que tiene la capacidad de razonar, planificar y aprender; son computadores embebidos que contienen esos agentes y que son normalmente llamados como "agentes-empotrados" (agentes-embebidos). Cada agente empotrado es una entidad autónoma, que tiene la parte o toda la inteligencia del dispositivo, que además tiene conexión de red que le permite comunicarse y cooperar con otros agentes empotrados, para formar así un sistema multi-agente empotrado[98].

Aunque en este enfoque también existe un problema, los mecanismos de inteligencia que poseen los agentes empotrados demandan por lo general grandes cantidades de recursos de computación, pero los sistemas empotrados también se caracterizan por sus limitadas capacidades de procesamiento y memoria disponible en los dispositivos computacionales, existiendo un compromiso muy estrecho de inteligencia y capacidad de procesamiento. Pero este problema puede ser atenuado utilizando técnicas de computación distribuida o de computación móvil, que como ya conocemos se utilizan en la computación ubícua.

2.5.7. Áreas de aplicación de la computación ubícua

Los principales tópicos de investigación encontrados en las publicaciones y eventos científicos de este campo se pueden resumir a continuación:

2.5.7.1. Desarrollo de habitáculos inteligentes.

Existen varios proyectos de investigación para crear entornos controlados con las características de los ambientes inteligentes, como lo son el proyecto que realiza Georgia Tech enfocado en un sistema basado en el contexto con ubícuidad en las capacidades sensoriales conocido como *Aware Home Project*[119]. El Tecnológico de Massachusetts MIT lleva un proyecto conocido como *Oxygen*[175], el cual crea un ambiente inteligente empotrando en el espacio gran cantidad de dispositivos de computación e interfaces como cámaras, micrófonos, etc., permitiendo al usuario comunicarse a través del habla, visión y reconocimiento de gestos.

Adicionalmente existen redes que pueden cambiar dinámicamente su configuración para satisfacer necesidades del usuario, soportando una gran cantidad de dispositivos manuales y con acceso móvil, como PDA, computadores portátiles y teléfonos móviles, etc.

Igualmente existen otros proyectos importantes, como el hogar inteligente (*MavHome: Smart Home*)[63] que es un desarrollo multi-disciplinario basado en la tecnología de agentes, que es muy similar al proyecto Oxygen, en donde el usuario interactúa con el entorno a través de interfaces naturales como el habla, visión para adecuar el entorno a sus necesidades. Corresponde a un "hogar inteligente" donde los sensores son invisibles al usuario y captan la presencia de la persona y que actividades realiza en el mismo.

En general, estos ambientes permiten verificar y probar nuevos sensores y herramientas para crear entornos inteligentes. Además permite medir los efectos psicológicos, de interacción, de estado anímico de las personas al estar inmerso en tales ambientes[145], es decir, los estados emocionales del humano al manejar este tipo de tecnología.

2.5.7.2. Desarrollo de nuevos sensores e interfaces.

Existen una gran cantidad de investigaciones en este sentido con el objetivo de ser capaz de percibir la información tal como lo hacen los humanos, por ejemplo con gestos y en el tono de la voz. Pero no sólo basta tener la capacidad de percibir, sino también de expresar la información como la hacen las personas, por ejemplo tratando de mostrar emociones.

Un ejemplo claro de esto, es la investigación realizada por la empresa Philips, conocida como iCat[193], un robot que es capaz de jugar con un niño el juego de Tic-Tac-Toe, tal como se ve en la Figura 2.21.

Figura 2.21: Robot iCat

Lo más resaltable de esté robot son las interfaces de entrada y salida, ya que es capaz de reconocer los comandos de voz del niño y, lo más importante, responde de igual forma, con una voz sintética y expresando algún tipo de emoción en su rostro: boca, ojos, cejas y párpados.

Otra investigación a resaltar es la cuchara[59, 64] y la tetera inteligente[64] desarrollada por el MIT Media Lab., para medir la calidad del alimento que se prepara. En la tetera se perciben variables del ambiente y el té, para determinar lo bueno que ha hecho el té: como temperatura, viscosidad de la preparación. También existe una cantidad considerable de investigaciones que tratan de usar los sistemas de visión como interfaces de entrada a los sistemas ubícuos, para tratar de leer las expresiones o emociones del rostros humano[179]. También existen sistemas tratan de reconocer las emociones a partir del habla, para ello se extraen características de la voz de la persona (se procesa el tono, el volumen, el timbre de la voz).

2.5.7.3. Redes de sensores.

Otra área destacables, donde los sensores no sólo pueden medir alguna variable física, sino que con ayuda de técnicas de negociación y acuerdos de los sistemas multi-agentes, puedan agruparse para hacer que el conjunto funcione mejor que la suma de sus partes[197]. Es decir, crear un sistema de sensores inteligentes que puedan cooperar para medir y seguir algún fenómeno físico, como también que el sistema sea tolerante a fallos en el momento de presentar la avería de un sensor, en ese caso otro agente pueda ocuparse para así poder cumplir con la tarea[14, 197].

Una red de sensores, se puede ver como una red de computadores empotrados (nodos), equipados con sensores, que colaboran en una tarea común. Las redes de sensores están formadas por un grupo de sensores con ciertas capacidades sensitivas y de comunicación inalámbrica los cuales permiten formar redes *ad-hoc* sin infraestructura física preestablecida ni administración central[61]. Esta clase de redes se caracterizan por su facilidad de despliegue y por ser auto-configurables, pudiendo convertirse en todo momento en emisor, receptor, ofrecer servicios de encaminamiento entre nodos sin visión directa, así como registrar datos referentes a los sensores locales de cada nodo. Otra de sus características es su gestión eficiente de la energía, que les permite obtener una alta tasa de autonomía que las hacen plenamente operativas[14].

La idea de estas redes es repartir aleatoriamente estos nodos en un territorio grande para que se organicen automáticamente, donde los nodos observan y miden hasta que sus recursos energéticos se agoten[13]. Los atributos “pequeño”, “barato” y “autónomo” dieron a conocer la idea como *polvo inteligente (smart dust)*. Es solo cuestión de tiempo para reducir hasta esto niveles estos dispositivos

electrónicos.

2.5.7.4. Manejo de la energía.

Este es campo de intenso estudio ya que es una de la debilidades más fuertes de la computación ubícua, ya que al trabajar con sistemas embebidos muchos de sus dispositivos utilizan baterías, y al incorporar inteligencia en estos sistemas empotrados, los usos de recursos de computación aumentan de forma considerable, al igual que el gasto energía[178].

Como es de esperarse las investigaciones en este campo se centran en la microelectrónica, para buscar dispositivos que usen de forma eficiente la energía[88]. Vale la pena mencionar que existen técnicas de software que también permiten obtener métodos de ahorro de energía, como las herramientas usadas en la negociación y cooperación de sistemas multi-agentes que permiten a los dispositivos trabajar en colaboración para buscar reducir el consumo de la batería: siendo egoísta a la hora de hacer una tarea o al contrario colaborar con algún agente o dispositivo que tenga su batería baja. Otras investigaciones a resaltar son las que tratan de desarrollar lenguajes de programación que pueden conocer el consumo de recursos que tiene cada instrucción al ser ejecutada, por ejemplo cuanta batería se gasta al hacer un conjunto específico de instrucciones[64].

2.5.7.5. Contexto, localización y seguimiento.

Las aplicaciones ubícuas tienen como objetivo proporcionar la información correcta a los usuarios, en el momento adecuado, en el lugar correcto y en el dispositivo correcto. Para lograr esto, un sistema debe tener un conocimiento profundo, y se puede decir, una *comprensión* de su entorno, las personas y los dispositivos que existen en ella, sus intereses y capacidades, así como las tareas y actividades que se están llevando a cabo. Toda esta información cae bajo la noción de *contexto*.

Muchas interpretaciones de la noción de contexto han surgido en varios campos de investigación[37, 46, 40, 27, 57]. Sin embargo, existen algunos puntos en común entre estas nociones: (i) el contexto se refiere a la información que describe el estado actual o situación de una entidad, que puede ser una persona (por ejemplo, el contexto de usuario), lugar u objeto; (ii) la información de contexto del usuario es una necesidad en aplicaciones de inteligencia ambiental para prestar servicios personalizados a los usuarios de una forma intuitiva e inteligente para apoyar sus actividades cotidianas.

El contexto tiene a menudo un impacto significativo en la forma en que los seres humanos (o máquinas) actúan y en la manera de interpretar las cosas. Por otra parte, un cambio en el contexto provoca una transformación en la expe-

riencia de los humanos (o entidades) en el entorno. Si bien la comunidad de las ciencias de la computación modeló el contexto como un concepto que describe la ubicación del usuario[46], en los últimos años este concepto ha sido considerado no sólo como un estado, sino parte de un proceso en el que los usuarios están involucrados[40, 166]. Un proceso activo que trata con la manera en que los humanos tejen su experiencia dentro de todo su entorno, con el objeto de darle sentido.

Se han propuesto varios modelos de contexto, para apoyar aplicaciones sensibles al contexto[37, 104, 46]. Sin embargo, en inteligencia artificial y en ambientes inteligentes, uno de los principales objetivos al modelar el contexto es usar técnicas de razonamiento existentes para permitir el razonamiento contextual. Los enfoques más usados son los de Lógica Proposicional del Contexto y Modelos Semánticos[40]. Mientras que la primera aproximación presenta el contexto como un "objeto primario" de una teoría lógica, en el segundo, el contexto se percibe como "una teoría parcial y aproximada del mundo desde la perspectiva de un individuo". La necesidad de modelar con éxito el contexto, es para permitir el razonamiento y proporcionar mecanismos expresivos para aprovechar el contexto, como lo demuestran las aplicaciones para la Web Semántica.

La necesidad de razonamiento en los sistemas sensibles al contexto se deriva de las características básicas de los datos de contexto. Principalmente se caracterizan por: la *imperfección* e *incertidumbre*[37]. Si además tenemos en cuenta la rapidez con que cambia el contexto y la enorme cantidad de datos disponibles, las tareas de razonamiento no son tareas fáciles. En general, la gestión del conocimiento en Inteligencia Ambiental debe permitir: (i) Razonamiento con los datos de contexto dinámicos y ambiguos; (ii) La gestión de gran cantidad de datos de contexto, teniendo en cuenta las capacidades de cómputo restringidas de algunos dispositivos; y (iii) la inteligencia colectiva, apoyado en el intercambio de información y el razonamiento distribuido entre las entidades del entorno.

Algunos de los trabajos en entornos sensibles al contexto se han basado en la localización como eje primario[105]. La localización es un concepto poderoso, cuando relaciona el uso del espacio por parte de las personas para la organización, la comunicación y la resolución de problemas. La localización es una variable muy específica que se maneja por medio de modelos geométricos, simbólicos e híbridos, que han sido ampliamente estudiados y analizados.

Por ello, se han desarrollado una amplia gama de tecnologías del posicionamiento para hacer disponible la información de localización como contexto para sistemas computacionales[139], por ejemplo las balizas infrarrojas para señalización en edificios o el Sistema de Posicionamiento Global (GPS, por sus siglas en inglés Global Position Systems) en entornos al aire libre. Sin embargo, la precisión de estas tecnologías es pobre comparada con la percepción humana del espacio, produciendo errores problemáticos cuando se utilizan como apoyo en

trabajos espaciales, o en ambientes interiores.

Una importante limitación de la localización como contexto es la que proporciona una descripción estática del entorno de un sistema, sin capturar aspectos relacionados con la actividad y la situación dinámica. Una técnica muy común para capturar tales aspectos es la visión por computador, que, al contrario que el posicionamiento, permite la adquisición de un contexto más genérico por medio de una tecnología de sensores[31]. La visión por computador, no obstante, todavía requiere entornos más o menos ideales para una ejecución fiable y además causa más preocupaciones sobre la privacidad que el posicionamiento.

Nuevas técnicas basadas en la integración de diversos juegos de sensores han demostrado ser una interesante alternativa, tanto en dispositivos móviles como en entornos inteligentes[178]. Por ejemplo, un proyecto de investigación reciente ha hecho que un teléfono móvil reconozca situaciones tales como encontrarse en una reunión, ir conduciendo, etc., a partir de datos multisensoriales proporcionados por un pequeño y barato juego de sensores de audio, luz y movimiento. El reconocimiento de situaciones permite un comportamiento más inteligente del teléfono y mejora los servicios de telecomunicación, por ejemplo con el gestor de interrupciones.

El contexto se puede caracterizar como entrada implícitamente disponible para un sistema computacional[57], sin interferencia de un usuario humano. Esto lleva a muchos investigadores a sugerir que las personas están fuera del circuito en la Informática sensible al contexto. En la actualidad se proponen programas completos de investigación en inteligencia ambiental considerando actividades computacionales realizadas para beneficio de las personas, pero sin que se den apenas cuenta de ellas. Lo que para algunos es el definitivo escenario para el uso del contexto causa seria preocupación para otros desde la perspectiva del factor humano, ya que afecta a la transparencia del sistema, la inteligibilidad y la capacidad de control.

2.5.7.6. Monitorizar y asistencia a la salud.

Este es otro campo de muchas aplicaciones [68, 74, 194] ya que se realizan una serie de prototipos para ayudar a los cuidados médicos de las personas enfermas, en especial de los ancianos. Esto es fácil de imaginar, nos estamos refiriendo a un ambiente inteligente, una casa inteligente, que le presta ayuda al paciente o al anciano a llevar sus actividades diarias sin descuidar para nada su salud.

Para esto se lleva un estricto seguimiento de la persona en el ambiente y con ayuda de interfaces, etiquetas inteligentes, se monitorizan posibles caídas, calidad de los alimentos que consume, la cantidad de líquidos tomada, el correcto uso de las medicinas, etc.[64]. Es decir, el ambiente y las *cosas* supervisan al paciente de modo que pueda hacer sus tareas del día a día y que el ambiente le recuerde

e inspeccione como cuida su salud o se recupera de una enfermedad.

2.6. MDD para Sistemas Ubícuos

Aunque la aplicación de enfoques "*model-driven*" no ha sido ampliamente adoptada en el ámbito de *sistemas ubícuos*, algunos esfuerzos heterogéneos que siguen este paradigma de desarrollo pueden ser identificados. Las propuestas más importantes incluyen: CML[104], pervML[143], VRDK[124], entre otros [52][104]. Estas propuestas usan un lenguaje de modelado para modelar el *sistema ubícuo*, enfocado en la descripción de los datos sensibles al contexto (o el marco del sistema), y en proporcionar herramientas visuales para facilitar el desarrollo del sistema. Algunas propuestas (tales como CML y VRDK) se enfocan directamente en el dispositivo como el principal componente del sistema y la implementación de la funcionalidad usando este dispositivo.

La propuesta en este trabajo ofrece abstracciones de alto nivel para desacoplar y separar las dependencias de los dispositivos con la funcionalidad que proporciona el dispositivo. Por lo tanto, se basa en encapsular el dispositivo en un servicio independiente controlado por los agentes. De manera similar como en pervML y otros[52]. Uno de los objetivos prioritarios es desacoplar la funcionalidad del sistema de la implementación de la aplicación (del software), utilizando una Arquitectura Orientada a Servicio (un entorno OSGi).

El marco OSGi[17] (Open Service Gateway Initiative), es una tecnología estándar que define un entorno para la ejecución de los servicios y la gestión de su ciclo de vida. El OSGi se enfocó originalmente en redes de dispositivos del mercado, utilizando los servicios que proporcionan acceso a las diferentes redes de dispositivos existentes (EIB, UPnP, X10, etc.). Estas características son muy interesantes para el desarrollo de *Sistemas Ubícuos*. El OSGi propone un componente llamado *Bundle*, que está basado en la tecnología *Java* y proporciona un mecanismo para la implantación y ejecución de aplicaciones.

En resumen, nuestro enfoque permite a los agentes gestionar, interactuar y controlar los dispositivos físicos (los mundos físicos) y así crear un modelo versátil del sistema ubícuo, puesto que los agentes (y el desarrollador) interactúan con el entorno utilizando servicios independientes, los cuales pueden ser combinados y/o compuestos, y así sucesivamente.

[17] http://www.osgi.org/

2.7. Conclusiones

Las plataformas de MAS presentadas en este capítulo son muy diversas. Existen las orientadas a organizaciones y otras que no poseen componentes que modelen los conceptos de sociedades directamente. Existen plataformas MAS orientadas a la implementación que soportan variados modelos de agentes como BDI, reactivos, etc. (pero generalmente estas plataformas dejan al diseñador construir su modelo propio de agente) y otras que tienen un modelo único de agente a diseñar (generalmente proporcionan herramientas que ayudan a diseñar el agente). Normalmente su implantación se realiza en la plataforma propietaria de la metodología seleccionada, en el caso de que se disponga de ella.

Si analizamos esta variedad de metodológias y plataformas para MAS, podemos concluir que el diseñador MAS cuenta con diferentes herramientas para abordar problemas complejos desde la perspectiva de MAS. Sin embargo, actualmente existen aplicaciones donde los MAS pueden ser empleados en campos muy diversos, como el comercio electrónico, servicios Web, computación móvil y ubícua. Estas aplicaciones han conducido al desarrollo del paradigma conocido como computación orientada a la interacción, es decir, una computación basada en la interacción entre entidades, convirtiéndose la computación en una actividad inherentemente social.

Aunque los MAS poseen una interacción, basada generalmente en los procesos de comunicación, que se considera una actividad social, muchas plataformas y/o metodologías carecen de componentes explícitos que permitan modelar las sociedades directamente. En muchas plataformas MAS, la sociedad es un comportamiento emergente del proceso de comunicación. Por ello, para soportar de forma más adecuada la computación orientada a la interacción, utilizar las Organizaciones Virtuales es un enfoque muy prometedor. Una VO posee los componentes específicos para diseñar y definir diferentes tipos de arquitecturas de sociedades. Se puede interpretar que la VO como una capa normativa y reguladora sobre un MAS que nos permite crear y definir explícitamente una sociedad de agentes.

Como la VO puede ser interpretada como una capa normativa del MAS, esta capa especifica cómo es la cooperación y las alianzas entre los agentes, y es generalmente una definición abstracta que no modifica los modelos de agente. Por ello, algunas de las plataformas descritas en este capítulo pueden ser usadas como plataformas que soportan las VOs, y que serán usadas en el desarrollo de este trabajo.

Sin embargo, diseñar aplicaciones usando VOs no es una tarea fácil, debido a la gran cantidad de componentes, plataformas y metodologías involucradas en el diseño. Pero, con la aparición de la enfoque *model-driven*, es una perspectiva prometedora para abordar la diversidad de componentes de las VOs, ya que utiliza

una visión integradora y guiada a través del desarrollo dirigido por modelos.

El MDD es un recurso relativamente nuevo en el campo de la ingeniería de software. El objetivo de MDD es la construcción de modelos que pueden ser leídos por computadoras, que pueda ser entendido por herramientas automáticas para generar plantillas de código, modelos de prueba, y que pueda integrar el código en múltiples plataformas y tecnologías[23, 75, 180, 35].

El enfoque MDD usa y crea diferentes modelos a distintos niveles de abstracción para combinarlos y ajustarlos cuando se necesita implementar la aplicación (CIM, PIM, PSM). Un aspecto fundamental de MDD es el *proceso de transformación* por medio del cual un modelo se convierte o se traslada a otro modelo, que puede ser del mismo nivel de abstracción o a un nivel de abstracción diferente[76, 123].

Este enfoque puede ser utilizado en los MAS, por ello el diseño de MAS actualmente ha identificado un conjunto de modelos que especifican diferentes características de un MAS, como por ejemplo Roles, Comportamientos, Tareas, Interacciones, Protocolos, etc. Estos modelos pueden usarse en la metodologías de MAS sin enfocarse en detalles específicos de la plataforma de ejecución[185] (un modelo tipo PIM). De esta forma, usando las transformaciones, es posible traducir el modelo abstracto del MAS en un modelo de implementación para diferentes plataformas de ejecución. Por ello actualmente la aplicación de MDD en MAS tiene diferentes aproximaciones o enfoques de acuerdo a los objetivos que busque el trabajo, sin embargo, algunas tendencias se pueden observar cuando se comparan estos trabajos.

La primera, en trabajos tales como: PASSI[66], TROPOS[45], INGENIAS[91], Sage[122], MetaDIMA[112] y otros[43]. El enfoque MDD se propone con el fin de aglutinar (un "*wrapper*") o de envolver la complejidad asociada con el desarrollo del MAS. Este "wrapper" se hace para recoger las diferencias de las diversas metodologías para el diseño de MAS, en un meta-modelo específico y propietario (rara vez es un meta-modelo unificado). Algunos de estos trabajos pueden generar implementaciones que se ejecutan en plataformas específicas propuesta por cada trabajo.

La segunda tendencia se observa en trabajos tales como: TAO[183], FAML[34], Agent UML (AUML)[28], AML[56], y otros[187], donde se persigue el objetivo de crear un meta-modelo unificado para diseñar y modelar diferentes metodologías de MAS, pero sin tener que preocuparse (por el momento) de la generación de código, que pueda ejecutarse en una plataforma. El principal objetivo de estos trabajos es obtener un *marco unificado y genérico* para analizar y entender las distintas abstracciones, componentes y sus interrelaciones, con el fin de soportar el diseño de agentes de las diferentes metodologías de MAS.

La tercera tendencia, es observada en los trabajos PIM4AGENT[99] y CAFnE[113], en los cuales su principal motivación es crear un meta-modelo unificado (menos

genérico que los trabajos anteriores) que permita el diseño de agentes de algunas metodologías basadas en MAS, donde también se permite la generación de código y que estas implementaciones puedan ejecutarse en diferentes plataformas de agentes.

Un análisis de estos enfoques indica, que sólo unos pocos soportan el uso del concepto de la organización, por ejemplo FAML, PIM4AGENT y TAO, pero ninguno de ellos soporta el uso de organizaciones como otro marco distinto del marco MAS. En otros trabajos, cada uno propone su propio modelo, con un punto de vista propio y con componentes específicos, lo que crea mayor complejidad para los desarrolladores. Además, sólo unos pocos de ellos alcanzan la fase de implementación, y sólo se definen los modelos de alto nivel. Esto complica enormemente el trabajo de los desarrolladores cuando tratan de obtener el código ejecutable. Finalmente, podemos agregar que algunos de enfoques no soportan una interacción con su entorno físico o es limitada, situación que dificulta a los desarrolladores cuando es necesario diseñar un MAS en escenarios donde se reciben o se controlan variables o parámetros en el entorno real, como lo son los sistemas ubícuos.

Siguiendo la tendencia de los trabajos previos, este trabajo se propone usar el Desarrollo Dirigido por Modelos en el diseño de MAS orientados a las Organizaciones y con capacidades ubícuas. Así, lo primero que se propone es crear un conjunto de meta-modelos basados en VOs que permitan generar implementaciones (código) que pueda ser ejecutado en diferentes plataformas que soporten VOs, tales como THOMAS y Electronic Institutions. Posteriormente, proporcionar a los MAS diseñados de la capacidad de percibir e interactuar con su entorno físico, como un sistema ubícuo, usando las funcionalidades de los dispositivos (objetos) ubicados en el ambiente. Los meta-modelos propuestos serán unificados y tendrá un nivel de generalización suficiente para permitir que muchas de las metodologías MAS sean analizadas en esta propuesta. Sin embargo, los meta-modelos propuestos son lo suficientemente generales como para soportar a las metodologías orientadas a la organización, como se demostrará en en la sección 3.5 donde se describen las transformaciones necesarias.

Resumiendo, a lo largo de este capítulo hemos realizado una revisión de diversos temas que están involucrados en este trabajo. Empezamos con una introducción de los agentes, sistemas multi-agente y plataformas, Organizaciones virtuales, Computación Ubícua, luego el estudio se centró en el enfoque MDD y, finalmente, se describió como se aplica está metodología en la tecnología de agentes. En los siguientes capítulos se mostrará el meta-modelo de Organización Virtual Ubícua propuesto, sus principales conceptos y cómo se relacionan con los otros componentes de los MAS y los sistemas ubícuos. Sobre este meta-modelo de VO se construirá un modelo de transformación para obtener a partir de los modelos, el código basado en la tecnología de agentes sobre distintas plataformas

específicas. También en capítulos posteriores, se describió una de las plataformas donde se ejecutan las implementaciones. Estas implementaciones se dividen en dos niveles (según la plataforma): en el código propio de la VO y el código del agente. Para las plataformas de VO el código generado será para THOMAS y Electronic Institutions, mientras que las plataformas de agentes usadas serán ANDROMEDA (una plataforma desarrollada en este trabajo que nos permite ejecutar los agentes en el sistema operativo *Android* sobre teléfonos móviles y tabletas) y JADE (particularmente JADE-Leap) ya que muchos de los agentes que se plantean diseñar serán empotrados.

3

Desarrollo Dirigido por Modelos de Organizaciones Virtuales Ubícuas

Índice

3.1. Introdución

La definición básica de una Organización Virtual (VO) es bastante simple: son organizaciones e individuos que se unen entre si, de forma dinámica, con el fin de

cooperar y compartir recursos a través de las alianzas temporales[22, 136]. Varios modelos se plantean a distintos niveles de abstracción cuando se trata de describir la alianza o la unión entre los miembros y el intercambio de recursos. Sin embargo, en este trabajo se presenta a la VO como una descripción abstracta de alto nivel, (i) que reglamenta o norma las relaciones y alianzas entre los miembros; (ii) que posee un ciclo de vida (*formación*, *operación*, y *disolución*); y (iii) que define un modelo para el uso de los recursos de la misma. Nos centramos en una VO que puede ser formada y gestionada automáticamente por agentes inteligentes. Es decir, una organización virtual vista como una sociedad de agentes[136].

Para apoyar la automatización mencionada los agentes representan a organizaciones e individuos (unidades organizacionales y agentes, que pueden exhibir algunos *rasgos humanos*) proporcionando y solicitando recursos/servicios. Los agentes están diseñados para incorporar requerimientos organizacionales e individuales, usados para apoyar los procesos de toma de decisiones de forma automática.

La motivación de solucionar problemas con una VO es crear un marco de colaboración, que especifique patrones o normas para el consumo de productos o el uso de servicios de los clientes de la organización. Las razones por las que las organizaciones pueden proporcionar este tipo de servicios y es porque permite la alianza y cooperación de sus miembros. Esta fusión virtual permite crear soluciones de gran complejidad técnica y constantemente cambiantes a las necesidades de los clientes y del mercado[136]. Los MAS por si solos también pueden abordar problemas complejos, pero en muchos escenarios el uso de VO facilita la resolución de estos problemas. Los MAS no cuentan con los mecanismos de las organizaciones, a menos que se unan virtualmente para compartir recursos, capacidades, experiencia, y servicios comunes, no es posible satisfacer las necesidades que demandan de la organización. En los MAS no es sencillo compaginar los objetivos colectivos de la organización con los objetivos individuales de los miembros, característica que las organizaciones si permiten.

Para resumir, los beneficios de uso de una VO se pueden simplificar en los siguientes aspectos[136]: (i) Las VOs hacen posible satisfacer constantemente necesidades cambiantes de los clientes y del mercado de una manera competitiva. (ii) Se hace posible proporcionar servicios adaptados con precisión a una necesidad específica del cliente, además, las VOs aumentan el rango total de servicios que una empresa puede ofrecer a sus clientes. (iii) Una organización en particular puede "multiplicarse" tantas veces como pueda participar en varias VOs y/o iniciar la posibilidad de coexistencia con otras VOs como si fuese una sola organización.

A pesar de los beneficios que aporta el uso de VO, cuando se requiere utilizar este enfoque para resolver problemas en computación móvil y ubícua, el diseñador se encuentra con una limitada interacción de las VOs con su entorno, pocas

propuestas tienen la capacidad de acceder al entorno físico o real. Es importante contar con propuestas que soporten servicios basados en dispositivos físicos presentes en el entorno de una VO, que los agentes puedan sensorizar y actuar sobre el entorno. Así, con el propósito de crear aplicaciones ubícuas complejas se proponen las *Organizaciones Virtuales Ubícuas*, como una propuesta para abordar los desafíos de desarrollar sistemas ubícuos funcionales.

Pero uno de los principales problemas en el desarrollo de este tipo de aplicaciones, es que hay que tener una vasta experiencia en el área, si se pretende abordar un problema verdaderamente complejo. El desarrollo de sistemas ubícuos basado en agentes es una tarea compleja, con múltiples actores y diferentes entornos de hardware/software, es decir, esto se debe a la heterogeneidad de los sistemas de computación (lenguajes, sistemas operativos, etc.) y a la diversidad de dispositivos empotrados en el entorno (sensores, interfaces, controladores, etc.), lo cual obliga al desarrollador a conocer los detalles de cada tecnología presente en el sistema, y donde es difícil encontrar un punto de vista compacto de todos los componentes. Esto hace que el desarrollo, el diseño e implementación de este tipo de sistemas sea una tarea bastante complicada[182, 104].

Sin embargo, el enfoque que proporciona el MDD puede ser una buena solución para los problemas antes mencionados, ya que el desarrollador diseña utilizando conceptos y primitivas abstractas del dominio del problema, en lugar de conceptos que dependan de la tecnología. Los modelos pueden capturar los detalles de la tecnología y hacerlos casi independientes de la misma, y con ello soportar la heterogeneidad y la evolución de los dispositivos y componentes utilizados en el sistema.

El objetivo de aplicar MDD al diseño de organizaciones virtuales ubícuas es el de proveer al usuario de un modelo organizacional visual, unificado e intuitivo. De esta forma, el usuario puede usar las transformaciones automáticas que permiten una implementación flexible (incluyendo la implantación) sobre diferentes plataformas de agente con soporte para organizaciones, facilitando la interoperabilidad del sistema con la mínima intervención del usuario. La Figura 3.1 muestra un diagrama que ilustra este proceso. En el nivel CIM existen las diferentes metodologías: de agente y de organizaciones virtuales. Estas metodologías aportan los conceptos, relaciones y componentes fundamentales para crear un meta-modelo de *Organinzación Virtual* que tenga la propiedad de ser independiente de la plataforma de ejecución (PIM). El desarrollador diseña su MAS usando este meta-modelo, y posteriormente necesitará traducirlo a un modelo de MAS específico (PSM), con el objetivo de implementar su diseño. Para esto, usa las transformaciones verticales, específicamente un modelo de transformación de PIM a PSM. En la Figura 3.1 se observa (en el nivel PSM) que las plataformas específicas utilizadas son THOMAS y E-Institutions. Finalmente, el desarrollador necesita el código de implantación para ejecutar su MAS, y esto lo logra

aplicando un segunda transformación vertical de modelo a código, donde obtiene las plantillas de código (o los esqueletos del código, que usualmente debe completar con código que el usuario agrega manualmente) que luego debe compilar. En la Figura 3.1 se obtienen plantillas de código de THOMAS y E-Institutions.

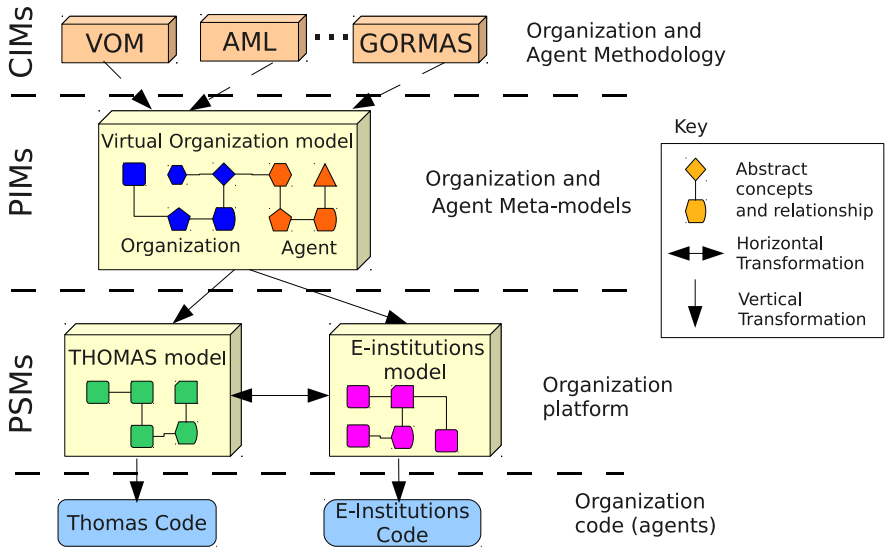

Figura 3.1: Relaciones entre los diferentes componentes MDD y las transformaciones automáticas

Este trabajo se enfoca en la capa del meta-modelo (nivel PIM), en la cual se definen los diferentes meta-modelos desarrollados para MAS abiertos (dominio de aplicación). Este conjunto de meta-modelos lo hemos denominado *Modelo de Organización Virtual Independiente de Plataforma* (πVOM, por sus siglas en inglés *Platform-Independent Virtual Organization Model*), y se presenta detalladamente en este capítulo.

La creación de este conjunto de meta-modelos se realizó por la detección de los conceptos comunes (análisis *bottom-up*) de las metodologías existentes de agentes y organizacionales (nivel CIM), complementado por una evaluación de los componentes necesarios en los MAS y VOs (análisis *top-down*). Después de esto, πVOM puede ser convertido o trasladado a un nuevo modelo que es orientado a la plataforma de implementación del MAS (nivel PSM). Esto es realizado a través de una transformación de modelo-a-modelo (PIM-a-PSM). Finalmente, la implantación del MAS se obtiene con una transformación modelo-a-texto, la cual corresponde a la generación de código por la filosofía de MDD.

Un reto fundamental cuando se define un meta-modelo independiente de plataforma para un MAS abierto, es seleccionar los conceptos o componentes que deben ser incluidos con el fin de modelar la organización. Esta no es una tarea

trivial, debido a que las metodologías existentes proporcionan un conjunto muy variado de abstracciones ajustadas a los diferentes dominios de aplicación. Cada metodología incluye sus propias abstracciones para el modelado conceptual y computacional, y por lo general no hay acuerdos acerca de un grupo común de abstracciones que puedan ser utilizadas a través de las diferentes metodologías. También, ciertos conceptos en un meta-modelo pueden ser contradictorios a los conceptos usados en otro meta-modelo de MAS.

Estos problemas son tratados en πVOM de dos maneras. Primero, debido a la capacidad de crecimiento de los meta-modelos, πVOM se puede adaptar a diferentes dominios ya que emplea conceptos comunes, que pueden extenderse para acomodar a nuevas abstracciones para nuevos dominios, de manera similar al enfoque planteado por TAO[183]. En segundo lugar, debido a la ambigüedad del lenguaje natural (donde diferentes términos representan el mismo concepto). En este sentido, la semántica de los conceptos usados en el meta-modelo pueden ser interpretados de manera muy amplia (de manera similar como lo plantea el enfoque FAML[34]). El desarrollador puede interpretar los conceptos de la manera más conveniente, los cuales están representados en lenguaje natural.

3.2. La integración de los conceptos en el meta-modelo

En las metodologías *orientadas a organizaciones*, una VO es considerada como una entidad social que está compuesta de un número específico de miembros que llevan a cabo diferentes tareas o funciones. Como se discutió en la sección 2.2, los principales aspectos considerados en una organización son la: Estructura, Funcionalidad, Dinámica, Normas y su Entorno. Por lo tanto, para modelar las características de estos componentes en nuestro enfoque, cinco conceptos claves son usados: *Unidad Organizacional*, *Servicio*, *Entorno*, *Norma*, y *Agente*[11, 22]. Estos conceptos hacen posible representar[67]:

- Cómo las entidades están agrupadas unas con otras, con el fin de definir la relación entre los elementos y su entorno.
- Qué funcionalidad ofrecen, incluyendo servicios para la entrada y salida dinámica de agentes en la organización.
- Qué restricciones existen con respecto a los comportamientos de las entidades del sistema.

La creación del meta-modelo ha sido un proceso iterativo. Usando un enfoque *bottom-up*, varias iteraciones se realizaron entre diferentes metodologías MAS

analizadas. Posteriormente, el subconjunto común de componentes identificado ha sido evaluado con una perspectiva *top-down*. Este trabajo identifica los conceptos que los desarrolladores con frecuencia utilizan en la metodologías orientadas a la organización. πVOM propone combinar varios enfoques de metodologías de modelado de organización, especialmente: AML[56], AGRE[83], MOISE+[107], INGENIAS[159], GORMAS[22] y OMNI[72]. De dicha combinación han surgido un conjunto de componentes que se han agrupados en varias dimensiones o grupos funcionales. Los cuales se describen brevemente a continuación:

- La *Dimensión Estructural* de πVOM toma en cuenta el concepto agente-grupo-rol empleado en AGRE; el grupo, los roles y la noción de enlace usado en MOISE+ y GORMAS; y también el concepto de unidad organizacional de AML y su uso relacionado con la teoría de organizaciones humanas. En AML, una *unidad organizacional* es vista de dos formas, como una entidad atómica global y como una asociación de entidades internas, las cuales se relacionan entre ellas de acuerdo a sus roles, funcionalidades, recursos y entorno. Por lo tanto, la dimensión estructural permite la especificación de un sistema a un alto nivel de abstracción por medio de los conceptos de rol y de unidad organizacional.

- La *Dimensión Funcional* es representada por medio de las tareas y objetivos globales que persiguen los agentes. Por ejemplo, en MOISE+, los objetivos globales son definidos y descompuestos en las misiones realizadas por los agentes. πVOM amplía las propuestas anteriores de la descripción funcional en tres formas:

  - Las funcionalidades globales son descritas como un *ComposedService* (servicios complejos) que están compuestos de algunos *SingleService* (servicios atómicos), por lo que una especificación de un servicio complejo describe cómo los comportamientos de los agentes son orquestados.
  - La funcionalidad es detallada de dos formas: servicios que las entidades realizan y servicios que las entidades necesitan.
  - La funcionalidad en πVOM es descrita empleando el estándar OWL-S[133, 137], el cual permite la descripción semántica de los servicios, mejorando su expresividad, por ejemplo, representando las precondiciones y efectos del servicio.

  Por lo tanto, nuestra propuesta se enfoca en expresar la funcionalidad de un sistema y sus componentes por medio de la descripción de los servicios

similar a GORMAS. De esta forma, los conceptos de la Computación Orientada a Servicios (SOC, de sus siglas en inglés Service Oriented Computing) tales como ontologías, modelos de procesos, coreografía, los facilitadores, los acuerdos de nivel de servicio y la calidad de las medidas de servicio, pueden ser aplicadas a los MAS.

- La *Dimensión Normativa* contiene un conjunto de mecanismos para asegurar el orden social y prevenir comportamientos auto-interesados. Esta propuesta hace uso del enfoque normativo propuesto en GORMAS, MOISE+ y OMNI. Estas propuestas definen las normas como una descripción de un comportamiento esperado. Sin embargo, ninguna desviación del comportamiento esperado es válida o permitida. En este sentido, asumen la existencia de un "middleware" que controla la interacción de todos los agentes. Esta propuesta no se basa en un ejecutor centralizado de las normas. Por ello, los agentes son libres de decidir respetar las normas. La dimensión normativa de πVOM define las sanciones y las recompensas como método persuasivo para el cumplimiento de las normas.

- La *Dimensión del Entorno*, se enfoca en la descripción de los elementos del entorno. Para su definición han sido considerados los siguientes trabajos: AGRE, AML, GORMAS e INGENIAS. La dimensión del entorno en πVOM describe los componentes del entorno de una manera estándar, integrando las principales abstracciones de las anteriores propuestas. El concepto *Recurso* ha sido adoptado de INGENIAS y de la metodología de GORMAS. Este concepto es similar a la abstracción *Body* del modelo de AGRE, que indica cómo los agentes realizan acciones sobre los recursos. Por otra parte, el concepto *Port* de AML y GORMAS también está integrado en πVOM, y representa una abstracción para acceder a los recursos del sistema y las funcionalidades de publicación. Por ello, la dimensión del entorno de πVOM permite que agentes heterogéneos puedan acceder a funcionalidades externas y a recursos.

3.3. Descripción del meta-modelo: πVOM

La intención es que πVOM proporcione un conjunto de conceptos genéricos y primitivas útiles para un lenguaje de modelado, mientras que no se compromete a proporcionar todos los detalles necesarios exigidos por cada plataforma específica orientada a agentes. Como se ha comentado, πVOM está estructurado en diferentes meta-modelos o vistas. Los diferentes meta-modelos utilizados en nuestro enfoque se describen en detalle a continuación.

3.3.1. Meta-modelo Estructural

Este meta-modelo (ver Figura 3.2) describe cuales son los elementos del sistema (*agentes* y *unidades organizacionales*) y cómo se relacionan. πVOM propone definir una *unidad organizacional* (OU, de sus siglas en inglés *Organizational Unit*) como una entidad social básica que representa el conjunto mínimo de agentes que llevan a cabo algunas actividades o tareas específicas y diferenciadas, siguiendo un patrón predefinido de cooperación y comunicación[11, 21, 92, 204]. Esta asociación también puede ser vista como una entidad única en las fases de análisis y diseño, ya que persigue objetivos, ofrece y solicita servicios y juega un *Rol* específico dentro de otras unidades. Una OU está formada por diferentes *entidades* (relación *has_member*) a lo largo de su ciclo de vida, que pueden ser tanto agentes individuales y/o otras unidades organizativas. Las *unidades organizacionales* presentan diferentes topologías y relaciones de comunicación en función de su entorno, el tipo de actividades que realizan y su propósito. Las tipologías básicas son[106, 22] :

1. *Jerarquía Simple*, en el que un agente supervisor tiene el control sobre todos los miembros.

2. *Equipo*, que son grupos de agentes que comparten un objetivo común, que colaboran y cooperan entre ellos.

3. *Plana*, en el no hay ningún agente que tenga control sobre los otros miembros. Cualquier otra estructura puede ser definida a partir de estas tres topologías básicas.

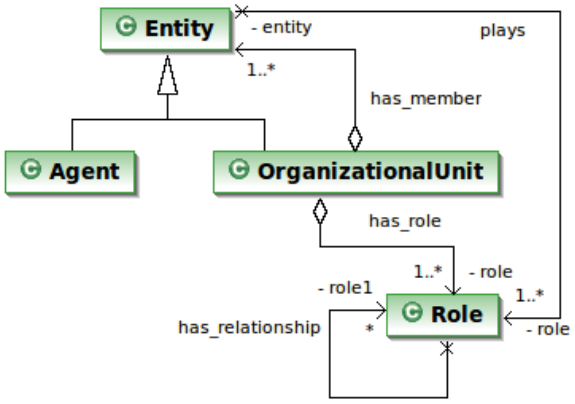

Figura 3.2: Conceptos usados en el meta-modelo Estructural

Una OU incluye un conjunto de roles que pueden ser adquiridos por sus miembros (*has_role*) y la clase de relaciones entre ellas (*has_relationship*). El concepto *Rol* (*Role*) es definido por tres atributos: *Visibilidad*, *Accesibilidad* y *Posición*. El concepto *Relación* (*Relationship*), basado en los trabajos [22, 183, 34], representa las relaciones sociales entre los roles. Esta relación conecta a los agentes que tienen derecho de conocerse unos a otros y para intercambiarse información pertinente. Está relación también puede implicar un proceso de *monitorización*, *supervisión* y *control* de la actividad del agente. La Tabla 3.1 resume los principales conceptos usados en el meta-modelo estructural.

Tabla 3.1: Principales conceptos usados en el meta-modelo estructural

Conceptos de πVOM	Descripción
Entity	Especificación de algo que tiene una existencia definida e individual dentro de la organización.
Agent	Una entidad racional y autónoma.
Organizational Unit	Especificación de un conjunto o grupo de entidades cooperativas (agentes y unidades organizativas) para alcanzar las metas de la organización.
Role	Especificación de un patrón de comportamiento que se espera de algunos miembros de una organización determinada.

3.3.2. Meta-modelo Funcional

Este meta-modelo (ver Figura 3.3) se enfoca en la integración de las tecnologías MAS y de Servicios. Los *Servicios* representan la funcionalidad que agentes y OUs ofrecen a otras *entidades*, independiente del agente en concreto que haga uso de ella. Los *Servicios* pueden ser atómicos (tareas simples) o formados por varias tareas. Esas tareas pueden ser ejecutadas por el agente que ofrece el servicio o pueden ser delegadas a otros agentes, por medio de la invocación, composición y orquestación del servicio.

Una *Entidad* (*Entity*) es descrita por un identificador y por una relación de pertenencia dentro de una unidad en la cual juega (*plays*) un rol específico (*Role*). Además, también es capaz de ofrecer algunas funcionalidades específicas a otras entidades. Su comportamiento es motivado por los objetivos (*Goals*) que persigue. Por otra parte, una unidad organizativa también pueden publicar sus requerimientos de servicios (relación *requires*), así los agentes externos pueden decidir si quieren participar internamente, y por ello proveer esos servicios. Cualquier servicio tiene uno o más roles que están a cargo de su prestación (*provides*)

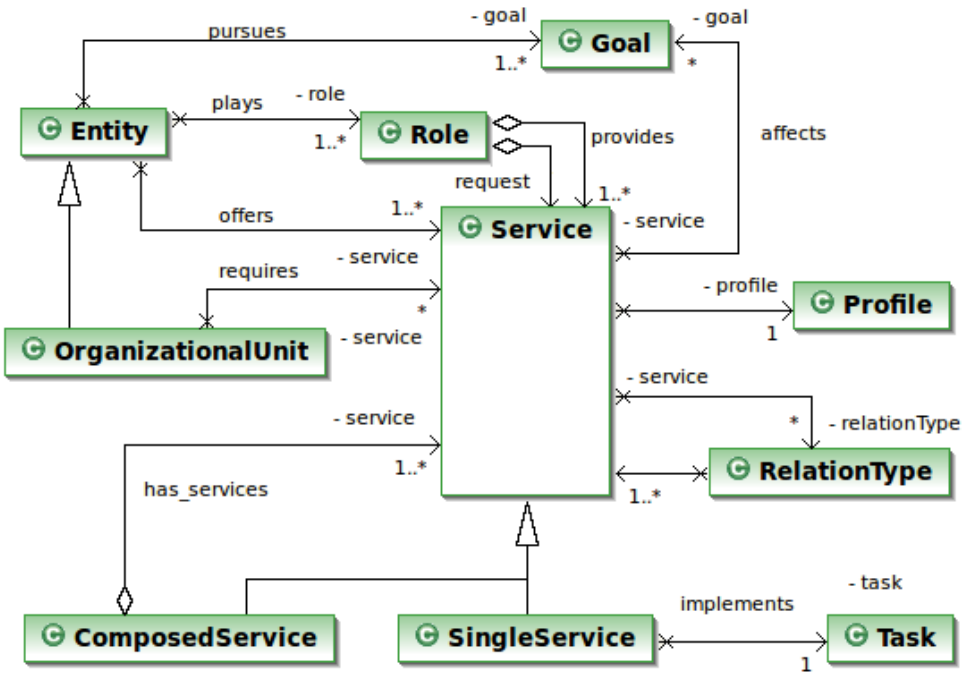

Figura 3.3: Conceptos usados en el meta-modelo Funcional

y otros que se encargan de consumirlo. Además, cualquier servicio puede tener influencia sobre los objetivos del sistema (relación *affects*). Un *Servicio* también puede estar compuesto de varios sub-servicios. Un flujo de trabajo puede ser definido usando el concepto *RelationType*. La Tabla 3.2 resume los principales conceptos utilizados en el meta-modelo funcional.

3.3.3. Meta-modelo Normativo

Este meta-modelo asume que la coordinación entre agentes se alcanza a través del uso de *normas sociales*. Las normas describen el comportamiento esperado de los miembros, es decir, qué acciones son permitidas, requeridas o necesarias y cuales deben evitarse. Las normas también incluyen sanciones que se aplicarán en el caso de producirse acciones no deseadas, y recompensas o reconocimientos que se aplicarán para las acciones llevadas a cabo según lo establecido por la norma (o regla). Las *Normas* (*Norms*) son usadas como mecanismo para limitar la autonomía de los agentes en sistemas grandes y para resolver problemas complejos de coordinación. Este meta-modelo (ver Figura 3.4) específica el conjunto de reglas y acciones definidas para controlar el comportamiento de los miembros de la organización, y específicamente, el de los *Roles* de la organización.

Tabla 3.2: Principales conceptos usados en el meta-modelo funcional

Conceptos de π VOM	Descripción
Entity	Especificación de algo que tiene una existencia definida e individual dentro de la organización.
Organizational Unit	Especificación de un conjunto o grupo de entidades cooperativas (agentes y unidades organizativas) para alcanzar las metas de la organización.
Role	Especificación de un patrón de comportamiento que se espera de algunos miembros de una organización determinada.
Service	Una simple actividad (o un complejo bloque de actividades) que representa una funcionalidad de un agente o una organización.
ComposedService	Un conjunto de sub-servicios que conforma un Servicio.
SingleService	Un servicio simple (único) que representa una funcionalidad.
Goal	Es la especificación de un estado que la organización y los agentes están tratando de alcanzar.
Task	Unidad fundamental que representa las acciones ejecutas por un agente.
Profile	Especificación de un servicio, incluyendo cualquier pre-condición y post-condición.
RelationType	Específica el tipo de relación (flujo) entre los servicios y sub-servicios.

Cada OU tiene un conjunto de normas (*norms*) que restringe el comportamiento de los miembros (*has_norm*). Una norma afecta directamente a un rol (*affects*), que está obligado, tiene prohibido o permitido llevar a cabo una acción específica. Las acciones válidas son solicitudes, registro o prestación de servicios. Las sanciones y recompensas son expresadas por medio de normas. Hay roles que son los responsables de controlar el cumplimiento de las normas (*is_follower*), mientras que hay roles defensores y promotores que son los responsables de llevar a cabo las sanciones y las recompensas, respectivamente. La Tabla 3.3 resume los principales conceptos usados en el meta-modelo normativo.

3.3.4. Meta-modelo de Agente

Un agente (*Agent*) es la entidad básica del MAS que está dentro de una o varias organizaciones, y utiliza una serie de protocolos de interacción. El meta-

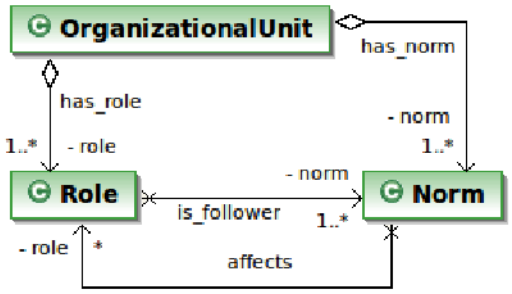

Figura 3.4: Conceptos usados en el meta-modelo Normativo

Tabla 3.3: Principales conceptos utilizados en el meta-modelo normativo

Conceptos de πVOM	Descripción
Organizational Unit	Especificación de un conjunto o grupo de entidades cooperativas (agentes y unidades organizativas) para alcanzar las metas de la organización.
Role	Especificación de un patrón de comportamiento que se espera de algunos miembros de una organización determinada.
Norm	Un conjunto de reglas que son usadas como mecanismo para limitar la autonomía de los miembros de la organización.

modelo de agente es un conjunto de componentes interrelacionados, donde cada uno cumple una función específica para la definición de agente y entre sus principales componentes tenemos: los Comportamientos (*Behaviours*), las Capacidades (*Capabilities*), y las Tareas (*Tasks*), mostrados en la Figura 3.5.

- Las Tareas (*Tasks*) representan el saber hacer (*know-how*) del agente y son los componentes donde las acciones o actividades son implementadas.
- Las Capacidades (*Capabilities*) representan las diferentes habilidades del agente y controlan donde se aplican las tareas. Las capacidades siguen un patrón de *evento-condición-acción*.
- Los Comportamientos (*Behaviours*) son la representación interna del concepto de rol, y agrupan las capacidades del agente.

La principal razón para dividir la forma de cómo resolver los problemas (*problem-solving*) es para proporcionar una abstracción que organice el conocimiento de una forma modular y gradual. El concepto Tarea (*Task*) es un componente que incorpora el saber-hacer necesario que le permite al agente el tratar

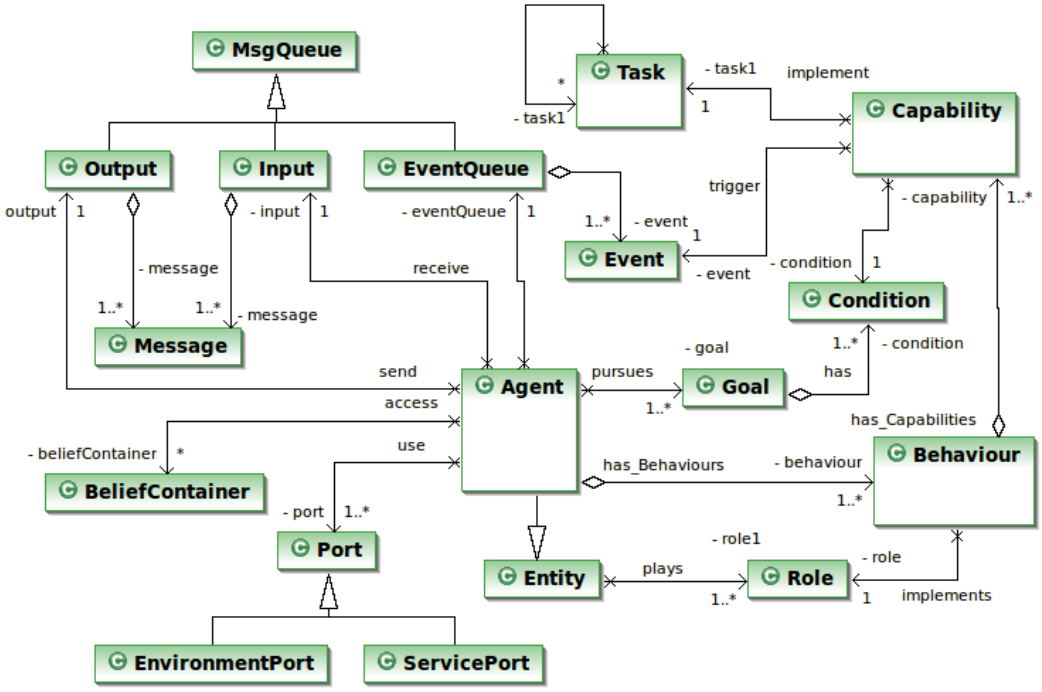

Figura 3.5: Conceptos usados en el meta-modelo del agente

de resolver un problema. Este concepto es encapsulado en una Capacidad (*Capability*), que es un componente orientado a eventos y permite expresar bajo qué circunstancias una tarea debe ser lanzada a ejecución.

Por otra parte, un conjunto de Capacidades pueden ser encapsuladas dentro de un Comportamiento (*Behaviour*), que modela la respuesta del agente a diferentes situaciones. Un estado del agente define una situación (que está representada por sus Creencias (*Beliefs*) y sus Objetivos (*Goals*) actuales) que activa un Comportamiento (*Behaviour*) o permite que esté siga siendo activado. La Tabla 3.4 resume los principales componentes y conceptos utilizados en el meta-modelo del agente.

El modelo propuesto recoge los principales componentes de los modelos de agentes de diferentes metodologías MAS, con el objetivo de abarcar los conceptos principales para su diseño. Como es un modelo de diseño se abstrae de los componentes de agente para su implementación. Además, como sigue el enfoque MDD sus componentes posteriormente deben ser transformados a una plataforma especifica de implementación para lograr su instanciación.

Tabla 3.4: Principales conceptos usados en el meta-modelo del agente

Conceptos de πVOM	Descripción
Agent	Una entidad racional y autónoma.
Behaviour	Encapsula un conjunto de Capacidades activadas en circunstancias específicas, representa el concepto abstracto del rol.
Capability	Representa un enfoque orientado a eventos para resolver un problema específico.
Task	Es el saber hacer relacionado con un problema específico.
Event	Se emplea para activar las Capacidades dentro de la agente. Indica la ocurrencia de algún cambio en el entorno o en el agente.
BeliefContainer	Una abstracción utilizada para representar el conocimiento del agente.
Goal	Una especificación de un estado que la organización y los agentes están tratando de alcanzar.
Condition	Una especificación de un conjunto de restricciones.
MsgQueue	Una especificación de un conjunto de diferentes mensajes (Entrada, Salida, Eventos).
Message	El mecanismo típico usado para la intercomunicación entre los agentes.

3.3.5. Meta-modelo de Entorno: un meta-modelo ubícuo

Antes de describir este meta-modelo, debemos recordar que el acceso al entorno físico de las diferentes propuestas de VOs y de MAS no es algo común, sino más bien algo limitado (como se describió en el capítulo 2). Por ello, que los agentes usen los dispositivos en un entorno real (artefactos electrónicos con capacidades de comunicación y sensorización), es uno de los objetivos de esta propuesta.

Nos proponemos crear este meta-modelo para permitir a los agentes (y a las entidades de la VOs) interactuar fácilmente con los diferentes dispositivos físicos en el entorno. Se les permite a los agentes a que puedan administrar, percibir y estar ubicados en un sistema ubícuo. Es por ello que este meta-modelo de Entorno es capaz de capturar los detalles y requisitos de un sistema ubícuo.

Este meta-modelo ha sido definido mediante la detección de conceptos comunes, en un ciclo iterativo que consiste en un análisis "*bottom-up*" de los elementos comunes en las metodologías de sistemas ubícuos existentes. Dichos elementos se han identificado e incorporado en el nivel del Modelo Independiente de Compu-

tación (CIM) (ver la Figura 3.6). Luego este modelo puede ser ajustado con los otros modelos MDD que especifican los conceptos más propios del MAS (y de la VOs), y combinar componentes como: Entidades, Roles, Comportamientos, Tareas con conceptos: Entornos, Artefactos o Dispositivos.

Así, la integración de los modelos pueden ser usados para describir la *Organización Virtual Ubícua* sin enfocarse en los detalles o requerimientos específicos de plataforma, como un Modelo Independiente de Plataforma (PIM). Después de esto, es posible transformar los modelos PIMs en Modelos Específicos de Plataforma (PSM). La Figura 3.6 muestra las relaciones propuestas entre los diferentes modelos MDD y sus transformaciones.

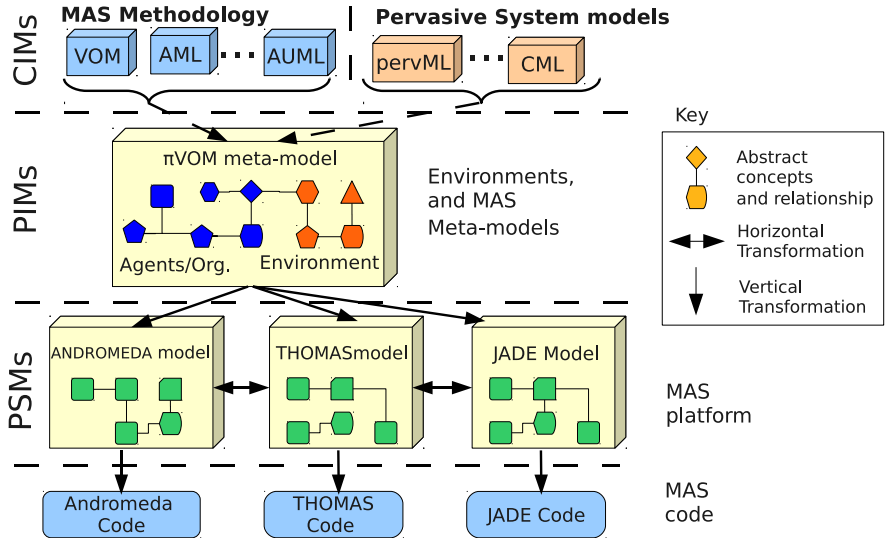

Figura 3.6: MDD para la Organización Virtual Ubícua

Al analizar la Figura 3.6 se puede observar, que para crear los meta-modelos de πVOM (que corresponde al modelo PIM) se utilizaron los principales componentes de las metodologías de organizaciones virtuales y de sistemas multi-agente (por ejemplo VOM, AUML, AML, etc., que corresponden a los modelos CIMs). Para que πVOM tuviese características ubícuas se tomaron los conceptos fundamentales de metodologías de sistemas ubícuos (por ejemplo pervML, CML, etc., que corresponden igualmente a los modelos CIMs). Una vez completados los meta-modelos de πVOM, se diseñan modelos de transformación que permiten traducir los modelos genéricos a modelos específicos de plataforma, como por ejemplo a las plataformas de VOs y/o MAS: Andromeda, THOMAS y JADE (en la Figura 3.6 esto corresponde al nivel PSMs). Finalmente la Figura 3.6 muestra también una última transformación para obtener las plantillas de código que permiten instanciar el diseño con πVOM a la plataforma que el desarrollador

desee.

El meta-modelo de entorno propuesto está enfocado en describir los componentes del ambiente (del entorno), representando las percepciones y acciones que pueden realizar los dispositivos. Por otra parte, define los permisos para acceder a ellos. En la Figura 3.7 se muestra el meta-modelo de Entorno propuesto. En dicho meta-modelo el concepto *Environment* representa el mundo físico donde los agentes están ubicados. El *Environment* puede ser percibido o censado usando la relación *Perceive*. El *Environment* es un modelo recursivo, el cual permite la creación de sub-mundos (ambientes o lugares) contenidos unos dentro de otros. Estos sub-mundos pueden estar conectados con otros mundos vecinos a través de la relación *neighborhood*.

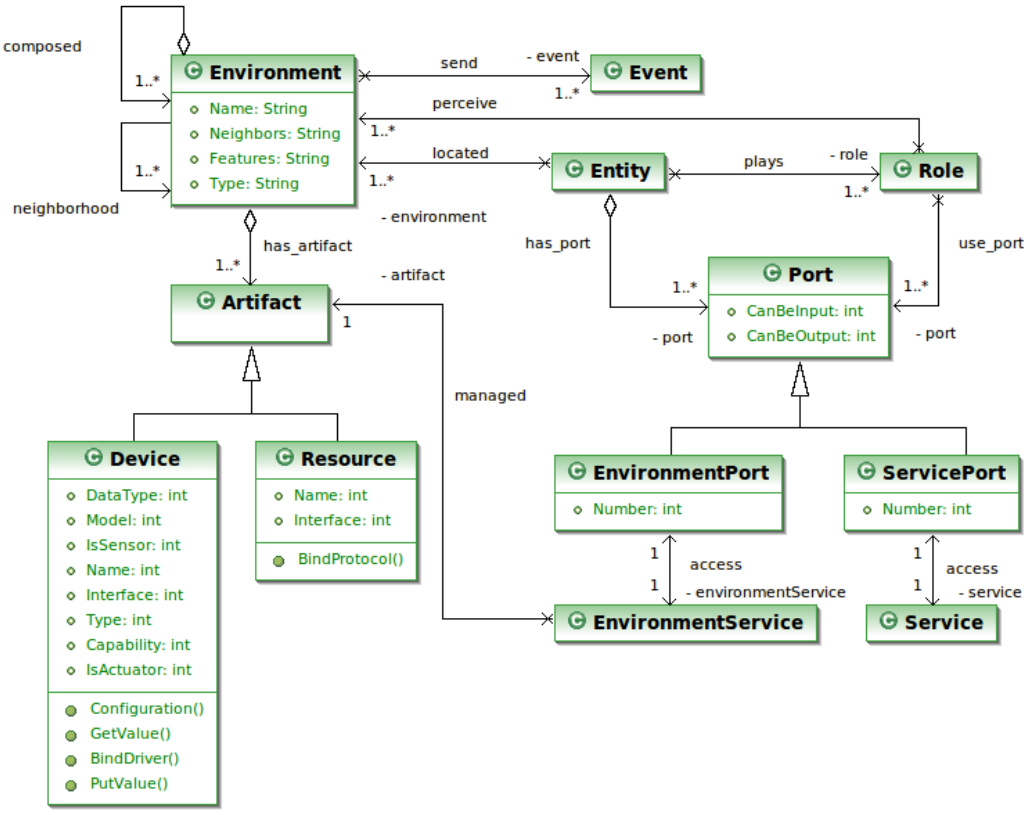

Figura 3.7: Conceptos usados y sus relaciones en el meta-modelo de Entorno

El *Environment* contiene *Resources* y *Devices*. Un *Resource* es un componente del entorno basado en software y su acceso es realizado a través de un protocolo estándar, que no requieren un ajuste de los controladores ("drivers") de dispositivo. Un *Device* representa un componente físico, en el cual su acceso se realiza directamente a través de una interfaz física (sensor o actuador). En este caso, se

requiere enlazar o vincular el "driver" de bajo nivel (*firmware*) a un componente software (de alto nivel).

El concepto *EnvironmentService* permite usar las funcionalidades de los dispositivos físicos a través de un servicio, desacoplando las abstracciones de bajo nivel. Los *Resources* y *Devices* se acceden a través de *EnvironmentPort*. Una *Entity* (Agente u Organización) es la encargada de gestionar los permisos de acceso a estos elementos utilizando la abstracción *Port*. Cada *Port* es controlado por una entidad y puede ser utilizado por uno o más roles (jugados por las entidades). Un *Port* representa un punto de interacción entre la *Entity* y el dispositivo físico, y sirve como una interfaz al mundo físico. Finalmente, todos estos conceptos básicos y relaciones permitirán a los agentes (y a los usuarios) crear nuevas representaciones de alto nivel de un *Sistema Pervasivo*, como los modelos sensibles al contexto[27], modelos de descubrimientos y composición de servicios[47], y modelos de adaptación[165]. La Tabla 3.5 resume los principales conceptos usados en el meta-modelo de Entorno.

Finalmente, se ilustra el uso de este meta-modelo, con un típico caso de estudio de un *Sistema Pervasivo*. El caso de estudio describe una pequeña oficina donde se ofrecen algunos servicios ubícuos. La oficina está estructura en tres ambientes: el mostrador de recepción, la oficina central y sala de reuniones. Cada ambiente proporciona un servicio de iluminación que se activa cuando se detecta en el espacio la presencia de un usuario. La oficina central y la sala de reuniones poseen servicios para contenidos multimedia (audio, vídeo y presentaciones). Cuando los empleados hacen una presentación o utilizan dispositivos multimedia en la sala de reuniones, la intensidad de la luz se reduce y las persianas se cierran automáticamente para mejorar la visibilidad.

La oficina ubícua también posee otros tipos de servicios, tales como: el de seguridad y de grabación. Estos servicios están disponibles en horas donde la oficina normalmente está cerrada y no hay presencia de usuarios. Se activan cuando se detecta en el ambiente la presencia de un intruso (un usuario fuera del horario permitido), en este caso, se genera una señal o un evento de que la seguridad ha sido violada y por ello, las cámaras de los dispositivos multimedia y las luces se activan automáticamente para grabar las actividades del intruso.

Con el fin de modelar la oficina ubícua, el desarrollador debe específicar los diferentes componentes que modelan a los distintos dispositivos, recursos, entornos, servicios y agentes del *Sistema Pervasivo*. La Figura 3.8 muestra el modelo parcial (enfocado en la sala de reuniones) de la oficina usando el meta-modelo de Entorno de πVOM. La Figura 3.8 muestra que la oficina ubícua utiliza diferentes dispositivos, tales como: bombillas inteligentes (utilizando el protocolo X10), bombillas de luz graduales y persianas automáticas; estos se utilizan para controlar la iluminación del ambiente. Además, la oficina ubícua utiliza sensores infrarrojos que son los detectores de movimiento y cámaras. Además, el desa-

Tabla 3.5: Principales conceptos usados en el meta-modelo de Entorno

πVOM conceptos	Descripción
Entity	Especificación de algo que tiene una existencia individual en el MAS.
Role	Especificación de un patrón de comportamiento esperado por algunas entidades del MAS.
Service	Una actividad sencilla (o un bloque complejo de actividades) que representa una funcionalidad de un agente/dispositivo/recurso.
Environment	Mundo físico, lugar donde los agentes están localizados.
Resource	Especificación de un artefacto de software, que tiene una representación razonable en el entorno, que puede ser percibido y compartido usando un protocolo de datos.
Device	Especificación de un artefacto de hardware, que podemos percibir y actuar a través de interfaces de bajo nivel (usando firmware propietario).
Port	Esta abstracción es una interfaz a los servicios, que permite la entrada/salida de datos.
EnvironmentPort	Punto de acceso para interactuar con el entorno (con el mundo físico).
ServicePort	Punto de acceso para usar un servicio basado en agentes.
EnvironmentService	Funcionalidad de alto nivel, que desacopla el protocolo o firmware de los artefactos del entorno (ubicados en el mundo físico).

rrollador puede crear nuevos servicios como el de Seguridad y Grabación, utilizando los servicios básicos (*EnvironmentService*) ofrecidos por cada dispositivo. El meta-modelo de entorno propuesto permite de una manera fácil y flexible el modelado de todos los entornos, dispositivos físicos, entidades (como el software) que conforman el ámbito de la oficina.

3.3.6. Extendiendo el modelo de eventos y tareas

Como esta propuesta se enfoca para aplicaciones de sistemas ubícuos, donde nos encontramos escenarios con entornos dinámicos. Estos sistemas tienen que hacer frente a un entorno que cambia en términos de los recursos disponibles y/o las normas de comportamiento individuales y colectivas. El entorno incluye

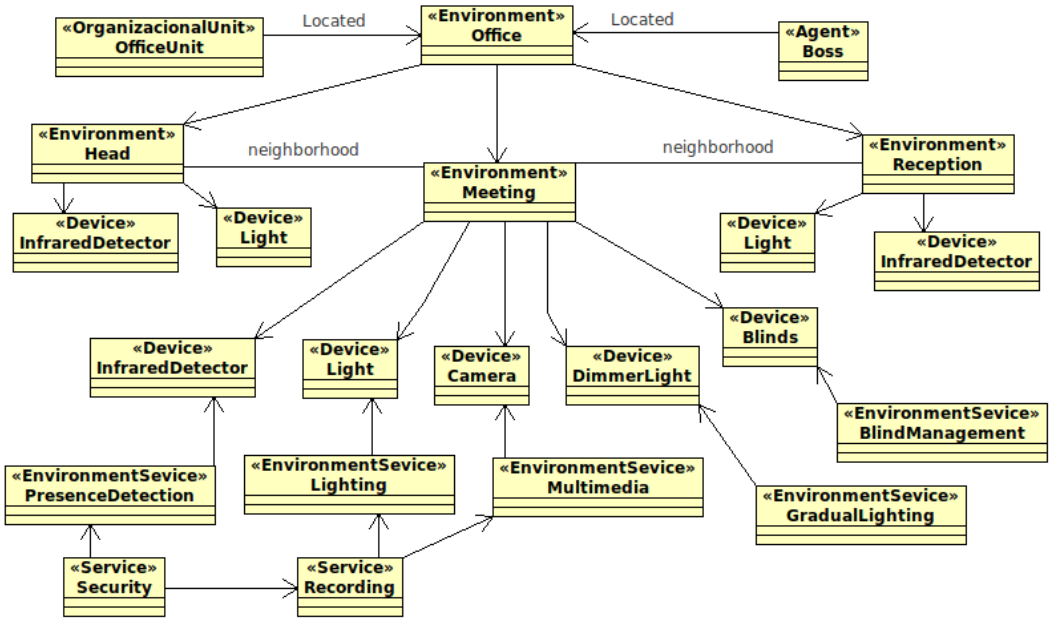

Figura 3.8: Vista parcial de la oficina usando el meta-modelo de Entorno

múltiples actores y objetos que pueden entrar o salir con frecuencia de la escena de forma inesperada. La naturaleza dinámica del entorno hace que las soluciones sean apropiadas en un momento dado e inadecuadas en otro instante en el tiempo.

Para soportar la dinámica de estos entornos proponemos dos abstracciones que extienden el modelo inicial de eventos y tareas de nuestro modelo de agente[12]. La primera abstracción permite al agente saber cómo procesar los estímulos repetitivos provenientes del entorno (como manejar eventos que producen cambios en el ambiente). La taxonomía permite que el agente identifique todos los eventos, y el orden en que serán procesados. La segunda abstracción permite al agente decidir cómo poner en marcha las acciones o tareas en respuesta a los cambios en el entorno (un modelo de evento-condición-acción). Con esta propuesta queremos proporcionar al desarrollador de las abstracciones necesarias con el fin de tener un agente especializado con capacidad de respuesta en entornos dinámicos.

3.3.6.1. Taxonomía de las Tareas

Como se describió anteriormente, un componente fundamental del modelo del agente es la tarea (*Task*). Las Tareas son los componentes que contienen el código asociado a la capacidad (*Capability*) del agente. Una tarea en ejecución pertenece a una sola capacidad, y puede ser vista como la respuesta del agente a un problema. Sin embargo, es el diseñador quien debe determinar si el problema

debe ser resuelto sólamente una vez o debe ser resuelto tantas veces como se produzca. De acuerdo con a la forma en que las diferentes instancias pueden ser activadas, las tareas se pueden clasificar en los siguientes tipos:

1. **Múltiples:** diferentes instancias de la tarea pueden estar activas al mismo tiempo. Por ejemplo, si una tarea tiene la respuesta a un mensaje específico por medio de un ACK, el diseñador puede decidir que estas respuestas son hechas en paralelo para los diferentes mensajes.

2. **Exclusiva:** Existe sólo una instancia de la tarea en un instante del tiempo. Este tipo de tarea puede ser dividido en dos sub-clases de acuerdo a la forma en que las nuevas activaciones de las instancias son tratadas:

   a) **No interrumpible:** La primera instancia se continúa ejecutándose hasta que finalice, retrasando así la posible ejecución de nuevas instancias de la misma tarea.

   b) **Interrumpible:** La nueva instancia elimina la antigua. Por ejemplo, si la tarea es para calcular una solución a un problema y la generación de una nueva instancia indica que los datos que están siendo utilizados en el cálculo de instancia antigua son obsoletos, entonces el cálculo que se está haciendo ya no es útil.

3.3.6.2. Taxonomía de los eventos

Un evento (*Event*) es cualquier notificación recibida por el agente informándole de que algo que puede ser interesante ha ocurrido en el entorno o en el interior del agente. Esto puede causar la activación de una nueva capacidad *Capability*. De forma similar a las tareas, una posible clasificación de eventos aparece en el proceso de administración de nuevas instancias de un mismo evento:

1. **Múltiples:** Puede haber diferentes instancias del mismo evento en la cola, y todos ellos tienen que ser gestionados.

2. **Exclusiva:** Existe sólo una instancia del evento a la espera de ser atendido. Dependiendo de la forma en que se administrarán las nuevas instancias de los eventos, los eventos de este tipo se pueden clasificar en dos sub-tipos:

   a) **No interrumpible:** Si llega a la cola una nueva instancia del evento está se elimina, sólo permanece la primera instancia.

   b) **Interrumpible:** La nueva instancia del evento elimina a la anterior y sólamente permanece la última instancia del evento recibida.

Esta taxonomía permite que el agente pueda gestionar mejor los cambios del entorno y esto se logra a través de la extensión de las abstracciones propuestas: de tarea y evento del agente ANDROMEDA, los cuales se resumen en la Figura 3.9.

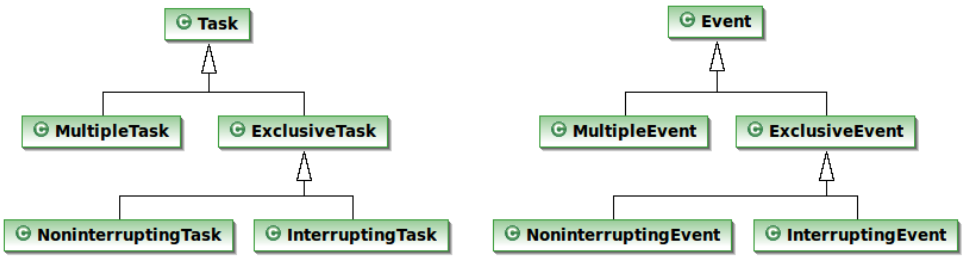

Figura 3.9: Extensión del modelo de eventos y tareas del agente

3.3.6.3. Taxonomía de las Capacidades

A partir de las taxonomías propuestas podemos analizar en detalle los diferentes tipos de capacidades (como resultado de la combinación de los diferentes tipos de eventos y tareas) que están disponibles para el desarrollador para resolver diferentes problemas. Sin embargo, antes de describir la taxonomía de la capacidad, vamos a utilizar un ejemplo para ilustrar esta taxonomía.

El ejemplo consta de un sistema multi-agente que gestiona y controla los procesos del sistema de transporte inteligente (*Smart Transport System*, que será descrito con más detalle la sección 6.3). El sistema de transporte inteligente es una aplicación que facilita la interconexión entre los pasajeros (ciudadanos, turistas), paradas de autobuses y vehículos de transporte (autobús, metro, tren, tranvía); delimita los servicios que cada uno puede solicitar u ofrecer. El sistema controla que servicios deben ser proporcionados por cada agente. El sistema propuesto proporciona servicios de datos inalámbricos, que permiten a los dispositivos móviles (por ejemplo: PDA, teléfonos móviles) comunicarse con los diferentes elementos del sistema que proporcionan servicios.

El agente inteligente instalado en el dispositivo embebido permite a los proveedores de servicio a descubrirse mutuamente. Esta interacción se ilustra (desde el punto de vista físico) en la Figura 3.10. El sistema puede proporcionar diferentes tipos de servicios para facilitar y mejorar la calidad del sistema de transporte público. El sistema de transporte Inteligente ofrece servicios móviles (para los pasajeros y vehículos), por ejemplo:

- Los pasajeros pueden recibir y solicitar información de las posibles rutas turísticas, servicios de noticias personalizadas, Tiempo estimado de llegada

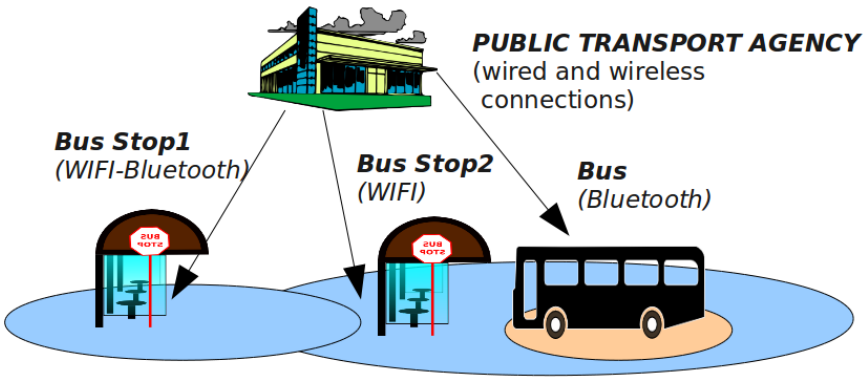

Figura 3.10: Resumen de la estructura del sistema de transporte inteligente

(ETA, de sus siglas en inglés Estimate Time Arrival), rutas más cortas hacia el destino, etc.

- Los servicios básicos del Agente de la parada de autobús podrían ser: (i) mostrar información en la pantalla; (ii) buscar una ruta de; (iii) el tiempo estimado de llegada ETA; y (iv) puntos de interés (POI, por sus siglas en inglés Point Of Interest). La información en la pantalla de la parada se ajusta en función de los perfiles de los pasajeros. Los pasajeros también pueden descargar sus noticias favoritas al terminal móvil (en función de su perfil de usuario).

Para analizar las diferentes características que posee nuestra propuesta, suponemos que contamos con un agente A_i que tiene varias tareas, capacidades y comportamientos (*Tasks*, *Capabilities* y *Behaviours*). Según esto, se puede definir lo siguiente:

1. Sea $C = \{C_1, C_2, \ldots, C_I\}$ el conjunto de todas las capacidades del agente, tal que su i-ésima capacidad es $C_i \in C | i = \{1, \ldots, I\} \wedge I \in \mathbb{N}$.

2. Sea E el conjunto de todos los eventos que puede manejar el agente. Estos eventos están agrupados en colas $E = \{E_1, E_2, \ldots, E_K\}$, donde E_K es la cola de eventos de la capacidad C_k, tal que $E_K \in E | K = \{1, \ldots, I\} \wedge I \in \mathbb{N}$.

3. Sea $E_K[t] = \{e_k[t_0], \ldots, e_k[t_n]\}$ la cola de eventos en el instante de tiempo t, que almacena una serie de eventos $e_k[t_i]$ producidos en el intervalo de tiempo $[t_0, t_n]$, tal que $e_k[t_i] \in E_K | (0 \leq i \leq n) \wedge (t_n \leq t_i) \wedge (t \geq t_n > t_0 > 0)$

4. Sea $T = \{T_1, T_2, \ldots, T_J\}$ el conjunto de todas las tareas del agente, tal que $T_j \in T | j = \{1, \ldots, I\} \wedge I \in \mathbb{N}$.

5. Sea $T[t]$ el conjunto de las tareas activas del agente en el instante t (con $T[t] \in T$).

6. Sea $T_k[t]$ la instancia de la k-ésima tarea que es lanzada en el instante t por la capacidad C_k en respuesta al evento $e_k[t]$ de la cola $E_K[t]$ (con $T_k[t] \in T[t] \in T$).

Finalmente, como se describió anteriormente, nuestro modelo de agente sigue un proceso *evento-condición-acción*, que es administrado por la capacidad del agente. Sin embargo, este proceso evento-condición-acción puede ser interpretado como una relación funcional, es decir, una capacidad puede ser vista como una función $y = C_k(x)$ (una función que activa o desactiva tareas). Una función que toma un evento $e_k[t]$ como entrada (generalmente el argumento x), y produce una tarea $T_k[t]$ (argumento y) como salida. De esta forma, una tarea activa es forzada a ser iniciada como $T_k[t] = C_k(e_k[t])$, si la condición de disparo es correcta. Resumiendo, al interpretar la capacidad como una función, ésta puede ser descrita como indica la ecuación 3.1.

$$T_k[t] = C_k(e_k[t]) = \begin{cases} \emptyset & \text{if } event\ condition = false \\ T_k & \text{if } event\ condition = true \end{cases} \tag{3.1}$$

Con estas definiciones, es posible analizar las diferentes combinaciones de las abstracciones propuestas que permiten al desarrollador resolver diferentes problemas en entornos dinámicos. La implementación de esta propuesta asume que en el ciclo de vida del agente, se reciben un conjunto de eventos con una frecuencia f_e, los cuales son almacenados en una cola. Estos eventos son procesados por el agente con una frecuencia o a una tasa dada por el planificador (f_s) y una tarea posteriormente es lanzada en respuesta a este evento (la cual tiene una duración τ). Para apreciar la utilidad de esta propuesta se supone que, $f_e \gg f_s \gg 1/\tau$, es decir, la velocidad con la cual los eventos llegan es muy alta. Teniendo esto en cuenta, la descripción anterior de la taxonomía de eventos y tareas genera nueve posibles tipos de capacidades, las cuales describimos a continuación.

3.3.6.4. Capacidad-mm: eventos y tareas múltiples

En este caso, hay una cola de eventos en el instante t, tal que $E_K[t] = \{e_k[t_0], \ldots, e_k[t_n]\}$ (con $t_n > t_0$), y se procesa el primer evento en la cola (que sigue pendiente) $e_k[t] = first(E_K[t]) = e_k[t_0]$, el cual es verificado por la capacidad C_k para activar la tarea $T_k[t] = C_k(e_k[t_0])$ para su lanzamiento. Las colas de eventos y tareas (en el agente) se pueden describir como indican las ecuaciones en 3.2.

$$\begin{aligned} E_K[t+1] &= E_K[t] - first(E_K[t]) \\ T[t+1] &= T[t] \cup \{T_k[t]\} \end{aligned} \tag{3.2}$$

Este escenario se supone que el agente reacciona a todos los estímulos del entorno y, por lo tanto, tiene todos los eventos en la memoria (en cola). En respuesta a esta monitorización, el agente reacciona lanzando tantas tareas como eventos procesados. Un ejemplo de este escenario implica que el agente tiene un control completo del entorno y responde a todas las dinámicas o cambios. En nuestro ejemplo, el Sistema de Transporte Inteligente, esta capacidad puede aplicarse para modelar el servicio del estado del tráfico (de la Agencia Central). Cada unidad de transporte puede solicitar el estado del tráfico y la información debe ser enviada a cada unidad de transporte.

3.3.6.5. Capacidad-im: eventos interrumpibles y tareas múltiples.

En este caso, se asume que hay una cola de eventos $E_K[t] = \{e_k[t_n]\}$ que almacena sólamente un evento significativo (el último evento). Sin embargo, también se puede interpretar que existe una cola de eventos real en el instante t, tal que $E_K[t] = \{e_k[t_0], \ldots, e_k[t_n]\}$ (con $t_n > t_0$), y como el modelo de eventos es interrumpible, sólamente necesitamos procesar el último evento $e_k[t] = last(E_K[t]) = e_k[t_n]$, y luego procedemos a vaciar la cola de los eventos que permanecen o están almacenados $E_K[t] = \emptyset$. Por lo tanto, una vez que el evento correspondiente es recibido, y es verificado por la capacidad C_k se activa la tarea $T_k[t] = C_k(e_k[t_n])$ para su lanzamiento. Las colas de eventos y tareas pueden describirse como indican las ecuaciones 3.3.

$$\begin{aligned} E_K[t+1] &= E_K[t] - last(E_K[t]) \Rightarrow E_K[t+1] = \emptyset \\ T[t+1] &= T[t] \cup \{T_k[t]\} \end{aligned} \tag{3.3}$$

En este escenario, el agente tiene una buena reactividad, y lanza nuevas tareas debido a los eventos relevantes. Sin embargo, tiene la capacidad de supervisión limitada (sólo el último evento es considerado como relevante). En nuestro ejemplo, el Sistema de Transporte Inteligente, esta capacidad permite modelar el servicio de noticias de última hora (de la Agencia Central). Este servicio envía las noticias más relevantes del sistema de transporte (y de la ciudad) al usuario y desecha las anteriores por obsoletas.

3.3.6.6. Capacidad-nm: eventos no interrumpibles y tareas múltiples.

En este caso, se asume que hay una cola de eventos $E_K[t] = \{e_k[t_0]\}$ que almacena sólamente un evento significativo (el primer evento). Sin embargo, también se puede interpretar que existe una cola de eventos real en el instante t, tal que $E_K[t] = \{e_k[t_0], \ldots, e_k[t_n]\}$ (con $t_n > t_0$), y como el modelo de eventos es no interrumpible, sólamente necesitamos procesar el primer evento $e_k[t] = first(E_K[t]) = e_k[t_0]$, y luego procedemos a vaciar la cola de los eventos

que permanecen o que están almacenados $E_K[t] = \emptyset$. Por lo tanto, una vez que el evento correspondiente es recibido, y es verificado por la capacidad C_k se activa la tarea $T_k[t] = C_k(e_k[t_0])$ para su lanzamiento. Las colas de eventos y tareas pueden describirse como indican las ecuaciones 3.4.

$$\begin{aligned} E_K[t+1] &= E_K[t] - first(E_K[t]) \Rightarrow E_K[t+1] = \emptyset \\ T[t+1] &= T[t] \cup \{T_k[t]\} \end{aligned} \tag{3.4}$$

En este escenario, el agente tiene una buena reactividad y lanza nuevas tareas en respuesta a eventos relevantes. Sin embargo, tiene la capacidad de supervisión limitada (sólo el primer evento se considera como relevante). En nuestro ejemplo, el Sistema de Transporte Inteligente, esta capacidad permite modelar el servicio de Comprobación en la parada de autobús o de la unidad de transporte. Este servicio comprueba la presencia del usuario en la unidad de transporte o una parada de autobús y sólo procesa la primera solicitud de todos los usuarios en el dominio, y el resto los desecha ya que con que se dé una sóla vez es suficiente.

3.3.6.7. Capacidad-mi: eventos múltiples y tareas interrumpibles.

En este caso, hay una cola de eventos en el instante t, tal que $E_K[t] = \{e_k[t_0], \ldots, e_k[t_n]\}$ (con $t_n > t_0$), y procesamos el primer evento de la cola (que está pendiente) $e_k[t] = first(E_K[t]) = e_k[t_0]$, el cual es verificado por la capacidad C_k para activar la tarea $T_k[t] = C_k(e_k[t_0])$ para su lanzamiento. Las colas de eventos y tareas pueden describirse como indican las ecuaciones 3.5.

$$\begin{aligned} E_K[t+1] &= E_K[t] - first(E_K[t]) \\ T[t+1] &= \begin{cases} (T[t] - \{T_k\}) \cup \{T_k[t]\} & \text{if } T[t] \cap \{T_k[t]\} \neq \emptyset \\ T[t] \cup \{T_k[t]\} & \text{otherwise} \end{cases} \end{aligned} \tag{3.5}$$

Este escenario supone que el agente supervisa todos los eventos, pero en respuesta mantiene una sóla tarea ejecutándose (de un tipo específico). Este escenario es muy reactivo, pero es costoso, ya que el agente debe detener de manera segura la tarea cuando se interrumpa. En nuestro ejemplo, el Sistema de Transporte Inteligente, esta capacidad permite modelar el servicio de temperatura en la unidad de autobús. El servicio ajusta la temperatura del acondicionador de aire en cada parada, buscando la temperatura media de los perfiles de cada usuario.

3.3.6.8. Capacidad-ii: eventos y tareas interrumpibles.

En este caso, se asume que hay una cola de eventos $E_K[t] = \{e_k[t_n]\}$ que almacena sólamente un evento significativo (el último evento). Sin embargo,

también se puede interpretar que existe una cola de eventos real en el instante t, tal que $E_K[t] = \{e_k[t_0], \ldots, e_k[t_n]\}$ (con $t_n > t_0$), y como el modelo de eventos es interrumpible, sólamente necesitamos procesar el último evento $e_k[t] = last(E_K[t]) = e_k[t_n]$, y luego procedemos a vaciar la cola de los eventos que permanecen o que están almacenados $E_K[t] = \emptyset$. Por lo tanto, una vez que el evento correspondiente es recibido, y es verificado por la capacidad C_k se activa la tarea $T_k[t] = C_k(e_k[t_n])$ para su lanzamiento. Las colas de eventos y tareas pueden describirse como indican las ecuaciones 3.6.

$$\begin{aligned} E_K[t+1] &= E_K[t] - last(E_K[t]) \Rightarrow E_K[t+1] = \emptyset \\ T[t+1] &= \begin{cases} (T[t] - \{T_k\}) \cup \{T_k[t]\} & \text{if } T[t] \cap \{T_k[t]\} \neq \emptyset \\ T[t] \cup \{T_k[t]\} & \text{otherwise} \end{cases} \end{aligned} \tag{3.6}$$

En este escenario, el agente tiene una buena reactividad, a pesar de que tiene la capacidad de memoria limitada (de eventos), sólo almacena el último evento. Ante la llegada de un nuevo evento, se detiene la tarea para este tipo de evento, y lanza una nueva tarea. Una nueva tarea que mejor se adapta o ajusta al nuevo evento recibido. En nuestro ejemplo, el Sistema de Transporte Inteligente, esta capacidad permite modelar el servicio de los POI's (puntos de interés). Este es un servicio que informa al usuario de los puntos de interés (por ejemplo, sitios turísticos, restaurantes) cerca de cada parada o estación. Este servicio describe en el dispositivo móvil de la dirección a seguir (*path*) para llegar al punto de interés utilizando los últimos eventos enviados por la parada de autobús.

3.3.6.9. Capacidad-ni: eventos no interrumpibles y tareas interrumpibles.

En este caso, se asume que hay una cola de eventos $E_K[t] = \{e_k[t_0]\}$ que almacena sólamente un evento significativo (el primer evento). Sin embargo, también se puede interpretar que existe una cola de eventos real en el instante t, tal que $E_K[t] = \{e_k[t_0], \ldots, e_k[t_n]\}$ (con $t_n > t_0$), y como el modelo de eventos es no interrumpible, sólamente necesitamos procesar el primer evento $e_k[t] = first(E_K[t]) = e_k[t_0]$, y luego procedemos a vaciar la cola de los eventos que permanecen o que están almacenados $E_K[t] = \emptyset$. Por lo tanto, una vez que el evento correspondiente es recibido, y es verificado por la capacidad C_k se activa la tarea $T_k[t] = C_k(e_k[t_0])$ para su lanzamiento. Las colas de eventos y tareas pueden describirse como indican las ecuaciones 3.7.

$$\begin{aligned} E_K[t+1] &= E_K[t] - first(E_K[t]) \Rightarrow E_K[t+1] = \emptyset \\ T[t+1] &= \begin{cases} (T[t] - \{T_k\}) \cup \{T_k[t]\} & \text{if } T[t] \cap \{T_k[t]\} \neq \emptyset \\ T[t] \cup \{T_k[t]\} & \text{otherwise} \end{cases} \end{aligned} \tag{3.7}$$

En este escenario, el agente tiene una buena reactividad, a pesar de que tiene la capacidad de memoria limitada (de eventos), sólo almacena un evento (el primero que es el relevante). Esta capacidad permite detener la tarea anterior (que ya no responde al evento), y lanza una nueva tarea.

3.3.6.10. Capacidad-mn: eventos múltiples y tareas no interrumpibles.

En este caso, hay una cola de eventos en el instante t, tal que $E_K[t] = \{e_k[t_0], \ldots, e_k[t_n]\}$ (con $t_n > t_0$), y procesamos el primer evento de la cola (que está pendiente) $e_k[t] = first(E_K[t]) = e_k[t_0]$, el cual es verificado por la capacidad C_k para activar la tarea $T_k[t] = C_k(e_k[t_0])$ para su lanzamiento. Las colas de eventos y tareas pueden describirse como indican las ecuaciones 3.8.

$$\begin{aligned} E_K[t+1] &= E_K[t] - first(E_K[t]) \\ T[t+1] &= T[t] \cup \begin{cases} \emptyset & \text{if } T[t] \cap \{T_k[t]\} \neq \emptyset \\ \{T_k[t]\} & \text{otherwise} \end{cases} \end{aligned} \tag{3.8}$$

En este escenario, el agente tiene recursos limitados, hay sólamente una tarea en ejecución (de un tipo de tarea específica) y el agente tiene una monitorización completa del entorno. Por lo tanto, tiene todos los eventos almacenados en la cola. En nuestro ejemplo, el Sistema de Transporte Inteligente, esta capacidad permite modelar al servicio de ETA (tiempo estimado de llegada) en las paradas de autobús. Las unidades de transporte envían un evento de localización a la parada de autobús, y la parada del autobús muestra en sus pantallas la información de ETA de cada unidad (como una cola de unidades de transporte).

3.3.6.11. Capacidad-in: eventos interrumpibles y tareas no interrumpibles.

En este caso, se asume que hay una cola de eventos $E_K[t] = \{e_k[t_n]\}$ que almacena sólamente un evento significativo (el último evento). Sin embargo, también se puede interpretar que existe una cola de eventos real en el instante t, tal que $E_K[t] = \{e_k[t_0], \ldots, e_k[t_n]\}$ (con $t_n > t_0$), y como el modelo de eventos es interrumpible, sólamente necesitamos procesar el último evento $e_k[t] = last(E_K[t]) = e_k[t_n]$, y luego procedemos a vaciar la cola de los eventos que permanecen o que están almacenados $E_K[t] = \emptyset$. Por lo tanto, una vez que el evento correspondiente es recibido, y es verificado por la capacidad C_k se activa la tarea $T_k[t] = C_k(e_k[t_n])$ para su lanzamiento. Las colas de eventos y tareas

pueden describirse como indican las ecuaciones 3.9.

$$\begin{aligned} E_K[t+1] &= E_K[t] - last(E_K[t]) \Rightarrow E_K[t+1] = \emptyset \\ T[t+1] &= T[t] \cup \begin{cases} \emptyset & \text{if } T[t] \cap \{T_k[t]\} \neq \emptyset \\ \{T_k[t]\} & \text{otherwise} \end{cases} \end{aligned} \tag{3.9}$$

En este escenario, el agente tiene recursos limitados, sólo hay una tarea en ejecución (de un tipo de tarea específica), y también una capacidad de monitorización limitada (sólo el último evento se considera como relevante). En nuestro ejemplo, el Sistema de Transporte Inteligente, esta capacidad permite modelar el servicio de ubicación de la unidad de transporte. Este servicio muestra la ubicación en la pantalla de la unidad en el mapa de ruta. Esta capacidad es computacionalmente costosa y sólo utiliza los últimos datos de la baliza.

3.3.6.12. Capacidad-nn: eventos y tareas no interrumpibles.

En este caso, se asume que hay una cola de eventos $E_K[t] = \{e_k[t_0]\}$ que almacena sólamente un evento significativo (el primer evento). Sin embargo, también se puede interpretar que existe una cola de eventos real en el instante t, tal que $E_K[t] = \{e_k[t_0], \ldots, e_k[t_n]\}$ (con $t_n > t_0$), y como el modelo de eventos es no interrumpible, sólamente necesitamos procesar el primer evento $e_k[t] = first(E_K[t]) = e_k[t_0]$, y luego procedemos a vaciar la cola de los eventos que permanecen o que están almacenados $E_K[t] = \emptyset$. Por lo tanto, una vez que el evento correspondiente es recibido, y es verificado por la capacidad C_k se activa la tarea $T_k[t] = C_k(e_k[t_0])$ para su lanzamiento. Las colas de eventos y tareas pueden describirse como indican las ecuaciones 3.10.

$$\begin{aligned} E_K[t+1] &= E_K[t] - first(E_K[t]) \Rightarrow E_K[t+1] = \emptyset \\ T[t+1] &= T[t] \cup \begin{cases} \emptyset & \text{if } T[t] \cap \{T_k[t]\} \neq \emptyset \\ \{T_k[t]\} & \text{otherwise} \end{cases} \end{aligned} \tag{3.10}$$

En este escenario, el agente tiene recursos limitados, sólo existe una tarea en ejecución (un tipo de tarea específica), y la capacidad de supervisión también es limitada (sólo el último evento es considero como relevante). En nuestro ejemplo, el sistema de transporte inteligente, esta capacidad permite modelar el servicio de búsqueda de rutas en las paradas de autobús. Este servicio calcula una ruta de transporte para el usuario, conociendo las paradas desde el punto de partida hasta el destino, e indica las posibles transferencia que necesita el usuario. Una vez que las condiciones para la búsqueda de la ruta se establecen el sistema inicia cálculo de la misma y no permite que nuevos datos se ingresen hasta terminar la búsqueda.

Este modelo extendido de capacidades y eventos le permite al modelo de agente propuesto abordar mejor diferentes escenarios de uso. Una vez concluido la descripción de los modelos para el diseño de agentes y otras entidades, se explica a continuación cómo es el proceso de desarrollo en nuestra propuesta, bajo el enfoque MDD.

3.4. Proceso de Desarrollo

Una vez que ha sido presentado el conjunto de modelos propuestos para el modelado de una *Organización Virtual Ubícua Independiente de Plataforma*, se debe definir el proceso de transformación de la VO a las diferentes plataformas de ejecución. El *proceso de desarrollo* está constituido por un conjunto de pasos o fases, que están guiados por la metodología MDD, y que concluyen finalmente con la generación de plantillas de código que posteriormente se deben compilar para obtener un MAS ejecutable. El proceso de desarrollo comienza con: (i) el primer paso, es la etapa de diseño de la organización virtual ubícua, que utiliza los conceptos y relaciones abstractas de los meta-modelos y permite crear un modelo de la VO definido por el usuario.

Este modelo de VO del usuario, se debe convertir a un modelo de la plataforma destino, utilizando las transformaciones (que se definen como un conjunto de reglas de mapeo). Por ello el segundo paso del proceso de desarrollo es, (ii) la primera etapa de transformación (el primer conjunto de reglas de mapeo) define qué conceptos del meta-modelo fuente (πVOM) son transformados a qué conceptos del meta-modelo destino. Este segundo paso, es una transformación modelo-a-modelo (PIM-a-PSM) y es ilustrada por la línea punteada de la Figura 3.11. El tercer paso del proceso de desarrollo es, (iii) la segunda etapa de transformación, donde se trasladan los modelos específicos de plataformas en plantillas de código de la organización, que pueden ser opcionalmente combinadas con código escrito manualmente por el usuario para su posterior compilación. Este tercer paso, es una transformación modelo-a-texto (PSM-a-código) y se ilustra en la Figura 3.11.

En el proceso de desarrollo, este trabajo se enfoca en el estudio de las transformaciones para dos plataformas de agentes: THOMAS[53] y E-Institutions[80]. La selección de estas plataformas se debe a que soportan de forma directa los conceptos de las organizaciones, ya que existen muchas plataformas MAS, pero no todas permiten utilizar los componentes de las organizaciones.

Es de resaltar que el proceso de transformación de la VO en diferentes plataformas, se utilizará según sea el caso y a discreción del diseñador. Por ejemplo, existen problemas que pueden resolverse utilizando solamente un modelo de una E-Institutions (o de THOMAS). En este caso, una vez que se modela y diseña

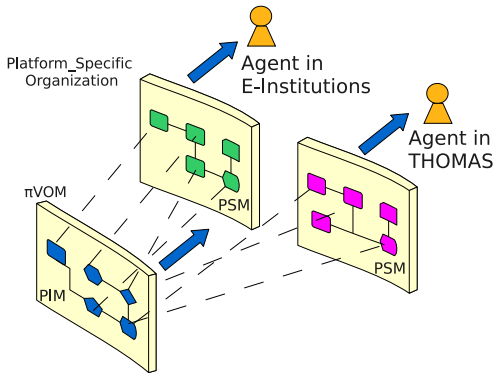

Figura 3.11: Transformación de πVOM a diferentes plataformas

la solución del problema con πVOM, sus modelos se transforman a los modelos específicos de E-Institutions (o de THOMAS, según sea el caso). Sin embargo, pueden existir problemas tan complejos que su solución pasa por usar dos organizaciones virtuales distintas. Por ello se utiliza THOMAS y E-Institutions simultáneamente, para complementar las bondades que ofrecen cada una de estas arquitecturas. En este caso el diseñador debe usar simultáneamente dos modelos de transformación, para obtener las plantillas de código de ambas organizaciones. Finalmente, una vez que se describe el proceso de desarrollo, es necesario mencionar el conjunto de herramientas que soportan todo el proceso, por ello, los pasos empleados en cada etapa del diseño y las herramientas necesarias se explican en las siguientes secciones.

3.4.1. Creación del modelo

El diseñador crea diagramas (a través de herramientas gráficas) que modelan diferentes unidades, roles, tareas, etc. del sistema a desarrollar. Para realizar este paso, se usa el Eclipse IDE[1] con un conjunto de módulos (*plugins*). Esos módulos son principalmente *EMF*, *Ecore*, *GMF*, y *GEF*, que permiten al usuario dibujar los modelos que representan la VO. Obviamente, los meta-modelos necesarios de πVOM (ver sección 2.3) deben ser cargados en el correspondiente ambiente de desarrollo (CASE tool) con el fin de generar los modelos apropiados de la VO.

Para ilustrar esta fase, se utiliza un ejemplo, un escenario que permite hacer las reservas de vuelos y hoteles. El programador debe dibujar la VO que representa la agencia de viajes (*Travel Agency*). Este escenario es modelado como una organización (la agencia de viajes), dentro de la cual hay dos unidades organizacionales (*Organizational Units*): HotelUnit y FlightUnit. Cada una dedicada a

[1] http://www.eclipse.org/

los hoteles y vuelos respectivamente. Dos clases de *Roles* pueden interactuar en la agencia de viajes: el rol del Cliente y el de Proveedor. La Figura 3.12 muestra la estructura de la agencia de viajes, con sus unidades, roles y sus relaciones de unos con otros (se usa la notación de UML, aunque con estas herramientas se puede definir una notación particular, como por ejemplo la usada en GORMAS[22]). Diagramas similares deben ser creados en esta fase de acuerdo con los diferentes modelos que forman parte de πVOM.

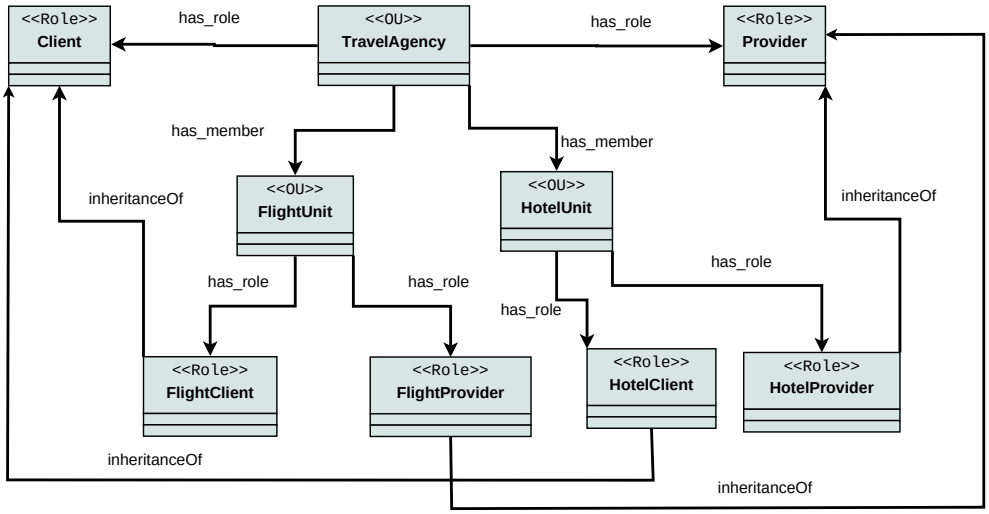

Figura 3.12: Modelo estructural de la agencia de viaje usando πVOM

3.4.2. Selección de la plataforma

Una vez que el PIM se ha completado a través de los diferentes vistas (Estructural, Funcional, Normativo, Entorno y Agentes), el desarrollador debe seleccionar las plataformas que serán utilizadas para ejecutar los diferentes componentes. El desarrollador debe escoger la plataforma o las plataformas, en la cual el usuario desea ejecutar los diferentes agentes que componen la VO. En esta fase los agentes pueden ejecutarse en diferentes plataformas de acuerdo con el modelado del sistema (escenario). Por ejemplo, un escenario posible es uno donde diferentes agentes ubícuos se ejecutan en distintas plataformas empotradas (PDAs, tabletas, teléfonos móviles o hardware empotrado) que interactúan con la *Organización Virtual* para solicitar diferentes servicios.

Para hacer esto, se debe aplicar una transformación modelo-a-modelo (PIM-a-PSM) usando Eclipse IDE con el modulo *ATL*[15] (*plugin*) que incorpora el apropiado conjunto de reglas de transformación. Es importante mencionar que el mismo modelo general de la VO puede ser transformado en distintas plataformas

específicas de VO. Las reglas para la transformación de los componentes entre los dos meta-modelos de VO (de πVOM a E-Institutions y THOMAS) es explicada en detalle en la sección 3.5. De esta manera, los conceptos de VO son mapeados desde los modelos fuentes a los modelos de destino. Es decir, los componentes de la VO son transferidos, trasladados o cambiados de un modelo a otro. Este paso se ilustra en la Figura 3.13.

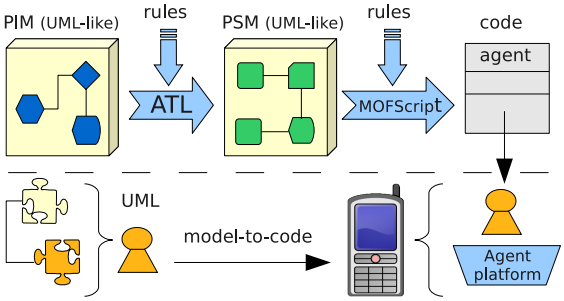

Figura 3.13: Proceso de Desarrollo para diseñar una VO usando dos etapas de transformación

Las reglas de transformación se ocultan al desarrollador y al programador, sólo se emplean cuando la plataforma de ejecución (PSM) es seleccionada. Mediante la aplicación de las reglas de transformación, el desarrollador obtiene un modelo específico para la plataforma elegida. Diferentes plataformas pueden ser elegidas para distintas partes del sistema. Después de esto, el desarrollador puede refinar el modelo agregando los detalles que se corresponden al nuevo nivel de abstracción.

Para ilustrar cómo se definen las reglas en el lenguaje *ATL*, la Figura 3.14 muestra la Regla 9 (en la sección 3.6.2, se encuentra su definición). Esta regla genera todas las *Scenes* (de las E-Institutions) a partir de las *OUs* (de πVOM) usando el mismo nombre de la Clase. Este código muestra la función `getAllRoles()`, que examina todos los *Roles* asociados con cada *OU*. La función mapea los *Roles* de los agentes que son usados en las diferentes *Scenes* (la función `getAllRoles()` también será usada por otras reglas, por ejemplo la regla utilizada en la transformación de los *Roles*).

3.4.3. Generación de código

En el último paso, el desarrollador aplica una transformación de modelo para convertir los modelos diseñados en código. Para ello, el desarrollador debe utilizar una transformación PSM-a-código. En este punto, se usa el modulo (*plugin*) *MOFScript*[15] de Eclipse que usa plantillas para realizar el proceso de traslación. Desde el punto de vista práctico, la traslación/transformación a código

```
helper context PIVOM!OrganizationalUnit
 def : getAllRoles() : OrderedSet(PIVOM!OrganizationalUnit) =
 self.children->iterate(child; hasRole: OrderedSet(PIVOM!OrganizationalUnit)=
    if child.oclIsTypeOf(PIVOM!Roles) then
          hasRole.append(child)
    endif
 );
rule OrganizationalUnit2Scene {
  from
       PIM : PIVOM!OrganizationalUnit(PIM.isOrganizationalUnit Root())
  to
       PSM : EInstitution!Scene (
          name <- PIM.name
        ...
```

Figura 3.14: Regla 9 (Unidad Organizacional a Scene) en el lenguaje ATL

consiste en recorrer un archivo XML que describe los componentes y las relaciones del meta-modelo fuente y luego generar otro archivo XML que contiene la especificación de las plataformas E-Institution o THOMAS que será usado como el lanzador de la aplicación. Esta fase se ilustra en la Figura 3.13, y se asume que el agente está ejecutándose en un dispositivo portátil, específicamente en un teléfono móvil.

La Figura 3.15 ilustra como una regla de transformación es implementada usando *MOFScript*, en concreto corresponde a la **Regla 17** (véase la sección 3.7.1). Esta regla genera el código para el concepto *Agente* en la plataforma JADE. Estas plantillas han sido desarrolladas para JADE, E-Institution y THOMAS. Adicionalmente pueden definirse transformaciones para otras plataformas de ejecución, para ello sería necesario específicar la reglas que mapean los conceptos a la plataforma destino.

3.5. Reglas de Transformación

Una vez que se ha presentado el meta-modelo de la *Organización Virtual* y se ha explicado el *Proceso de Desarrollo*, las reglas de transformación desde un PIM a diferentes PSMs deben ser descritas. Para ello, aplicamos una transformación de modelo-a-modelo (PIM-a-PSM). Los componentes y conceptos son trasladados o cambiados a partir de un modelo a otro. Estas transformaciones se realizan en dos niveles: a nivel de organización (marco organizacional) y a nivel de los agentes (miembros de la organización). Una vez que se obtienen los PSMs se deben obtener las plantillas de código de las organizaciones y agentes para su posterior ejecución.

La Figura 3.16 muestra un esquema completo de la propuesta. El proceso se inicia con el diseño de las entidades, agentes, servicios, dispositivos, eventos etc.,

```
texttransformation UMLAGENT2JADEM (in myAgentModel:uml2)
...
//Rule17: Agent transformation
uml.Package::mapPackage () {
  self.ownedMember->forEach(c:uml.Class)
     if (c.name != null) if (c.name = Agent) c.outputGeneralization()
}
uml.Class::outputGeneralization(){
   file (package_dir + self.name + ext)
   self.classPackage()
   self.standardClassImport ()
   self.standardClassHeaderComment ()
   <% public class %> self.name <% extends Agent { %>
      self.classConstructor()
      <% // Attributes  %>
      self.ownedAttribute->forEach(p : uml.Property) {
            p.classPrivateAttribute()
      }
      newline(2)
 <%}%>
...
}
```

Figura 3.15: Ejemplo de transformación del concepto *Agente* usando MOFScript.

que componen la organización (estos son modelos independientes de plataforma). Estos modelos diseñados deben ser transformados a las entidades, servicios del modelo de implementación de la Organización Virtual, que en nuestra propuesta corresponde a las plataformas THOMAS[2] y E-Institutions[3]. La transformación de la plataforma de implementación también involucra la plataforma de agentes, que en nuestra propuesta puede ser la conocida JADE[4] o JADE-Leap (para agentes empotrados) y la plataforma de ejecución ANDROMEDA[5].

Una vez que se obtienen los modelos de implementación se debe obtener el modelo de implantación. En nuestra propuesta se genera código final para su compilación o código intermedio para ser cargado por un lanzador, según la plataforma seleccionada.

3.6. Transformación a nivel organizacional

En esta sección se explica la forma de traducir el modelo que representa el marco de la organización (PIM), en dos modelos de la plataforma de destino (PSMs). Los PSMs elegidos son (como se comentó anteriormente): *THOMAS* y

[2]http://users.dsic.upv.es/grupos/ia/sma/tools/Thomas
[3]http://e-institutions.iiia.csic.es
[4]http://jade.tilab.com/
[5]http://users.dsic.upv.es/grupos/ia/sma/tools/Andromeda

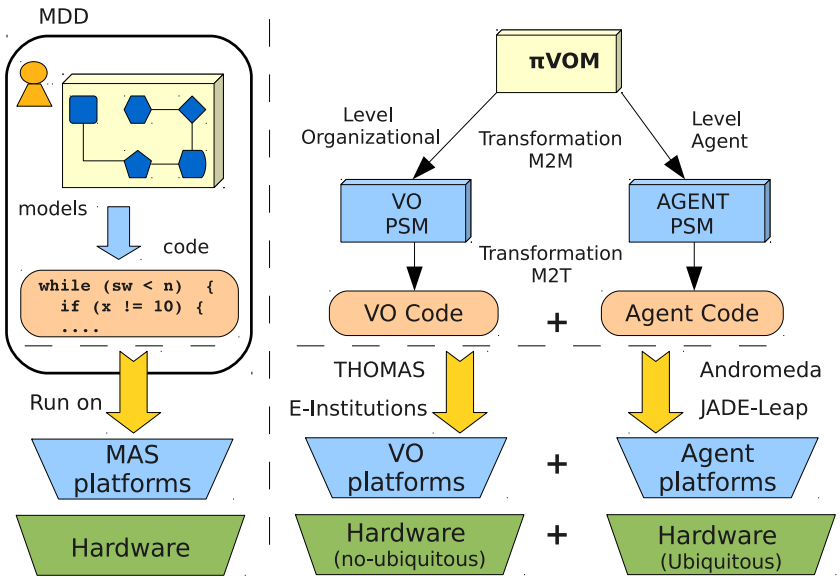

Figura 3.16: Visión MDD de la propuesta: desde el diseño hasta el código

E-Institutions. El mismo proceso tiene que ser hecho para cualquier otra plataforma. La única limitación es que la plataforma debe incluir los conceptos organizacionales[11].

3.6.1. THOMAS: Marco de Ejecución

THOMAS (por su siglas en inglés, MeTHods, Techniques and Tools for Open Multi-Agent Systems), es una arquitectura de sistemas multi-agente abierta que está compuesta de un conjunto de módulos interconectados que son adecuados para el desarrollo de sistemas aplicados en entornos que trabajan como una "sociedad"[116]. Debido a los avances tecnológicos de los recientes años, el término "sociedad", necesariamente debe cumplir varios requisitos: distribución, evolución constante, flexibilidad para permitir que miembros puedan entrar y salir de la sociedad; un apropiado manejo de la estructura organizativa que define a la sociedad; ejecución de los agentes en multi-dispositivos incluyendo dispositivos con recursos limitados.

El meta-modelo πVOM, presentado en este trabajo, es muy similar al modelo de organización que está implementado en THOMAS, dado que ambos trabajos están parcialmente basados en la metodología y componentes propuestos por GORMAS (esta metodología y sus componentes se describe en [22]). Por esta razón, las transformaciones automáticas son relativamente fáciles de describir. Casi la totalidad de los conceptos abstractos de πVOM están representados en THO-

MAS, por lo que las reglas de transformación de modelo-a-modelo se expresan casi como una relación uno-a-uno. Es conveniente tener en cuenta que algunos de los conceptos en THOMAS tienen una característica más detallada que en, πVOM, esto se debe a que THOMAS es un modelo de una plataforma-específica.

Aunque en este caso la transformación es una relación uno-a-uno, existen otras plataformas de VO en que las transformaciones no serán directas. Podemos pensar en las reglas de transformación como una función que convierte los modelos de πVOM a modelos específicos de plataforma. En el caso de πVOM a THOMAS, la función de transformación es como una relación "lineal", pero existen otras plataformas, como E-Institutions (o otras), donde la función de transformación no es directa. En este caso, se debe cambiar esta función de transformación a otra distinta a la usada en THOMAS. El enfoque propuesto permite especificar una función de transformación para cada plataforma específica de VO que pretende ser usada.

Las principales reglas de transformación que deben realizar la traducción entre los diferentes modelos se muestran en la Tabla 3.6 (desde la **Regla 1** a la **Regla 8**). Se puede observar en la Tabla 3.6, que entre los dos modelos (PIM y PSM) hay casi un mapeo uno a uno, y estas reglas se explican a continuación.

Tabla 3.6: Reglas de πVOM a la plataforma THOMAS

Reglas	Concepto	Transformación a THOMAS
1	Organizational Unit	πVOM.OU $\Rightarrow$ THOMAS.OU
2	Agent	πVOM.Agent $\Rightarrow$ THOMAS.Agent
3	Role	πVOM.Role $\Rightarrow$ THOMAS.Role
4	Service	πVOM.Service $\Rightarrow$ THOMAS.Service
5	Norm	πVOM.Norm $\Rightarrow$ THOMAS.Norm
6	RelationType	πVOM.RelationType $\Rightarrow$ THOMAS.Process
7	Resource	πVOM.Resource $\Rightarrow$ THOMAS.Service
8	Goal	πVOM.Goal $\Rightarrow$ THOMAS.Goal

- La **Regla 1** hace referencia que una unidad organizacional de πVOM, que corresponde exactamente a una unidad organizacional de THOMAS.
- La **Regla 2** realiza la transformación del agente. La conversión es directa, ya que los *Agents* son parte de la organización en THOMAS. Los agentes son los que entran/salen de las unidades organizacionales y usan/consumen los servicios.
- La **Regla 3** permite la transformación del concepto *Rol*, que especifica el patrón de comportamiento de la *Entidad* en los meta-modelos de πVOM, y coincide con la definición en THOMAS.

- La **Regla 4** transforma el *Servicio*, que representa la funcionalidad que agentes y OUs ofrecen a otras entidades tanto en los meta-modelos de πVOM y en THOMAS. La transformación implica que el concepto servicio se debe definir y registrar en el *Service Facilitator* (SF) de THOMAS. Este componente publicita servicios (simples y complejos) ofertados por los agentes y organizaciones activas.

- La **Regla 5** mapea las *Normas*. Las normas son un mecanismo para limitar la autonomía de los miembros de la organización a través de los roles que juegan las entidades en THOMAS. En este caso la transformación también es directa.

- La **Regla 6** traslada el concepto *RelationType*, que específica el tipo de relación (flujo) entre los servicios y sub-servicios. Esto se corresponde con la utilidad *Process* en THOMAS que proporciona el SF. La cual define cómo es la implementación del servicio, en otras palabras, el proceso a seguir de pasos para poner en funcionamiento el servicio.

- La **Regla 7** transforma el concepto *Resource*. Un *Resource* representa un componente del entorno, y debe ser mapeado como un servicio simple en THOMAS. Las acciones realizadas por el *Resource* son trasladadas a las funcionalidades de un servicio del entorno.

- La **Regla 8** hace referencia al concepto *Goal*. Este concepto representa un estado que la organización y/o los agentes están tratando de alcanzar, y es es mapeado de forma directa en el concepto *Goal* de THOMAS que se define como el objetivo final de un determinado servicio.

3.6.2. Plataforma E-Institutions

E-institutions proporciona un conjunto de herramientas que son ampliamente utilizadas con el fin de modelar organizaciones electrónicas. E-institutions puede ser efectivamente diseñada e implementada como una *institución electrónica* compuesta por un gran número de agentes heterogéneos (humanos y software) que desempeñan diferentes funciones e interactúan por medio de actos de habla (*speech acts*)[80]. Esta plataforma está basada en las tradicionales organizaciones humanas y ofrece un modelo computacional general mediado por agentes, que sirve para crear una plataforma para *instituciones electrónicas mediadas por agentes*. La Figura 3.17 muestra el modelo de E-Institutions (la relación entre los principales componentes). La Tabla 3.7 resume los principales conceptos usados en E-Institutions.

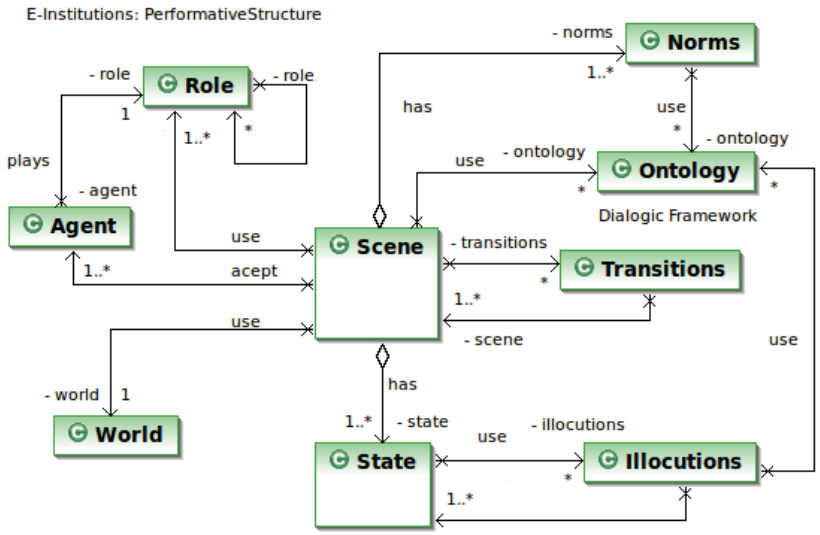

Figura 3.17: Principales conceptos usados en E-Institutions (meta-modelo destino)

Las principales reglas de transformación de πVOM a E-Institutions son mostradas en la Tabla 3.8 (desde la **Regla 9** a la **Regla 16**). Las cuales se describen a continuación:

- La **Regla 9**, indica que cada OU es una Scene. Las escenas son las entidades donde los agentes colaboran para llevar a cabo las acciones de la organización.

- Las **Regla 10** y **Regla 11** tiene un mapeo 1 a 1, dado que ambos conceptos *Agent* y *Role* son representados en las dos plataformas. Es decir, los *Agents* son parte de la organización en ambas plataformas y en cada una son asignados los *Role* usados.

- La **Regla 12**, hace referencia al servicio, que representa la funcionalidad de una OU. En E-Institutions el estado (*State*) puede ofrecer una funcionalidad. Por ello, un *SingleService* debe ser traducido como un *State*. En el caso que exista un servicio compuesto (*ComposedService*), éste puede ser traducido en un conjunto de estados *States* que proporcionan la funcionalidad en una escena *Scene*.

- La **Regla 13**, describe el nivel de flujo de trabajo entre los servicios. Esta regla está estrechamente relacionada con la *Regla 12* y la *Regla 9*. Si el *RelationType* está ubicado entre *SingleService*, la *RelationType* debe ser transformada en una *Illocution*, ya que representa el flujo entre los estados

Tabla 3.7: Principales componentes usados en E-Institutions

Componentes de E-Institutions	Descripción
Agent	Una entidad racional y autónoma dentro de la E-Institutions.
Role	Especificación del patrón de comportamiento esperado de los agentes de la E-Institution.
Scene	Una escena es un patrón de interacción del MAS.
State	Es un nodo de un grafo orientado de estado finito, el cual describe el protocolo de la escena (Scene).
Transitions	Especificación de un flujo de trabajo entre las escenas (Scenes).
Illocutions	Las expresiones válidas del lenguaje de comunicación entre agentes. Representa los arcos entre los estados (States).
Ontology	Es el conocimiento del agente y de la E-Institutions.
World	Punto de acceso para interactuar con el entorno.
Norm	Un conjunto de reglas que se utilizan como mecanismos para limitar la autonomía de los agentes en la E-Institution.

Tabla 3.8: Reglas de πVOM a E-Institutions

Regla	Conceptos	Transformación a E-Institutions
9	Organizational Unit	πVOM.OU $\Rightarrow$ EI.Scene
10	Agent	πVOM.Agent $\Rightarrow$ EI.Agent
11	Role	πVOM.Role $\Rightarrow$ EI.Role
12	SingleService	πVOM.Service $\in$ OU $\Rightarrow$ EI.State $\in$ Scene
13	RelationType	πVOM.RelationType $\Rightarrow$ EI.Transition OR EI.Illocutions
14	Norm	πVOM.Norm $\Rightarrow$ EI.Norm
15	Goal	πVOM.Goal $\Rightarrow$ EI.Norm
16	Resource	πVOM.Resource $\Rightarrow$ EI.World

en E-Institutions. Sin embargo, cuando la *RelationType* se ubica entre *ComposedService*, esta *RelationType* permite conocer el flujo (la dirección en el movimiento) que hay entre cada unidad organizacional, que es equivalente al flujo entre las escenas. En este caso, la traducción de la *RelationType* corresponde a una *Transition* en E-Institution.

- La **Regla 14** y **Regla 15**, describen las normas en E-Institutions, es decir,

lo que un agente puede hacer en cada escena y estado. Por esta razón, las *Norms* y objetivos perseguidos (*Goals*) en πVOM son traducidos a las *Norms* en E-Institutions. Ellas específican lo que un agente tiene permitido hacer. El *Goal* es traducido como una norma, ya que representa lo que un agente debe hacer para alcanzar una meta, éste puede ser representado como una directriz o pauta que debe realizar el agente.

- **Regla 16**, describe los recursos que son accesibles por los agentes o entidades (en πVOM). Estos son traducidos al concepto abstracto del entorno en E-Institutions (*World*).

Estas reglas de transformación generan plantillas de código para su ejecución (java, para su compilación) o código intermedio (XML, para ser cargado por un lanzador y ser ejecutado posteriormente). Las reglas de transformación permiten demostrar que nuestro meta-modelo tiene la generalidad suficiente como para transferir el diseño del MAS a diferentes plataformas orientadas a la organización.

3.7. Transformación a nivel de agente

Como se mencionó anteriormente, cada agente identificado en el sistema debe modelarse de acuerdo con el meta-modelo de agente propuesto. De esta forma, cada agente modelado en el sistema puede ser transformado en código de acuerdo con la plataforma de agente específica donde el agente es ejecutado. El modelo de agente (de la plataforma destino) a ser analizado en este trabajo es el de la plataforma JADE y Andromeda.

3.7.1. Plataforma JADE

JADE[30], es una de las plataformas más populares que soporta la ejecución de agentes, y se utiliza ampliamente, ya que proporciona los conceptos de programación que simplifican la implementación de un MAS. JADE es compatible con FIPA para la infraestructura de comunicación entre agentes. Una de las ventajas de esta plataforma es que es soportada por THOMAS y E-Institutions.

Uno de los principales conceptos usados en la implementación del agente JADE es el Comportamiento (*Behaviour*)[6]. Un *Behaviour* representa una tarea específica que el agente ejecuta. Hay diferentes tipos de comportamientos que el agente puede ejecutar. Para dar soporte a esto JADE proporciona diferentes clases, las cuales son una especialización de *SimpleBehaviour*, tales como: temporal,

[6]el cual tiene un significado diferente con respecto al meta-modelo de agente propuesto en este trabajo

secuencial, y paralelo, etc. La Tabla 3.9 resume los principales comportamientos (*Behaviours*) usados por los agentes en JADE.

El paradigma de comunicación adoptado por JADE es el paso de mensajes asíncronos. Un agente debe compartir el mismo lenguaje, vocabulario y protocolo. Esto pasa por definir una ontología (*ontology*), que permita utilizar una semántica en el contenido de los mensajes intercambiados entre los agentes. Otro concepto importante en JADE es el esquema (*Schema*). Un esquema es un marco estructurado, que representa la estructura de los conceptos que conforman una ontología. Los esquemas de JADE son conceptos que ofrecen un tipo de estructura de datos que directamente mapea la estructura de una ontología. La Tabla 3.9 resume los principales conceptos usados en JADE.

Tabla 3.9: Los componentes del modelo de JADE

Concepto	Uso	Descripción
Agent	Behaviour	Las tareas que un agente ejecuta.
	Ontology	El conocimiento del agente.
Ontology	Schema	Estructura de datos del mensaje.
ACL Communications	Performative	Mensajes compatibles con FIPA.
Tipo Behaviours	**Especialización**	**Descripción.**
SimpleBehaviour	OneShot	Ejecuta una acción solo una vez.
	Ticker	Ejecuta una acción periódicamente.
	Weaker	Espera un periodo de tiempo para ejecutar una acción.
	Cyclic	Ejecuta una acción cíclicamente.
CompositeBehaviour	Sequential	Ejecuta varias acciones secuencialmente.
	FSM	Las acciones son ejecutadas como una máquina de estados finita.
	Parallel	Ejecuta varias acciones en paralelo.

Por otra parte la Tabla 3.10 muestra las reglas de transformación (desde la **Regla 17** a la **Regla 24**) necesarias que se han definido para traducir del modelo de agente en πVOM al modelo de agente en JADE (modelo PSM), que será conocido en este trabajo como JADEM. A continuación se describen las principales reglas de transformación:

- La **Regla 17** realiza la transformación del agente. La conversión es directa ya que nuestro modelo de agente coincide con el modelo de agente en JADE. Después de la transformación, los métodos tienen que ser revisados para comprobar que el agente JADE funciona correctamente.

Tabla 3.10: Reglas de transformación del meta-modelo de *agente* a JADE

Regla	Concepto	Transformación a JADE
17	Agent	πVOM.Agent ⇒ JADEM.Agent
18	Behaviour	πVOM.Behaviour ⇒ JADEM.ParallelBehaviour
19	Capability	πVOM.Capability ⇒ JADEM.OneShotBehaviour
20	Task	πVOM.Task ⇒ JADEM.Behaviour
21	Events	πVOM.Event ⇒ JADEM.ACLMessage
22	Beliefs	πVOM.BeliefContainer ⇒ JADEM.Schema
23	Goal	πVOM.Goal ⇒ JADEM.Ontology
24	Message	πVOM.Message ⇒ JADEM.ACLMessage

Cada modelo de agente tiene sus métodos, uno de los más importante es `init()`, ya que este método contiene el código que debe ejecutar. Por ello, el método `init()` se traslada al método `setup()` de JADEM (que es donde se escribe lo que el agente ejecuta). Otros métodos del modelo del agente también son transformados: el método que destruye al agente `destroy()` → `takeDown()` y el método que agrega los comportamientos `addBehav()` → `addBehaviour()`.

- La **Regla 18** realiza la transformación del comportamiento. Un comportamiento (*Behaviour*) en nuestro modelo, es un conjunto de acciones que pueden ser ejecutadas. Por ello, un *Behaviour* se traduce como un `CompositeBehaviour` en JADEM. Específicamente, cada *Behaviour* utilizado en nuestro modelo se transforma en un `ParallelBehaviour` en JADEM. Este *ParallelBehaviour* estará vacío en un comienzo, hasta que se realice la transformación de *Capability* y *Task* del agente.

- La **Regla 19** realiza la transformación de la capacidad. Una *Capability* es un componente que puede o no lanzar una actividad dependiendo de la llegada de un evento. Para emular está funcionalidad, cada *Capability* se traduce a un *simpleBehaviour* en JADEM, el cual tiene como objetivo verificar la llegada de un evento. Si el evento es el correcto, entonces la actividad será puesta en marcha.

- La **Regla 20** describe la transformación de la tarea. Una *Task* en nuestro modelo de agente puede ser una acción simple o compleja. El tipo de *Task* establece una transformación específica a un `SimpleBehaviour` o a un `CompositeBehaviour`. Por ejemplo, si hay una tarea cíclica en el agente, un `CyclicBehaviour()` debe ser agregado en JADEM. Por cada *Task* definida en el agente, un tipo específico de *Behaviour* debe ser añadido en

JADEM. La *Task* es el lugar donde los usuarios escriben su código, específicamente en el método `doing()`. Por ello, este método se transforma en el método `action()` en JADE (que es el método donde los usuarios escriben su código).

- La **Regla 21** permite la transformación de un evento (*Event*). Esta transformación no es directa, sin embargo el funcionamiento de un evento se puede emular fácilmente, al utilizar o hacer corresponder para cada tipo de evento un tipo de mensaje ACL en JADE con una performativa concreta.

- La **Regla 22** permite traducir el *BeliefContainer*, éste almacena el conocimiento del agente (lo cual corresponde al concepto de *Schema* en JADE), que se utiliza en la definición de la ontología.

Una vez obtenido el modelo PSM a partir del PIM, el modelo PSM debe ser transformado en código (PSM-a-código), traduciendo cada concepto que se incluye en el modelo PSM a una plantilla de código. Esta transformación se realiza como se describió en la sección 3.4.3 y que se ilustra en la Figura 3.18. Se ejecuta cuando se necesitan obtener las plantillas de código en JADE (que el usuario combina con código escrito manualmente) para su posterior compilación. En dicha Figura se observan las dos transformaciones necesarias en este proceso: la primera entre modelos, se parte del PIM para obtener el PSM (de πVOM a JADEM) y la segunda transformación, de PSM a plantilla de código (de JADEM a código Java).

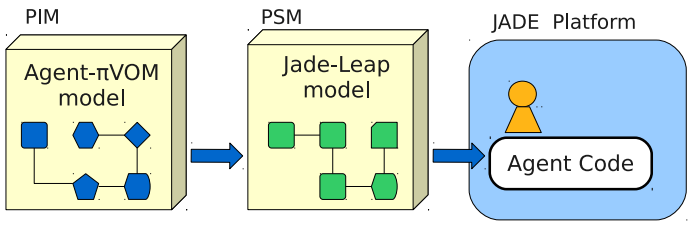

Figura 3.18: Proceso de transformación MDD para JADE

Finalmente, la Figura 3.19 muestra, como ejemplo, una plantilla de código obtenida para el modelo de agente en JADE. En este caso, se muestra una plantilla de código de un agente de un pasajero, este agente `Passenger` es cliente de un *Sistema de Transporte Inteligente* (ejemplo que se describe más adelante, en la sección 6.3.5.5).

También se observa en la Figura 3.19 que el modelo del agente tiene un comportamiento (*Behaviour*) llamado `StopClient` que le permite al agente interactuar con el agente de la parada (*Stop*). Este comportamiento tiene una capacidad

```
public class Passenger extends Agent {
    ...
   protected void setup(){                                    Behaviour
      ParallelBehaviour StopClient =
         new ParallelBehaviour( ParallelBehaviour.WHEN_ALL );

      StopClient.addSubBehaviour( new SearchRoute(this) );
      ...
      addBehaviour(StopClient);
    }  // ----------- end setup()
                                                              Capability
      class SearchRoute extends OneShotBehaviour {
          public SearchRoute(Agent a) {
            super(a); }

          public void action() {
            ... //check condition == true
            addBehaviour(new TaskRoute(this) );
          }
      } // ----------- end OneShotBehaviour

      ...                                                     Task
      class TaskRoute extends Behaviour {
          public TaskRoute(Agent a) {
             super(a); }

          public void action()  {
              //...this is where the code Task goes !!
          }
      } // ----------- end Behaviour

} // ----------- end class Agent
```

Figura 3.19: Plantilla de código de un agente para la plataforma JADE

(*Capability*) llamada `SearchRoute` que le permite al agente solicitar un servicio a la parada. Este servicio es para buscar la ruta más corta entre la parada donde se ubica el pasajero y un punto de destino.

3.7.2. ANDROMEDA

Andromeda[2, 3] es una plataforma de ejecución de agentes que implementa los conceptos propuestos en el modelo de agente de πVOM y que fue desarrollado para evaluar las propuestas hechas por esta Tesis. Andromeda permite la ejecución de agentes en dispositivos móviles y empotrados que soporten la plataforma *Android*[7] de Google. Andromeda proporciona al programador los componentes de programación que simplifican la implementación del agente πVOM, y es compatible con FIPA en su infraestructura de comunicación entre agentes. Por ello, puede registrarse en una plataforma JADE (y en otras, como MAGENTIX2[188]), para posteriormente interactuar con plataformas de organizaciones como THOMAS y E-Institutions. Andromeda es explicado con más detalle en el capítulo 5, ya que se presenta como una nueva plataforma de ejecución de agentes que fue desarrollada en el transcurso de esta Tesis.

Como Andromeda se ha diseñado usando los mismos conceptos del meta-

[7]http://code.google.com/android/

modelo de π *VOM*. Un agente en ANDROMEDA se implementa utilizando los mismos conceptos empleados en el meta-modelo abstracto. Este diseño simplifica en gran medida la transformación automática entre el PIM y PSM, debido a que el PSM de ANDROMEDA es muy similar al PIM propuesto a nivel abstracto. La Tabla 3.11 muestra un sub-conjunto de las relaciones o reglas entre ambos meta-modelos (desde la **Regla 25** a la **Regla 32**).

Tabla 3.11: Reglas de transformación del meta-modelo de *agente* a ANDROMEDA

Regla	Concepto	Transformación a ANDROMEDA
25	Agent	πVOM.Agent $\Rightarrow$ ANDROMEDA.Agent
26	Behaviour	πVOM.Behaviour $\Rightarrow$ ANDROMEDA.Behaviour
27	Capability	πVOM.Capability $\Rightarrow$ ANDROMEDA.Capability
28	Task	πVOM.Task $\Rightarrow$ ANDROMEDA.Task
29	Events	πVOM.Event $\Rightarrow$ ANDROMEDA.Event
30	Beliefs	πVOM.BeliefContainer $\Rightarrow$ ANDROMEDA.BeliefContainer
31	Goal	πVOM.Goal $\Rightarrow$ ANDROMEDA.Goaly
32	Message	πVOM.Message $\Rightarrow$ ANDROMEDA.Message

Como se observa la relación de transformación es uno-a-uno. En consecuencia, de manera práctica el segundo paso del proceso de transformación MDD no es necesario. En este caso, las reglas de transformación necesarias son principalmente de modelo-a-texto, generando directamente el código de ANDROMEDA. De esta forma, los dos últimos pasos del proceso de transformación MDD pueden combinarse en uno solo, tal y como se muestra en la Figura 3.20.

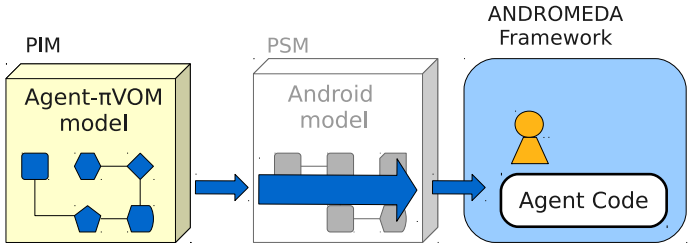

Figura 3.20: Proceso de transformación MDD para ANDROMEDA

Por lo tanto, la transformación necesaria es la de código, y una vez que se obtiene la plantilla de código para ANDROMEDA, ésta puede ser combinada con código adicional proporcionado por el desarrollador. La Figura 3.21 muestra esta plantilla. De forma similar al agente de JADE, este caso corresponde al código de un agente de un pasajero. Este agente `Passenger` es cliente de un *Sistema de Transporte Inteligente*, que se describe más adelante, en la sección 6.3.5.5.

```
public class Passenger extends Agent {
     public void init(){
          . . .
      //define agent components
                                                    Behaviour
      Behaviour StopClient= new Behaviour("StopClient");
                                                    Capability
      Capability SearchRoute =  new Capability("Route");

       Task TaskSearch =new Task("TaskSearch");

        . . .
        //add objects into of the agent components
        SearchRoute.addTaskRun(TaskRoute);
        StopClient.add(SearchRoute);
        addbehav(StopClient);
    }  // ----------- end init()
                                                    Task
   class TaskSearch extends Task{
      doIt {
        //here the user write the  code
      }  // ----------- end doIt
      . . .
   }  // ----------- end Class Task

    . . .
}  // ----------- end Class Agent
```

Figura 3.21: Plantilla de código de un agente para ANDROMEDA

En la plantilla se observa (en la Figura 3.21) que el agente `Passenger` tiene un comportamiento (*Behaviour*) llamado `StopClient` que le permite al agente interactuar con el agente de la parada (*Stop*). Este comportamiento tiene una capacidad (*Capability*) llamada `SearchRoute` que le permite al agente solicitar un servicio a la parada. Este servicio es para buscar la ruta más corta entre la parada donde se ubica el pasajero y un punto de destino. En la tarea `TaskSearch`, es donde el usuario debe escribir su código de forma manual.

Finalmente, una vez comentadas todas las transformaciones, se obtiene una metodología versátil para el desarrollo e implementación de MAS (con un enfoque MDD), ya que se parte de modelos genéricos y unificados en el diseño y posteriormente por medio de transformaciones se puede obtener las plantillas de código para la implementación del MAS en las plataformas de ejecución finales. Simplemente con variar las reglas de transformación se puede cambiar la plataforma de ejecución del MAS, factor muy valorado en la implementación de agentes.

3.8. Discusión y Conclusiones

Este capítulo ha presentado un enfoque MDD para desarrollar *Organizaciones Abiertas Basadas en Agentes*. El MDD puede ser considerado como un nuevo paradigma para desarrollar sistemas de software en el cual diferentes etapas del proceso de desarrollo software pueden conectarse automáticamente mediante la

definición de mapeos y transformaciones. Por lo tanto, en el contexto del uso del MDD en la *Ingeniería de Software Orientada a Agentes* (AOSE, de sus siglas en inglés *Agent Oriented Software Engineering*) se han identificado las siguientes ventajas que nuestro enfoque MDD ofrece:

- El uso de un meta-modelo abstracto para diseñar y modelar Sistemas Multi-Agente basados en Organizaciones Virtuales denominado πVOM.
- La generación de un proceso de transformación de PIM a PSM, el cual proporciona una interfaz simple para implementar los modelos generados de πVOM. Por lo tanto, este enfoque reduce:
  - La brecha entre el diseño y la implementación.
  - El conocimiento requerido para la implementación de un MAS con respecto a las plataformas de implantación (*deployment*) del MAS.

Enfoques similares pueden ser observados en trabajos como: TROPOS, Sage, y MetaDIMA. Sin embargo, estos trabajos usan meta-modelos propietarios, y típicamente generan implantaciones que solo se pueden ejecutar en plataformas específicas.

En trabajos como TAO, FAML, Agent UML (AUML), y AML, el principal objetivo, es proporcionar un *Marco Conceptual Unificado y Genérico* que pueda modelar distintas abstracciones con el fin de soportar el diseño de agentes de diferentes metodologías de MAS. Estos enfoques se centran principalmente en la fase de análisis, mientras que la fase de implementación no existe o es mínima. En cambio, el enfoque presentado en este trabajo proporciona un meta-modelo relativamente genérico, que permite analizar algunas metodología MAS, y adicionalmente busca obtener implementaciones de los diseños realizados.

Trabajos como PIM4AGENT y CAFnE están dirigidos a crear un meta-modelo unificado que permita diseñar agentes con algunas metodologías MAS, y también permiten la generación de código del agente. Estas implementaciones pueden ejecutarse en diferentes plataformas. Sin embargo, estas aproximaciones tienen una visión limitada de la organización de agentes (así como los trabajos FAML y TAO), ninguna de ellas trata o soporta organizaciones (*Organizaciones Virtuales*). Además, prácticamente ningún trabajo contempla la interacción de las entidades del MAS con el entorno físico, enfoque que limita la implementación de estos sistemas en escenarios donde sea necesario percibir el mundo real, como lo es en los sistemas ubícuos.

Este trabajo presenta cómo desarrollar *Organizaciones Virtuales Ubícuas* usando MDD de una forma completa. Proporcionando un meta-modelo de *Organización Virtual* (llamado πVOM), el cual permite que las organizaciones en MAS sean

modeladas usando componentes abstractos que son independientes de la plataforma de implementación. Este meta-modelo se divide en cinco vistas que se centran en los aspectos más importantes de las *Organizaciones Virtuales*. Estas vistas se pueden extender fácilmente a un dominio específico si es necesario.

El meta-modelo propuesto puede ser usado para crear plantillas de código para plataformas de organización específicas. Este trabajo ha discutido el uso de las transformaciones en las plataformas THOMAS y E-Institutions. Esas transformaciones muestran que el meta-modelo puede ser considerado independiente de plataforma. Este trabajo ha sido probado a diferentes niveles de abstracción. En [7, 5, 11] las transformaciones a nivel de plataforma de organizaciones fue evaluado, al mismo tiempo que las transformación a nivel de agente se verificaron en [6, 8].

4

Arquitectura de implantación

Índice

4.1. Introducción

Tal y como se ha ido observando a lo largo de esta Tesis, este trabajo propone usar un modelo homogéneo y unificado para implementar una *Organización Virtual Ubícua* que permite su traslación (transformación) en diferentes plataformas de ejecución de MAS a través de MDD. En este sistema los agentes actúan-perciben acerca de su entorno y por ello administran y controlan al *Sistema Ubícuo*. Esto implica que el usuario puede diseñar el sistema ubícuo con modelos

unificados, intuitivos y visuales, es decir, con un alto nivel de abstracción. Posteriormente, el usuario puede obtener el código de la *Organización Virtual Ubícua* automáticamente usando transformaciones MDD con una mínima intervención del usuario. Finalmente, los controladores (*drivers*) o firmware deben ser añadidos para soportar los dispositivos del entorno. Los *drivers* deben ser encapsulados dentro de un servicio (un servicio OSGi), para exportar sus funcionalidades como una abstracción de alto nivel (la cual es administrada por agentes). Finalmente, este código debe ser compilado para su ejecución en la plataforma OSGi[1] (Open Service Gateway Initiative). Todo este proceso se muestra de forma resumida en la Figura 4.1.

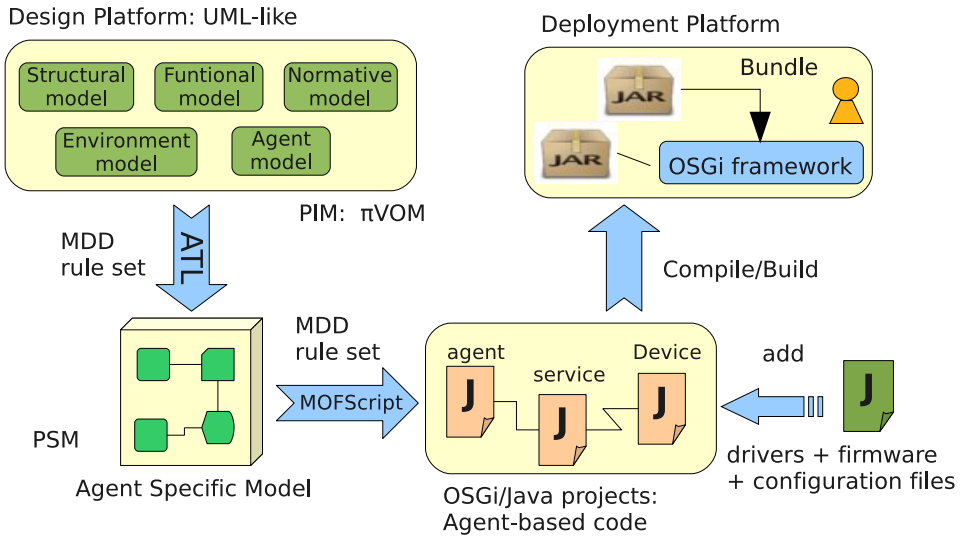

Figura 4.1: Implementación de una Organización Virtual Ubícua usando MDD

En este capítulo se describe en detalle la arquitectura de software y hardware sobre la que se puede apoyar el proceso de desarrollo visto en el capítulo anterior.

4.1.1. Desde el diseño hasta la implantación

El proceso de diseño y especificación de *Sistema Ubícuo* arranca modelizando los agentes, entidades de la VO y los dispositivos usando los componentes abstractos de πVOM (el meta-modelo propuesto, comentado en el capítulo anterior). El proceso continua trasladando los modelos abstractos a una plataforma de implementación (como se observa en la Figura 4.1). Esto se realiza usando un conjunto de transformaciones (del tipo modelo-a-modelo: M2M) a dos niveles para la VO

[1] http://www.osgi.org/

y para la plataforma de agente. Se continua con un mapeo de los componentes específicos de cada plataforma para obtener código final (como Java) o para un lanzador (como XML). Además, una vez obtenido el código, éste debe ser combinado con el código del usuario, drivers, y fichero/código de configuración, para finalmente obtener código OSGi-Java para ser compilado y ejecutado en la plataforma hardware seleccionada (la plataforma de implantación). La propuesta ubícua soporta un sistema heterogéneo y distribuido, con diferentes componentes hardware interconectados a través de redes que permiten la ejecución de los agentes con sus servicios. Esta propuesta se resume en la Figura 4.2 y de la cual extraemos los siguientes pasos desde el proceso de diseño hasta la implantación de la organización ubícua.

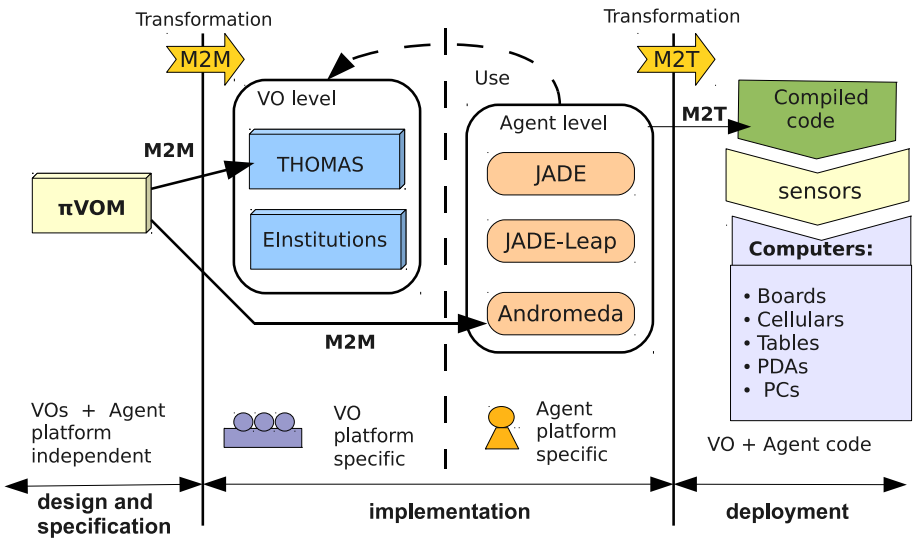

Figura 4.2: Esquema completo de la propuesta: desde el diseño hasta la implantación

- **Paso 1:** Para comenzar, el desarrollador debe crear diferentes diagramas usando la herramienta gráfica, que modela los diferentes dispositivos, recursos o servicios de los agentes e incorpora los meta-modelos definidos. Para ejecutar este paso, usamos el IDE de Eclipse con un conjunto de librerías (o *plugins*), que permiten al usuario dibujar los modelos que representan al *sistema ubícuo* basado en los componentes, conceptos y relaciones de πVOM.

- **Paso 2:** Una vez que se completa el modelo, es necesario seleccionar en qué plataforma el usuario desea ejecutar los agentes. Esta fase corresponde con la definición del modelo PSM de cada agente. Para hacer esto, es necesario

aplicar una transformación modelo-a-modelo (PIM-a-PSM). Esto se realiza con la ayuda del *plugin* conocido como ATL del IDE de Eclipse el cual tiene cargado con anterioridad el conjunto de reglas de transformación. Es importante destacar que el mismo modelo de agente puede ser transformado en diferentes arquitecturas de agentes, que son específicos de cada plataforma. Las Tablas mostradas en el capítulo anterior ilustran como son las reglas de transformación para nuestro modelo de agente a otras plataformas MAS o de VOs, como por ejemplo a THOMAS[4], ANDROMEDA[3] o JADE-Leap[32]. De esta forma, los conceptos de las entidades y agente son mapeados desde los modelos fuentes a los modelos destinos, y cada componente del agente es trasladado o cambiado a otro modelo.

- **Paso 3:** Posteriormente, el desarrollador debe aplicar una transformación para convertir los modelos en el código MAS del sistema ubícuo (basado en OSGi). Para realizar esto, debemos usar una transformación PSM-a-código. En este caso se usa *MOFScript* que es un *plugin* del IDE de Eclipse que permite usar plantillas para hacer la traslación. Tal y como se comentó, esas plantillas se han desarrollado para dos plataformas de agentes JADE-Leap y ANDROMEDA y para dos plataformas de VO, THOMAS y E-Institutions.

- **Paso 4:** Finalmente, el proceso termina por añadir los controladores (drivers) necesarios para los diferentes dispositivos utilizados por el *Sistema Ubícuo*. Todas las funcionalidades de los dispositivos físicos son encapsuladas como servicios OSGi, que permite a los agentes usarlos sin preocuparse acerca de las características de bajo nivel. Sin embargo, es necesario proporcionar cada driver/firmware/protocolo de los nuevos dispositivos que no están en la librería de los *bundle* de OSGi. Esta aplicación tiene una librería de *bundles* de OSGi, que almacena los *EnvironmentService* (los servicios de los dispositivos) para su uso frecuente o re-utilización.

4.2. Arquitectura de implantación

Implementar un *sistema ubícuo* es una tarea compleja, ya que no existen todavía sólidos conocimientos básicos sobre cómo implementar esta clase de sistemas. Muchos de los actuales esfuerzos de los investigadores se están enfocando en la implementación de prototipos[77]. Sin embargo, algunos prototipos de *Sistemas Ubícuos* comparten un estilo arquitectónico común, que bien se ajusta a los requisitos de estos sistemas.

Como se discutió anteriormente, los requisitos de este tipo de sistemas son básicamente dos: (i) es esencial que el *sistema ubícuo* soporte la integración de los servicios prestados por los dispositivos externos y por el sistema software; (ii)

el aislamiento de la abstracción de bajo nivel de los dispositivos (la dependencia del fabricante), es decir, los dispositivos del entorno deben ser bien encapsulados en funcionalidades independientes y genéricas. Por lo tanto, un estilo de arquitectura que cumple con estos requisitos, es la bien conocida arquitectura por capas[77]. Por medio de esta arquitectura, los elementos del sistema se organizan en diferentes niveles con responsabilidades bien definidas. Nuestra propuesta sigue este estilo de arquitectura. La Figura 4.3 muestra la plataforma de despliegue (de implantación), que es un marco basado en la tecnología OSGi. Las principales capas de esta plataforma de implantación son:

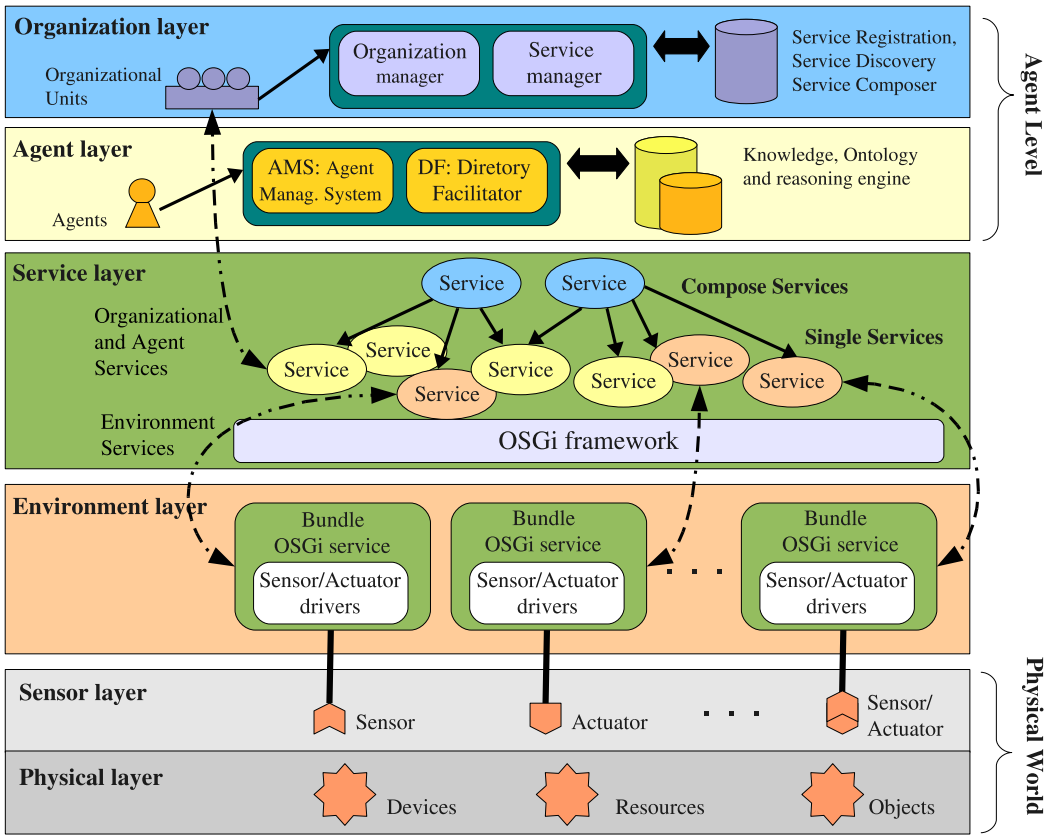

Figura 4.3: Arquitectura propuesta para el sistema ubícuo

Capa Organizacional (*Organizational layer*), es utilizada como un marco normativo para la coordinación, la comunicación y la interacción entre las diferentes entidades computacionales. Esta capa está formada por un conjunto de individuos e instituciones que necesitan para coordinarse los recursos y servicios a través de las fronteras institucionales. Esta capa soporta la interoperabilidad de alto nivel para integrar diversos sistemas de información con el fin de compartir conocimientos y facilitar la colaboración entre las entidades. Esta capa

es un sistema abierto formado por la agrupación y la colaboración de entidades heterogéneas. Desde un punto de vista técnico, estas funcionalidades se obtienen utilizando la plataforma THOMAS[53], que consiste básicamente en un conjunto de servicios modulares que permitan el desarrollo de las organizaciones basadas en agentes en entornos abiertos.

Capa de agente (*Agent layer*), soporta los mecanismos con el fin de registrar, des-registrar y el descubrimiento de los agentes. En esta capa los agentes trabajan juntos a través de diferentes patrones de interacción para soportar tareas complejas de forma colaborativa y dinámica. Esta capa también soporta el manejo de información y conocimiento (*knowledge*) y los modelos de conocimiento necesarios, incluyendo los motores de razonamiento (*reasoning engine*) y las ontologías necesarias por los agentes. Adicionalmente, esta capa proporciona los mecanismos para soportar la comunicación y los lenguajes necesarios usados por los agentes, tales como FIPA ACL.

Capa de servicio (*Service layer*), proporciona las funcionalidades del sistema, ofreciendo los servicios que el *Sistema Ubícuo* debe suministrar. Los servicios son proporcionados por los dispositivos situados en el mundo físico, y por las entidades del MAS (agente o unidad organizativa). Además, en este nivel los servicios pueden ser simples (*single service*) o compuestos (*composite services*), formados por la composición de otros servicios.

Capa de entorno (*Environment layer*), tiene la responsabilidad de encapsular la dependencia tecnológica del fabricante de los dispositivos del entorno. Los controladores ("drivers") que conforman este nivel exportan directamente sus funcionalidades a través de un *bundle*. Los *bundles*, que administran dispositivos similares con firmware de diferentes tecnologías o fabricantes (vendedores), se implementan como una interfaz común, con el fin de proporcionar una manera uniforme de comunicación con los dispositivos del entorno.

Capa sensores (*Sensor layer*), tiene la responsabilidad de acceder a los dispositivos físicos, a través de actuadores y sensores, que permiten cambiar o leer el estado de los dispositivos. Un sensor es un dispositivo capaz de detectar magnitudes físicas, como son las variables del entorno, y transformarlas en valores útiles para un computador. Un sensor también puede decirse que es un dispositivo que convierte una forma de energía en otra. Un actuador es un dispositivo capaz de transformar una señal eléctrica o un valor digital en la activación de un dispositivo con la finalidad de generar un efecto sobre un ambiente. Éste recibe la orden de un controlador (un computador) y en función de ella, genera la orden para activar un elemento final de control como, por ejemplo, una válvula.

Capa física (*Physical layer*), cuenta con los recursos y los objetos que son percibidos en el entorno. Representa completamente el mundo real, donde se encuentra el *Sistema Ubícuo*, del cual se pueden leer el estado de un conjunto de variables y cambiar el valor de algunos parámetros controlables del ambiente.

En las siguientes secciones, se describe con más detalle cada una de estas capas. La descripción comienza desde la capa con mayor nivel de abstracción computacional (la capa organizacional), hasta la capa con menor nivel de abstracción computacional, la capa física que representa el nivel físico o el mundo real.

4.3. Capa Organizacional

En la arquitectura propuesta se incluyen todos los componentes necesarios para cubrir las características y necesidades con el fin de desarrollar soluciones para entornos colaborativos para escenarios como: sistemas ubícuos, organizaciones virtuales, entornos inteligentes, sistemas distribuidos orientados a servicios, etc.

Estos escenarios necesitan de la creación de sistemas dinámicos (de sistemas abiertos que cambian bajo demanda), compuesto por diferentes entidades y dispositivos computacionales que colaboran con el fin de satisfacer las necesidades de los usuarios de forma rápida y fácil. Actualmente, los entornos colaborativos han demostrado un gran potencial en la respuesta a estas necesidades[116, 203]. Un entorno colaborativo que integra diversos componentes computaciones y sistemas de información puede permitir la creación de coaliciones u organizaciones con la finalidad de compartir sus conocimientos y habilidades para dar un conjunto de soluciones parciales con eficacia y eficiencia, que resuelven el problema global.

En los MAS, una de las metas más importantes es la construcción de sistemas que sean capaces de tomar decisiones de forma autónoma y flexible. Sin embargo, para lograr los objetivos sobre estos escenarios, las entidades tienen que dirigir sus iniciativas individuales con la flexibilidad para acoplarse a procesos globales. Un entorno colaborativo debe automatizar los esfuerzos individuales de las entidades y mediar entre ellas para promover la colaboración en el contexto de un proceso global determinado. Para poner en práctica este entorno, se requiere un nuevo paradigma de computación: *computación como interacción*[158]. En este paradigma, la computación es algo que ocurre a través de la comunicación entre las entidades computacionales. Teniendo en cuenta este enfoque, parece natural ver estos sistemas (o escenarios) en términos de los servicios que ofrecen, y por lo tanto, en términos de las entidades, agentes o dispositivos que proporcionan o consumen servicios[47]. Estas entidades o dispositivos pueden haber sido diseñados por diferentes equipos de desarrollo, es decir, tienen una naturaleza muy heterogénea, además pueden entrar o abandonar el entorno en diferentes momentos y por diferentes razones. La tecnología de los MAS, sobre todo de MAS abiertos basado en organizaciones, tienen algunas características que de-

muestran su potencial para soportar estos escenarios[150, 9, 120] (en un entorno cooperativo), para implementar este nuevo paradigma de la computación como interacción.

Un ejemplo de este paradigma, puede ser un *sistema de transporte inteligente*, con autobuses, trenes, tranvías, etc., y paradas, que apoyan y permiten un proceso de colaboración a lo largo del recorrido del pasajero. En tal proceso, cuando se inicializa el proceso de colaboración, el entorno requerirá que el pasajero registre su solicitud de viaje (el servicio de viaje al destino). En la solicitud se requiere la identificación del pasajero y la especificación del destino a donde se quiere desplazar. El servicio entonces se anuncian, y los actores y entidades del sistema colaboraran para satisfacer la demanda. Por ello se ubica el origen del pasajero, gracias a su dispositivo móvil o la ubicación de la parada y se encuentra que unidades de transporte cumplen con su plan de viaje. Así, en este escenario de colaboración y para completar la petición del servicio, las unidades de transporte reportan su ubicación actual aproximada y luego se le indica al pasajero cuales unidades (y el tiempo estimado de llegada) le permiten hacer su viaje. Para proporcionar valor agregado al escenario de colaboración, se establecen múltiples alianzas virtuales entre los actores del sistema para cumplir con la meta de todo el escenario, por ejemplo, si una unidad de transporte sufre una avería, ella reporta tal situación para sacar del escenario esta entidad y solicitar a una unidad nueva que entre al sistema para suplir la falta.

La complejidad de este escenario es alta y por ello, se propone el uso de una *capa organizacional*, ya que es un paradigma que ofrece un claro soporte a los requerimientos de estos problemas. Estos requisitos están influenciados por los siguientes enfoques: la computación es una actividad inherentemente social, los modelos de computación emergen como servicios, una aplicación no es proceso monolítico, los componentes computacionales forman coaliciones u organizaciones virtuales con un comportamiento autónomo y coordinado; entornos de ejecución distribuidos y en diferentes dispositivos.

La *capa organizacional* permite satisfacer todos estos requerimientos, su tecnología proporciona características de interacción entre entidades independientes e inteligentes que pueden adaptarse, coordinarse y organizarse. Actualmente, existen pocas propuesta de MAS que explícitamente soporten organizaciones, la arquitectura propuesta propone el uso de dos enfoques de esta área (pero no excluye a otras propuestas) que se describen a continuación:

- Una *organización virtual* basada en agentes (representada por THOMAS en esta propuesta), es una estructura que se crea dinámicamente mediante la construcción de las diferentes unidades situadas en el escenario de aplicación. Además una Organización Virtual presenta:
    - Un soporte para las funcionalidades de los sistemas abiertos, tales

como entrada y salida de entidades a una determinada organización, que permite el reclutar a nuevos miembros, así como la deserción voluntaria o expulsión de miembros. Este entorno también soporta la creación dinámica, eliminación y modificación de las organizaciones. En otras palabras, se necesitan funcionalidades para permitir que las organizaciones dinámicas existan siempre que se requieren (por ejemplo, un nuevo usuario llega temporalmente al entorno ubícuo, llevando consigo recursos o dispositivos que prestan servicios nuevos, y que necesitan la colaboración de los ya existentes) y que puedan ser borradas cuando no hay necesidad de ejecutarlas (por ejemplo, cuando el usuario sale del entorno ubícuo y ya no presta ningún servicio o recurso).

- Existencia de mecanismos de regulación que garanticen una coordinación eficaz a nivel global en sistemas abiertos, es decir, el cumplimiento de las diferentes políticas de la organización virtual. Estos mecanismos de regulación pueden incluir por ejemplo: las normas de creación, supresión y modificación de roles.
- Soporte para las interacciones orientadas a servicios, con el fin de poner en práctica la interoperabilidad directa con los sistemas basados en la Web, que facilitan (en tiempo de ejecución) la creación, eliminación y modificación de nuevas funcionalidades en las organizaciones virtuales a través de servicios. Este soporte requiere facilidades sintácticas y semánticas para la localización, descubrimiento y composición de servicios.
- La organización virtual (THOMAS) puede ser ejecutada sobre diferentes plataformas MAS. La versión de THOMAS utilizada en este trabajo se ejecuta en una plataforma JADE, que permite hacer un uso confiable de la capa organizacional. Además, se permite la ejecución de agentes externos que pueden ser implementados en diferentes dispositivos para la gestión segura y optimizada de los sistemas multiagente abiertos.

- Una institución electrónica (representada por Electronic Institutions[79]), define qué agentes están autorizados a hacer algún tipo de acción y las consecuencias de estas acciones. Para definir una institución electrónica es necesario definir un lenguaje común, la actividad (lo que se puede hacer dentro de la institución) y las consecuencias que las acciones de los agentes tienen dentro de la institución. Además, una E-Institutions presenta:

  - Una capa que soporta algunas características de la organización. La capa social es la encargada de manejar los aspectos institucionales en

tiempo de ejecución y el cumplimiento de las normas institucionales a los agentes participantes. Para ello, la capa social es la encargada de controlar los roles que los agentes están jugando, las conversaciones en las que están participando y de las obligaciones que cada agente ha adquirido.

- La capa de organización sólo se puede especificar en el tiempo de diseño, es decir, en ejecución la estructura de la organización no puede cambiar. A pesar de esta limitación, la E-Institución puede usarse sin restricciones en organizaciones que tienen características estáticas.
- Los agentes externos interesados en participar en las instituciones de esta plataforma, deben previamente conocer la estructura de la institución, sus normas y el marco de comunicación (ontología, ilocuciones, etc). De esta forma los agentes externos serían capaces de comunicarse de manera eficiente dentro de la institución.
- La plataforma de implementación que soporta el "middleware" social (la parte superior de la plataforma) es JADE, haciendo uso de sus servicios de forma confiable. Además los agentes pueden ser implementados en diferentes dispositivos con una gestión segura.

4.4. Capa de Agentes

La capa de agente es la encargada de suministrar la plataforma de ejecución para los agentes del sistema, además de proporcionar los elementos necesarios para su buen funcionamiento, como por ejemplo, el soporte para la comunicación entre agentes internos y externos a la plataforma. Además, la capa de agente sirve como capa de soporte para la ejecución de la capa organizacional, es decir, los agentes que componen las entidades de la organización se ejecutan en la capa de agente. Por ello, existe la necesidad de la integración del paradigma de sistemas multi-agente y el paradigma de computación orientada al servicio, y no sólo los agentes pueden usar y proveer servicios, sino que en esta propuesta las unidades organizativas también pueden ofrecer y consumir servicios.

Existen diferentes arquitecturas y plataformas de agentes que abordan el problema de la integración de los agentes y servicios. JIKIT[44] es un marco multi-agente que está relacionado con la gestión del conocimiento, en la cual los agentes difunden piezas de conocimiento de información (IK, de sus siglas en inglés Information Knowledge) entre los miembros de una organización. Los agentes de servicio recolectan y detectan las piezas IK relevantes y se las dan a los agentes personales que actúan en nombre de los usuarios de la organización. MASE es otra plataforma que considera información semántica relacionada con

los servicios[164]. MASE es una plataforma que permite la composición dinámica de servicios Web e integra la semántica para enriquecer la descripción de los servicios. Los agentes son las entidades que permiten el soporte ontológico y un motor WS-BPEL distribuye los flujos de trabajo que regulan la ejecución de los servicio. CArtAgO-WS[163] es una extensión de la plataforma de la CArtAgO[171], que integra los servicios, junto con los agentes y el concepto de artefactos. La principal característica de este enfoque es que los servicios Web se insertan (o se ocultan) dentro de los artefactos en el sistema, por lo que sólo cambios menores tienen hacerse para que la plataforma CArtAgO permita implementar estos servicios. PANGEA[205] es otra plataforma que integra agentes y servicios, específicamente servicios Web para prestar las funcionalidades del MAS. La plataforma posee un agente que recibe el nombre de *ServiceAgent*, que es el agente responsable del seguimiento y control de los servicios que se ofrecen. Además, esta nueva plataforma permite el uso de conceptos relativos a las organizaciones, ya que el agente *OrganizationManager* es el responsable de la administración (creación e interacción) de las diferentes sociedades o grupos de agentes.

Aunque existen un buen numero de plataformas de agentes[20], sin embargo esta capa de agente se enfoca y utiliza aquellas plataformas que son lo suficientemente abiertas para permitir la entrada y salida de agentes, que puedan proporcionar soporte para la capa de organizaciones (Organizaciones Virtuales) y que adicionalmente su programación permita fácilmente la implementación (y el uso) de servicios. También es importante recordar como la totalidad de artefactos del entorno (dispositivos físicos y recursos) son encapsulados en servicios OSGi, sería ideal que la plataforma de agentes soporte el uso de *bundles*.

Un ejemplo de estas plataformas base es **JADE**[2][30] utilizada en el desarrollo de agentes compatibles con FIPA, que permite implementar agentes en Java. JADE también es usada como plataforma base para THOMAS y para E-Institutions, y la plataforma se percibe desde fuera como una única entidad, pero se puede dividir en diferentes contenedores de agentes, cada uno ejecutándose en una máquina virtual Java y que puede ser distribuido en diferentes computadores y dispositivos. Cuando la plataforma se pone en marcha, el FIPA AMS (Agent Management System), el DF (Directory Facilitator) y el ACC (Agent Communication Channel) se inician. Cada agente en JADE se ejecuta dentro de un contenedor de agente y debe tener un identificador único global en la plataforma. Uno de los contenedores de agente se llama el contenedor principal, y allí se ubican el AMS y el DF.

JADE-Leap (JADE Light Extensible Agent Platform) es una versión de JADE utilizada para implementar agentes empotrados, que se ejecutan a través de J2ME (Java Micro Edition). Esta capa de agente de la arquitectura de

[2]http://jade.tilab.com/

implantación permite el uso de agentes JADE.

Por otro lado, ANDROMEDA es una plataforma desarrollada en esta investigación que permite la ejecución de agentes sobre el sistema operativo *Android*[3] de Google. ANDROMEDA permite la ejecución de agentes en dispositivos móviles. Los agentes son compatibles con el protocolo FIPA ACL[86] de comunicación, por ello pueden registrase en plataformas (interactuando como agentes externos) que utilicen el protocolo FIPA, como por ejemplo JADE y MAGENTIX2[188]. Así, un agente ANDROMEDA (en esta arquitectura), puede ser un agente externo en JADE (o en otra plataforma) y registrarse en una *Unidad Organizativa* en THOMAS (y/o en E-Institutions), para luego utilizar los servicios de la organización, como también, prestar servicios a otros agentes que así lo soliciten. Lo cual permite una alta flexibilidad en el diseño del sistema.

Finalmente, la arquitectura de implantación propone que la capa de agentes utilice diferentes plataformas, donde los agentes puedan colaborar y trabajar dentro de la organización a través de infraestructura de servicios (SOA), los cuales son solicitados, suministrados y compuestos por los distintos agentes del sistema (o entidades de la organización) para resolver los problemas encontrados en escenarios ubícuos. Por ello, la arquitectura propuesta necesita de una estructura de implementación distribuida y abierta, que permita la participación de plataformas y dispositivos heterogéneos, en la Figura 4.4 se puede observar el enfoque que supone la arquitectura propuesta.

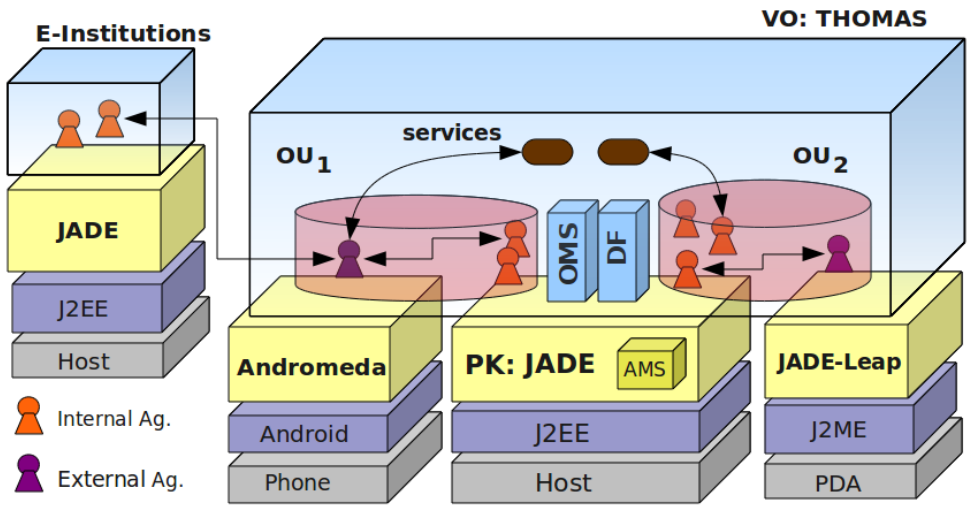

Figura 4.4: Uso de plataformas heterogéneas en la arquitectura

En la Figura 4.4 se observa como los agentes de dos organizaciones diferen-

[3]http://code.google.com/android/

tes, THOMAS y E-Institutions pueden interactuar. Estas organizaciones están en la *capa organizacional* de la arquitectura propuesta y soportan los grupos o sociedades de agentes (por ejemplo, las OUs en THOMAS). Las E-Institutions y THOMAS se ejecutan sobre una plataforma JADE (que se ubican en la *capa de agente* de la arquitectura propuesta). En THOMAS esta plataforma de ejecución corresponde al PK (Platform Kernel).

Existen dos unidades organizacionales, como se observa en la Figura 4.4, que están formadas por agentes de otras plataformas (ANDROMEDA y JADE-Leap). El agente ANDROMEDA está registrado tanto en la E-Institutions como en la OU_1 de THOMAS. Este agente puede interactuar con los agentes THOMAS/JADE y los agentes E-Institutions/JADE. Aun cuando el agente ANDROMEDA, es un agente externo, puede ofrecer y solicitar servicios de las OUs. Los servicios se ubican en la *capa de servicio* de nuestra arquitectura y se describe en la siguiente sección.

La OU_2 está formada con otros agentes externos, con agentes JADE-Leap, e igualmente puede solicitar los servicios de la OU_1. Como la arquitectura propuesta es de naturaleza distribuida, permite agentes, organizaciones y servicios en diferentes lenguajes de programación y con distintos tipos de hardware base. En la Figura 4.4 se observa, que existen cuatro unidades computacionales (hardware): (i) un *smartphone* con *Android* ejecuta los agentes ANDROMEDA, (ii) un PDA con J2ME ejecuta los agentes JADE-Leap, y (iii) dos computadores con J2EE ejecutan los agentes JADE.

4.5. Capa de servicio

Las áreas de la computación orientada a servicios (SOC) y sistemas multi-agentes están comenzando a solaparse. Ambas se aplican a los mismos tipos de escenarios, los cuales se caracterizan por que las tareas deben ser: flexibles, persistentes, distribuidas y con un débil acoplamiento (*loosely-coupled*)[51]. El uso de agentes y organizaciones como entidades prestadoras de servicios no es nuevo, pero el interés emergente de usar estos paradigmas en nuevos escenarios, como los ubicuos, enfoca el uso de los servicios como bloques de construcción de sistemas complejos y reutilizables. Por lo tanto, el esfuerzo principal en la integración de los agentes y servicios está dirigido a enmascarar servicios con el fin de la agregación, integración, o la administración. Con el fin de hacer esto, dos enfoques se han explorado: la integración directa de los servicios y agentes por medio de intercambio de mensajes y la consideración de agentes como *matchmakers* para el descubrimiento y la composición de servicios.

4.5.1. Los servicios Web

La integración y la interoperabilidad de los servicios Web y los agentes han sido estudiadas desde dos puntos de vista: (i) donde existe una entidad intermediaria entre agentes y servicios Web, por ejemplo el trabajo *AgentWeb Gateway*[150], y (ii) donde la idea es proporcionar a los agentes con una interfaz para comunicarse directamente con servicios Web, cuyo un trabajo representativo es *WS2JADE*[146]. Sin embargo, la principal área de investigación ha sido la consideración de agentes como *matchmakers*[191, 190], este enfoque describe acerca del descubrimiento dinámico y la coordinación de los servicios Web semánticos basados en agentes. El *matchmaking* y la intermediación (*brokering*) son los mecanismos de coordinación del sistema multi-agente para los servicios Web. Ambos mecanismos tienen compromisos de desempeño (ventajas y deficiencias), pero el lenguaje OWL-S (de sus siglas en inglés, Web Ontology Language for Semantic Web Services) permite abordar algunas de las deficiencias[118]. Uno de sus principales fortalezas es el protocolo de *brokering* que se basa en dos complejas tareas de razonamiento: el descubrimiento y la mediación[190, 137].

La capa de servicio de la arquitectura propuesta utiliza igualmente este tipo de servicios, y los mecanismos semánticos que permiten componerlos, descubrirlos, etc. Al observar la Figura 4.3 este soporte lo proporciona la capa organizacional. Sin embargo, se debe recalcar que estos mecanismos, como por ejemplo el descubrimiento de servicios es un proceso donde se emparejan (*matching*) los requerimientos de las entidades de la organización ubícua y la descripción del servicio. Para soportar el descubrimiento de servicios (y otros mecanismos) en la arquitectura, los servicios se expresan en OWL-S[133] (soportado por THOMAS), que brinda tres componentes que potencian la descripción semántica de los servicios[137]. Estos tres componentes son los siguientes (ver la Figura 4.5):

- Perfil de servicio (*Service Profile*), que describe principalmente la función de servicio, es decir, la entrada del servicio, la salida, pre-condiciones y efectos, y también incluye el nombre del servicio, el proveedor del servicio (ver Figura 4.6). El proceso de búsqueda de servicio implementa servicio de *matching* a través del perfil de servicio, para descubrir los servicios Web que satisfacen al solicitante del servicio.

- Modelo de Servicio (*Service Model*), describe principalmente los detalles de cómo se implementa el servicio, es decir, el cómo hacerlo.

- *Service Grounding*, que describe principalmente cómo acceder a servicios, incluyendo el protocolo de comunicación, etc.

Cada servicio anunciado o publicitado es una instancia directa del perfil del servicio. Por lo tanto, el descubrimiento de servicios Web basado en el perfil

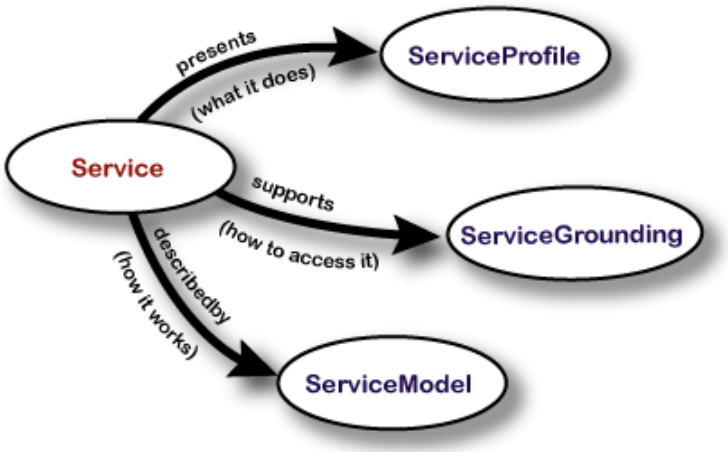

Figura 4.5: Descripción de un servicio con OWL-S

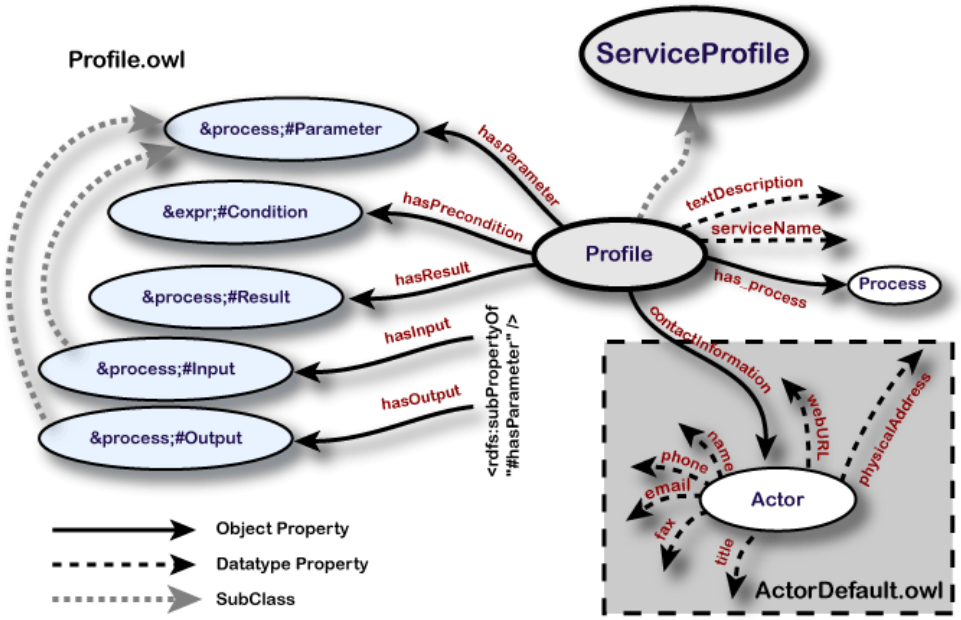

Figura 4.6: Perfil de servicio en OWL-S

de servicio incluye expresiones de la instancia del perfil y el *matching* de los requerimientos de la entidad y la publicidad del servicio. Además se puede lograr una descripción semántica jerárquica, que se expresa por la lógica de OWL-S, la cual está basada en árboles de ontologías, y con esto el *matching* del servicio es en realidad el *matching* de los árboles de ontología. Los principales objetivos de alto nivel para utilizar OWL-S en esta capa son:

1. Proporcionar un marco de representación de propósito general para describir los servicios Web.

2. Soportar la automatización de la administración de los servicios y que puedan ser usados por los agentes.

3. Ser lo suficientemente amplio como para soportar todo el ciclo de vida de las tareas de los servicios.

De esta forma, la descripción de alto nivel de los servicios en esta capa, permite a los agentes cambiar y adaptar la estructura y la funcionalidad del sistema, permite que los sistemas sean flexibles y altamente dinámicos a través del descubrimiento y la composición de servicios, creando un mecanismo de acuerdos que soporta la gestión y negociación automatizada y la reconfiguración del sistema de forma autonómica. Los servicios se utilizan, y no son propietarios, es decir, no se integran físicamente en las entidades o sistemas que los utilizan y se ejecutan en la infraestructura de servicio de esta capa (la infraestructura que permite la ejecución de los servicios en esta capa es una plataforma OSGi, tal y como se explicará a continuación).

Resumiendo, la descripción de alto nivel de los servicios (los servicios Web) añade características únicas que permiten abordar sistemas con los siguientes requisitos:

- En sistemas intrínsecamente distribuidos, y con diferentes configuraciones de implementación.

- En sistemas donde se puedan cambiar de forma independiente los servicios equivalentes el uno del otro, sin que esto tenga repercusiones en su funcionalidad y con ello garantizar la robustez a fallos.

- En sistemas que implementan comportamientos de adaptación al entorno, ya sea mediante la sustitución de los servicios individuales o añadir otros nuevos, o cambiando su configuraciones.

- La operación del sistema es compartida entre los diferentes grupos de interés, y por lo tanto su funcionalidad requiere de la coordinación de estos grupos de interés.

4.5.2. Servicios OSGi: marco de implementación

Como la capa de servicio se ejecuta sobre un marco OSGi (muchos de los servicios Web, al igual que muchos de los agentes dentro de la organización), este marco OSGi está basado igualmente en servicios. OSGi (de sus siglas en inglés

Open Services Gateway Initiative) es un organismo independiente sin ánimo de lucro que trabaja para definir y promover especificaciones abiertas para la prestación de servicios administrados en entornos de red. Estas especificaciones definen un marco de Servicios OSGi, el cual consta de dos partes: el marco OSGi y un conjunto de definiciones de servicios estándar. El marco OSGi, que se encuentra en la parte superior de una máquina virtual de Java, es el entorno de ejecución de servicios. Por otra parte, el marco de ejecución se puede dividir en dos elementos principales: (i) una plataforma de servicios, y (ii) una infraestructura de implementación[60].

- La *plataforma de servicios* se define como una plataforma software que soporta la interacción orientada a servicio representada en la Figura 4.7. Esta interacción involucra tres actores principales: los proveedores de servicios, los solicitantes de servicios y un directorio de servicios, aunque sólo el directorio de servicios pertenece a la plataforma de servicios. En la interacción orientada a servicio, los proveedores de servicios publican (*publish*) sus descripciones de los servicios, y los solicitantes de servicios descubren (*discover*) los servicios y se unen (*bind*) a los proveedores de servicios. La publicación y descubrimiento se basan en una descripción de servicio (*service description*).

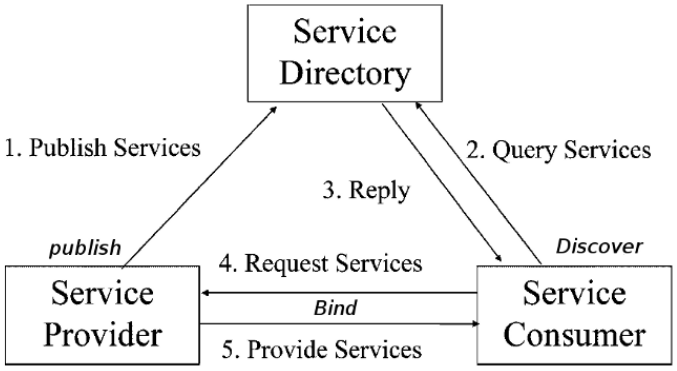

Figura 4.7: Interacción orientada a servicio de OSGi

En el contexto de OSGi, un servicio se describe como una clase Java o interfaz, la interfaz de servicio, junto con un número variable de atributos y las propiedades del servicio que son pares atributo y valor. Las propiedades del servicio permiten a los diferentes proveedores del mismo, a prestar los servicios con una interfaz común. El registro de servicio permite a los proveedores de servicios a que puedan ser descubiertos a través de

las búsquedas formuladas al directorio de servicio. Además, los mecanismos de notificación permiten a los solicitantes de servicios recibir eventos que señalan cambios en el directorio de servicios. Estos cambios incluyen la publicación o la recuperación de un servicio particular.

- La *infraestructura de implementación* se realiza de acuerdo a una serie bien definida de estados representados en la Figura 4.8[100]; estos estados corresponden con el ciclo de vida del *bundle* (módulo, paquete o *deployment* OSGi). El *bundle* se puede instalar dinámicamente, puede ser iniciado, ser detenido, actualizado y desinstalado. Durante la ejecución, el *bundle* realiza la transición entre los estados, la cual es controlada a través del API de gestión de módulos OSGi. A continuación se describen los estados del *bundle*:

  1. Instalado (INSTALLED): el *bundle* se ha instalado correctamente,
  2. Desinstalado (UNINSTALLED): el *bundle* se ha desinstalado correctamente y ya no existe en la plataforma.
  3. Resuelto (RESOLVED): se ha instalado el *bundle*, y sus dependencias han sido satisfechas. Este estado indica que el paquete ya está listo para ser iniciado o se ha detenido.
  4. Iniciado (STARTING): un estado temporal que el *bundle* pasa mientras comienza a ejecutarse.
  5. Detenido (STOPPING): un estado temporal que el *bundle* pasa mientras termina de ejecutarse.
  6. Activo (ACTIVE): el *bundle* está ejecutándose.

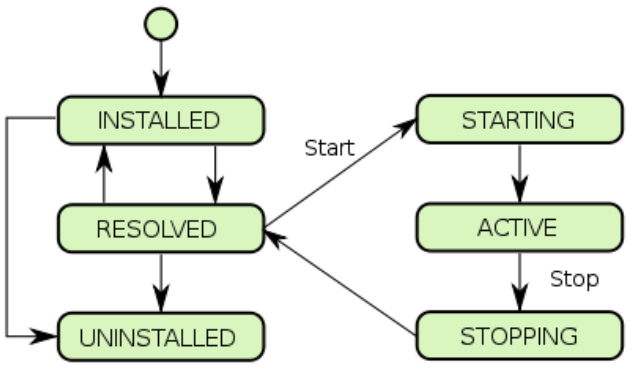

Figura 4.8: Ciclo de vida del Bundle

La plataforma de servicios de OSGi incluye todos los mecanismos necesarios (*administración de la dependencia de servicios)* para crear los enlaces (*binding*) entre los *bundles* lógicos durante el tiempo de ejecución (ver Figura 4.9). Esto incluye la publicación de servicios, el descubrimiento, el enlace (*binding*) y la adaptación con respecto a los cambios, debido a la disponibilidad dinámica (la llegada o salida) de los servicios que están siendo utilizados por un *bundle* durante la ejecución. La administración de la dependencia de servicios es la clave para la creación de aplicaciones para el marco OSGi. A diferencia de las dependencias de código, las dependencias de servicio no están garantizadas, o gestionadas por el marco, lo que significa que un *bundle* físico puede ser activado, incluso si los servicios requeridos por el *bundle* lógico no están disponibles.

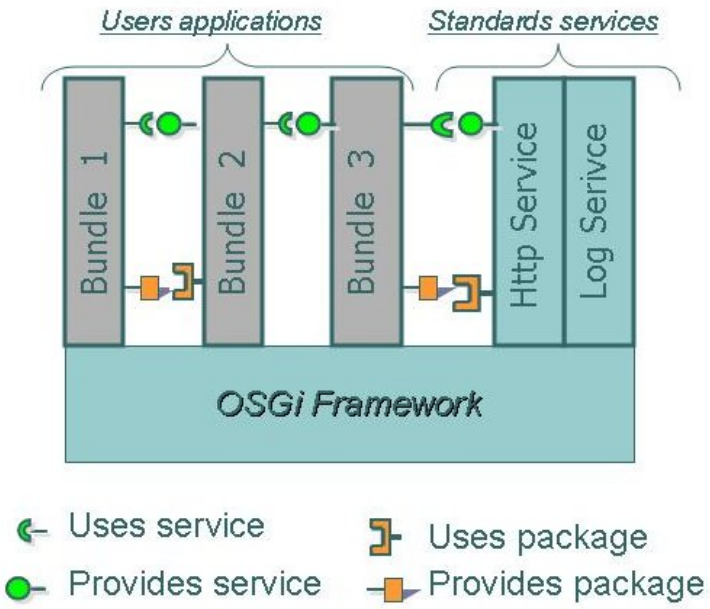

Figura 4.9: Los servicios de los *bundles* sobre el marco OSGi

La Figura 4.10 representa una aplicación típica OSGi, modelada como una serie de *bundles* lógicos que están conectados (*binding*) a través de proveedores y solicitantes de servicio. Los *bundles* de sensores (o actuadores) están conectados a sensores físicos tales como detectores de humedad, medidores de temperatura, relés, etc. Un *bundle* monitor pregunta a los sensores y responde a situaciones particulares. Por ejemplo, el monitor puede utilizar los servicios proporcionados por un servidor HTTP para registrar un *servlet*. De esta forma, la lectura o el control se puede realizar de forma remota a través de una página Web. En el ejemplo, el *bundle* lógico del monitor utiliza los servicios proporcionados por los

otros *bundles* lógicos para realizar su tarea. En este caso, el *bundle* del monitor se encarga de descubrir los servicios que requiere. Por otro lado, los *bundles* lógicos que proporcionan servicios son responsables de la publicación en el directorio de servicios. En la Figura 4.10 se observa que el *bundle* monitor posee más de una interfaz para solicitar servicios. Esto es posible ya que la plataforma de ejecución proporciona un mecanismo conocido como *componentes de servicio*, en la que múltiples servicios (como proveedor o solicitante) pueden ser implantados dentro de un *bundle* físico, por lo que un sólo *bundle* puede conectarse (*binding*) con múltiples *bundles* para crear la aplicación.

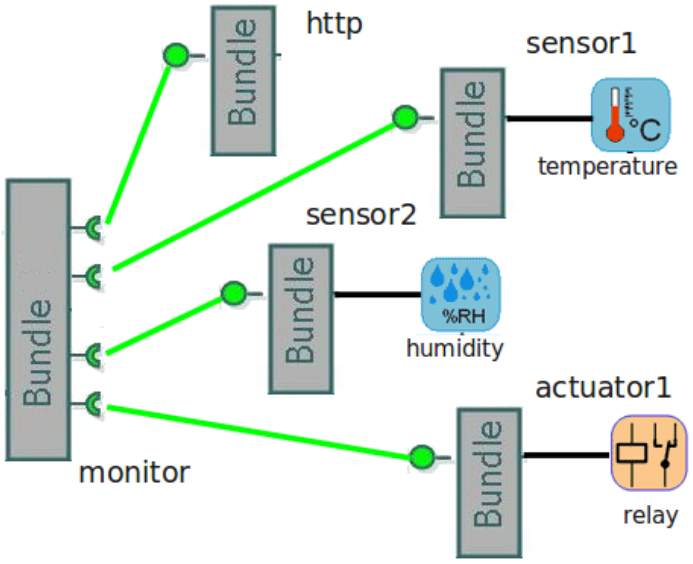

Figura 4.10: Una aplicación OSGi

Durante la ejecución, los diferentes servicios presentan una dinámica a medida que se introducen o se retiran del entorno de ejecución, como consecuencia de las acciones de implantación de los *bundles* físicos. Se puede suponer, por ejemplo, que los nuevos *bundles* con sensores pueden ser introducidos en el entorno de ejecución en cualquier momento, sin la necesidad de reiniciar la arquitectura. Los *bundles* lógicos son responsables del enlace a los servicios recién llegados o de la liberación de los servicios que salen. La disponibilidad dinámica de los servicios permite que una aplicación OSGi sea capaz de ensamblarse y adaptarse dinámicamente. Lo cual es muy importante para el tipo de sistema propuesto en nuestra arquitectura.

Es bueno resaltar que la dinámica de este enfoque en la arquitectura presentada no es exclusiva de los servicios, si no que muchos agentes (principalmente los internos a la organización) también la poseen, ya que estos agentes se im-

plementan como un *bundle* en el marco OSGi, por lo que pueden ser cargados o expulsados (o que finalizan su ejecución) de la organización de forma transparente, sin afectar el funcionamiento de la plataforma MAS.

4.5.2.1. OSGi distribuido

El plan inicial de OSGi es desarrollar aplicaciones en domótica (con pasarelas domésticas). Las aplicaciones implementadas por lo general están diseñadas de acuerdo con el paradigma cliente-servidor, con un modelo central de servidor[101]. Como resultado de ello, sólo hay una plataforma OSGi en las arquitecturas propuestas, de modo que todos los servicios y las tareas de cálculo se administran en la pasarela de una manera centralizada, y todos los otros dispositivos de información sólo pueden ser controlados por la pasarela doméstica en lugar de estar funcionando de forma autónoma (hay el riesgo de que colapse todo el sistema al fallar el único punto de administración). Sin embargo, cada día habrá más y más dispositivos compatibles con las especificaciones OSGi, además de la pasarela del hogar en el futuro, tales como teléfonos celulares, televisores, automóviles y otros[100].

Algunas investigaciones [85, 170, 135, 203] han señalado que una arquitectura distribuida es superior al tradicional paradigma centrado en el servidor, ya que la arquitectura centrada en el servidor no puede sacar el máximo provecho de las capacidades de los dispositivos de computación ubícua y por lo tanto, no es adecuada para entornos ubícuos. Además, las principales SOA tienen un estilo de computación distribuida que ayuda a las entidades a compartir la lógica y los datos entre múltiples aplicaciones.

Sin embargo, los marcos OSGi distribuidos han sido estudiados en sólo unos pocos enfoques hasta ahora. Uno de las principales propuestas es el proyecto R-OSGi[170] que permite varios marcos OSGi través de una capa de *middleware*. Se ofrece acceso remoto a los servicios OSGi, así como la notificación de eventos a distancia. En trabajos como el de Vilas et al.[196] se proporciona una solución a este problema basada en la tecnología de servicios Web. En trabajos del perfil de dispositivos para servicios Web (DPWS, de sus siglas en inglés Devices Profile for Web Services)[85] mejora el enfoque anterior, mediante la estandarización del proceso de consumo y la exposición de servicios Web con una huella ligera. Los objetivos de DPWS es utilizarlos en dispositivos con recursos limitados, de modo que se puede aplicar a una variedad de dispositivos empotrados ampliamente utilizados en los hogares y empresas. En el trabajo de Wu et al.[203] la distribución se alcanza por medio de una combinación de servicios Web, con agente móviles.

La distribución es un componente crucial en nuestra arquitectura, pero la especificación OSGi no define explícitamente mecanismos de comunicación estándar entre marcos OSGi (inter-OSGi), lo que provoca que aplicaciones imple-

mentadas (servicios OSGi) en cada marco individual OSGi a aislarse a través de diferentes marcos. Aunque uno de los enfoques más completo para resolver este problema es el proyecto R-OSGi, como la arquitectura de implantación ya posee los mecanismos para la integración de servicios Web, como la composición, descubrimiento, etc., es lógico pensar que esta infraestructura pueda ser usada para la distribución de los marcos OSGi en nuestra arquitectura, a continuación se describe este proceso

4.5.2.2. Interacción inter-OSGi con servicios Web

La evolución de la tecnología del hogar (de nuestros espacios de vida) hace prever que en futuro aparecerán cada vez más dispositivos basados en OSGi[100], adicional a las pasarelas domesticas, como teléfonos celulares, televisores y automóviles. En consecuencia, en lugar de la arquitectura cliente-servidor con modelo central, se propone en esta arquitectura utilizar modelo OSGi distribuido con una interacción P2P (peer-to-peer)[203] para los dispositivos y entidades inteligentes como se muestra en la Figura 4.11. La plataforma OSGi aparece en varios dispositivos, distribuyendo así los servicios que dependen de los dispositivos sobre varios marcos. La carga del marco tradicional (un solo marco, la pasarela doméstica) se reduce debido a la implementación distribuida de los *bundles*, y estos *bundles* locales dependientes de los dispositivos aun pueden funcionar normalmente si se bloquea el marco tradicional (la pasarela doméstica).

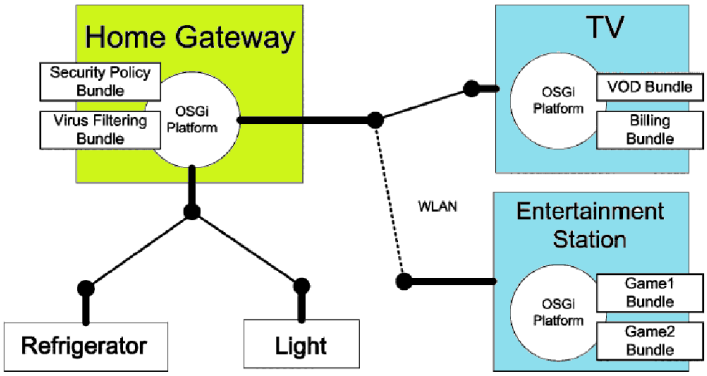

Figura 4.11: Modelo P2P de las plataformas OSGi

El principal problema de este modelo P2P, son los mecanismos de comunicación o el protocolo entre las plataformas OSGi. El enfoque orientado a servicios que se propone para hacer frente a este problema es que, en cada plataforma OSGi se implanta un *bundle* de puerta de enlace basado en servicios Web. De esta manera, cada plataforma OSGi puede publicar sus servicios, lo que hace que

otras plataformas OSGi sean conscientes y capaces de utilizar dichos servicios. Cada servicio OSGi que sí se registra en este *bundle* es capaz de interactuar con otros servicios Web a través del estándar de servicios Web propuesto anteriormente.

El mayor beneficio en utilizar los servicios Web estándar es que proporciona interoperabilidad entre plataformas de OSGi y no-OSGi. Por ejemplo, en la Figura 4.12, la plataforma OSGi puede publicar sus servicios locales como servicios Web públicos a través del *bundle* de servicios Web, y los servicios Web que son publicados pueden ser consumidos por otras plataformas OSGi o J2EE. Sin embargo existe una limitación, los *bundles* en una plataforma pueden acceder normalmente a otros *bundles* locales (ver Figura 4.13), pero en plataformas remotas sólo se puede hacer con servicios Web. Esta limitación no es crítica en la plataforma propuesta donde se proporciona a los agentes servicios de alto nivel, es decir servicios Web.

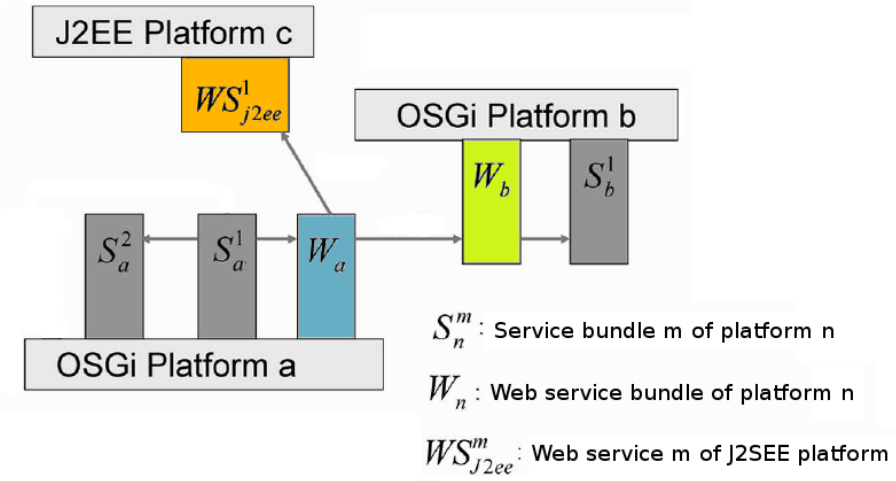

Figura 4.12: Interacción entre plataformas OSGi a través de servicios Web

Finalmente, es importante destacar que el OSGi es un marco que puede soportar la ejecución de servicios de alto nivel (servicios Web), de entidades activas (como los agentes), y de servicios de bajo nivel (los servicios de los dispositivos físicos: sensores y actuadores). Sin embargo, los servicios OSGi de los dispositivos generalmente no se acceden directamente desde la capa de agentes o la organizacional, si no que se crea un segundo servicio (generalmente un servicio Web, aunque no es excluyente, puede ser otro servicio OSGi) de un nivel más alto de abstracción que llama al servicio OSGi del dispositivo. El objetivo de crear un segundo servicio es proporcionar a las entidades de la organización una interfaz común al servicio, con una descripción semántica de alto nivel que pueda ser ad-

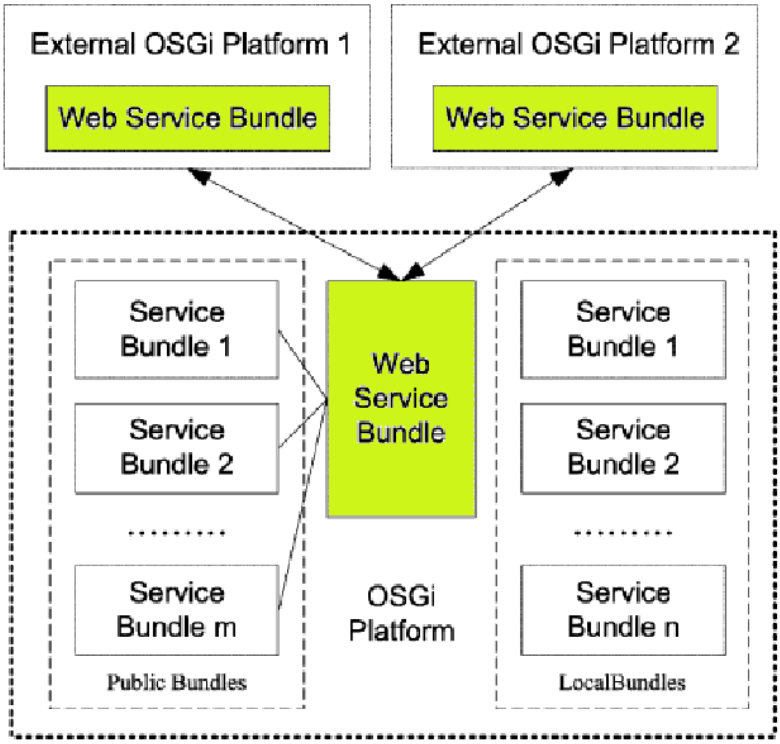

Figura 4.13: Plataformas OSGi conectadas por servicios Web

ministrada automáticamente por los agentes de la organización. Esta interacción entre estos dos servicios se describe con más detalle a continuación en la capa de entorno.

4.6. Capa de entorno

Esta capa tiene como objetivo acceder a los artefactos del mundo real para leer variables del entorno (a través de sensores, como por ejemplo detectores de humo, detectores de presencia, el nivel de líquido en un tanque, etc.) o accionar dispositivos para efectuar tareas de control sobre el entorno (escribir en el entorno con actuadores, como por ejemplo cerrar una válvula, bajar unas persianas, encender una luz, cambiar la temperatura de la calefacción).

En la capa de entorno cada sensor/actuador tiene una frontera de acción (un espacio de operación limitado) dentro del entorno ubícuo, "capturando" los objetos que están unidos a él. La capa de entorno puede comunicarse con una amplia variedad de dispositivos, aparatos, sensores y actuadores, y representa

para las capas superiores de la arquitectura un middleware uniforme para acceder al mundo físico. La capa de entorno convierte efectivamente cualquier sensor o actuador de la capa física a una entidad de software que suministra un conjunto o un flujo de datos.

Sin embargo, para cumplir con los requerimientos antes mencionados, la capa de entorno debe ocultar las características de bajo nivel de los dispositivos, ya que depende de la tecnología de cada fabricante. Existen muchos casos, que un mismo sensor o actuador fabricado por diferentes compañías posee distintos parámetros de configuración, con distintas unidades de medida, y con protocolos de acceso propietarios, a pesar que el dispositivo adquiere una misma variable física. Debido a la diversidad de dispositivos en el mercado y que todos los sistemas con entornos ubícuos usan un conjunto muy heterogéneos de dispositivos, es necesario buscar una forma de solucionar este problema.

Es por ello, que la mayoría de fabricantes cuando comercializan sensores o actuadores, vienen acompañados de los *drivers* o *firmware* propietarios, que permiten ponerlo en funcionamiento. Sin embargo, esta es una solución parcial, ya que el funcionamiento y uso del *driver* de los dispositivos aún posee características del fabricante, como por ejemplo protocolos propietarios y no estándar de acceso.

Una solución más general es realizar un "wrapper" o un encapsulado de los driver propietarios y que este encapsulado proporcione un interfaz abierta y estándar para su uso. Nuestra propuesta es usar un servicio OSGi para acceder a los dispositivos, a través del encapsulamiento del *driver* del fabricante usando un *bundle*[103] (como se observa en la capa de entorno de la Figura 4.3). Esta implementación proporciona una interfaz común (la de un servicio OSGi), con el fin de que la comunicación sea de una manera uniforme cuando se accede a dispositivos del entorno. Esto supone que dos sensores de fabricantes diferentes una vez encapsulados en el *bundle*, tienen una misma interfaz de acceso.

Una vez que el *driver* del dispositivo está encapsulado como un servicio OSGi, ya tenemos todas las funcionalidades de una plataforma basada en servicios. El objetivo de la arquitectura, es que los agentes puedan usar los artefactos del entorno de forma automática y por ejemplo pueda combinar o componer nuevos servicios dinámicamente. Por ello, el *bundle* con el driver del dispositivo (*Device bundle*) se exporta a un nuevo servicio de alto nivel (*Device Web service*). Un servicio Web que tiene descripción semántica y sintáctica para que los agentes y entidades de la organización puedan "razonar" con él. Este enfoque se muestra en la Figura 4.14.

Cada *bundle* se describe a continuación:

- *Device bundle*: módulo del dispositivo

    1. Este *bundle* está solamente conectado a los dispositivos y es respon-

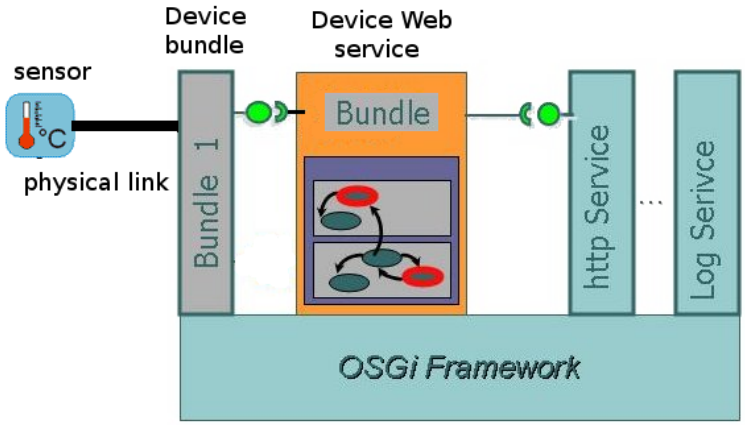

Figura 4.14: Uso de los dispositivos como un servicios Web

sable de la comunicación directa con los dispositivos. Por lo general, será un *bundle* dependiente del dispositivo, cuya tarea principal será controlar el dispositivo, para supervisar su estado. Además, puede cooperar con otros *bundles*, como por ejemplo, recibir las solicitudes de servicio y/o reportarle el estado del dispositivo. También puede hacer una monitorización del estado del entorno y transformar los fenómenos observados en datos crudos o en datos sin procesar.

2. Este *bundle* se registrará en el directorio de servicios para proporcionar la información sobre cómo otros *bundles* pueden interactuar con él, y con ello recibirá las solicitudes de servicio de otros componentes. Por ejemplo, los datos "crudos" (las variables del entorno sin procesar) se pueden enviar a un agente para que realice un proceso de inferencia del contexto, aplicando algún algoritmo inteligente sobre estos datos.
3. Los artefactos controlados por este *bundle* se pueden clasificar en dos categorías: los dispositivos y los recursos. La diferencia entre estos dos tipos de artefactos es que este último puede ser controlado para servir a los usuarios, mientras que el primero sólo se puede utilizar para recoger datos de fenómenos físicos o controlar una variable del entorno.

- *Device Web service*: servicio Web del dispositivo.

1. Es un *bundle* que usualmente es independiente del dispositivo, el cual consta de funcionalidades básicas predefinidas. Realiza una administración o un control directo de alto nivel del *Device bundle* y transforma los datos en bruto en datos semánticos de alto nivel, que son

mucho más significativo para los agentes y entidades de la organización (o para los seres humanos).

2. Este servicio Web se registra a sí mismo en el DF de la organización para proporcionar información de interacción y recibir las solicitudes de servicio de otras entidades. A través del directorio DF, cada agente puede recuperar esta información sobre los dispositivos existentes en el entorno, y por lo tanto puede saber cómo interactuar con ellos para realizar servicios específicos. Los servicios y datos con alto nivel semántico permiten a los agente realizar funciones de inferencia sobre estos.

Es importante señalar que la plataforma de servicio OSGi, se ejecuta normalmente sobre Java, sin embargo, como ya se ha mencionado una posible plataforma de ejecución de agentes es ANDROMEDA, que está basado en *Android*. Este sistema operativo posee una máquina virtual muy similar a Java estándar, aunque posee algunas variaciones para hacerla más eficiente para dispositivos móviles. Al ser un sistema que permite la ejecución de programas Java, es posible ejecutar el marco OSGi dentro del dispositivo móvil, como lo muestra la Figura 4.15

Figura 4.15: Plataforma OSGi sobre Android

Sin embargo, el Bytecode que utiliza *Android* difiere del Bytecode del clásico Java de ORACLE, por ello para ejecutar la plataforma OSGi, como por ejemplo

la distribución APACHE Felix[4] (que es la usada en la arquitectura propuesta), debemos hacer un proceso de conversión, tanto de la plataforma como de los *bundles*. Este proceso se ilustra en la Figura 4.16. El proceso consiste en aplicar algunas herramientas del SDK de *Android* para traducir el Bytecode. Una vez compilado el *bundle* con el Java ORACLE, se obtiene un fichero del tipo `jar` (por ejemplo, `Bundle.jar`). Este fichero se procesa con la herramienta (comando) `dx`, que genera un fichero del tipo `dex` con un Bytecode compatible con *Android*. Posteriormente, se reemplaza y se inserta el Bytecode compatible dentro del fichero jar (en `Bundle.jar`) utilizando la herramienta `aapt`. Con esto el fichero `Bundle.jar` es un *bundle* compatible *Android*, que ya puede ser copiado dentro del dispositivo, con la herramienta `adb`.

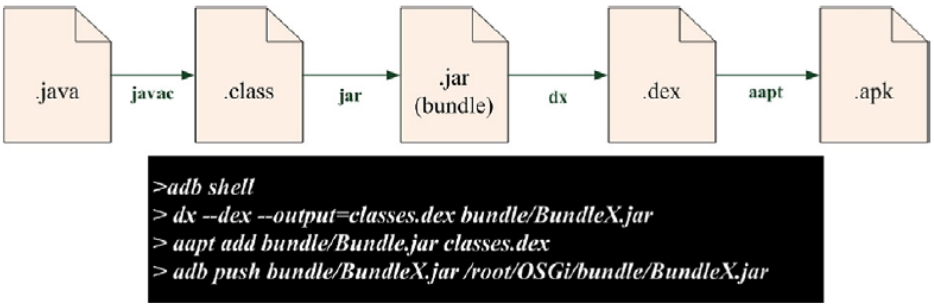

Figura 4.16: Proceso de conversión de los *bundles* para ejecutarse sobre Android

Aunque, *Android* soporta el marco OSGi, es importante comentar que en muchos casos los sensores se pueden conectar y encapsular directamente con los componentes nativos en *Android*. Estas tecnologías (o plataformas) tienen algunas características comunes[58], (que se resumen en la Figura 4.17). Utilizando únicamente *Android*, se pueden enmascarar los parámetros y funcionalidades de bajo nivel de los sensores, ya sea usando su API Java o el NDK (de sus siglas en inglés, Native Development Kit) que permite compilar y utilizar código C++ en aplicaciones *Android*. Sin embargo, para proporcionar un enfoque uniforme y homogéneo en la arquitectura esta característica no es utilizada.

Finalmente, en la siguiente sección, se describe la última capa de la arquitectura, la capa física donde se ubican todos los objetos reales y físicos: sensores, hardware, artefactos, etc.

[4]http://felix.apache.org/

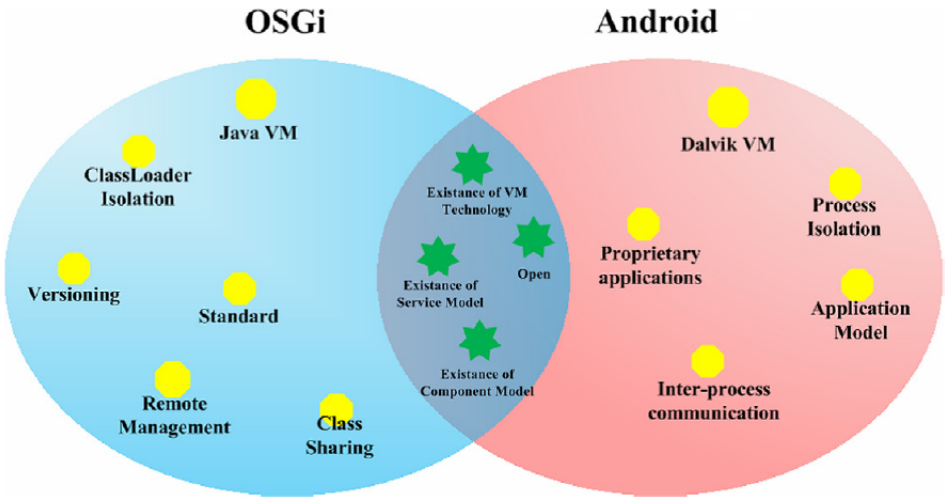

Figura 4.17: Comparación entre OSGi y Android

4.7. Capa física y de sensores

Esta capa se compone de los diversos artefactos, dispositivos (sensores y actuadores) y aparatos que utilizan las entidades de la arquitectura. Muchos de éstos se encuentran en un típico entorno ubícuo. Por ejemplo en el hogar encontramos lámparas, persianas automáticas, cámaras IP, electrodomésticos, radios despertadores y timbres, etc. En otros escenarios como el de transporte urbano observamos, detectores de presencia, LCDs interactivas, pantallas, teléfonos inteligentes, balizas de detección, GPS, brújulas, etc. Incluso una típica alarma que monitoriza la seguridad, puede ser parte de la capa física, igualmente sensores y actuadores, tales como detectores de humo, RFID (siglas de Radio Frequency IDentification) activas y pasivas, aires acondicionados, sistemas de calefacción, termostatos, detectores de movimiento y humedad, etc. Además, esta capa puede incluir cualquier objeto que cumple un papel importante en un espacio, como en el hogar, lo es una silla o mesa y en el transporte urbano, una señal de tránsito, un semáforo, dispositivos para el cobro del pasaje, etc.

Cuando se utilizan los sensores/actuadores, en esta capa, debemos clasificar dos métodos para el empleo de los mismos:

1. El enfoque clásico propone que uno o varios sensores se conecten a través de su interfaces estándar, tales como USB, RS232, RS485, RS422, I^2C, SPI, lazos de 4-20 mA, etc, o interfaces de datos industriales, tales como ASI, BITBUS, ProfiBus, FieldBus, etc., a un equipo con capacidad computacional (un *host*). Es decir, el sensor/actuador tiene un rol pasivo y es

controlado por el host a través de su *driver*.

2. El enfoque basado en redes Ethernet, en el cual el sensor cuenta con una interfaz estándar de redes (RJ45 o WIFI) para conectarlo en redes LAN o a Internet, y así las acciones sobre sensor-actuador (lectura, escritura y configuración) se realizan como una página/servicio Web o con protocolo propietario sobre HTTP.

Los agentes incorporan estos dispositivos y artefactos a través de los servicios. Los agentes pueden usar las funcionalidades de los componentes físicos llamando a los servicios ofertados por el dispositivo. En primer lugar, las funcionalidades del dispositivo se exportan a la capa del entorno usando *Bundles* OSGi. Posteriormente, el servicio OSGi del dispositivo puede ser invocado por los agentes y otras entidades de la VO en la capa de servicio. No importa como se accede al dispositivo, lo importante es *encapsular* las características y funcionalidades de bajo nivel (ya sea del *driver* o del protocolo HTTP propietario o del servicio Web básico propietario), en un servicio Web de alto nivel de abstracción que puedan usar los agentes, por medio de su descripción semántica.

En el entorno existen otros componentes que forman parte del sistema como por ejemplo, los equipos de redes: antenas, routers, pasarelas, concentradores (*hubs*), etc. Sin embargo, estos no se describen, ya que se explican principalmente los equipos con capacidades de computación que permiten la ejecución de los programas y los sensores-actuadores que permiten convertir fenómenos físicos en datos y viceversa. A continuación se describen con más detalle los principales componentes de hardware que utiliza la arquitectura.

4.7.1. Computadores empotrados

Este es uno de los componentes de hardware más importante a resaltar, es el computador empotrado que puede estar incrustado en el entorno sin que los usuarios noten su presencia. En el entorno doméstico pueden estar ubicados en los diferentes ambientes del hogar, como sala, cocina, comedor, habitaciones, etc., y se encargan de recoger la señal de los sensores y de proporcionar capacidad de cómputo y ejecución de programas. En entornos urbanos como en el trasporte inteligente, se pueden encontrar ubicados en las paradas, puntos de control/información, en balizas de detección, dentro de los vehículos de transportes: autobuses, taxis, metros, tranvías, etc.

Un computador embebido o computador empotrado, en su forma clásica, es un sistema de computación diseñado para realizar una o algunas funciones dedicadas. Al contrario de lo que ocurre con los ordenadores de propósito general (como por ejemplo una computadora personal) que están diseñados para cubrir

un amplio rango de necesidades, los sistemas empotrados se diseñan para cubrir necesidades específicas. Sin embargo, con el avance de la electrónica los sistemas empotrados cada día tienen más capacidades y características similares a las PCs. Aunque una característica que mantienen es que un sistema empotrado la mayoría de los componentes se encuentran incluidos en la placa base.

Actualmente, en el mercado están disponibles algunos computadores empotrados, con diferentes características según su propósito. Pero, entre esos computadores podemos mencionar a la tarjeta electrónica conocida como *Beagleboard*[5], como uno de los hardwares más populares y con buenas prestaciones para implementar sistemas empotrados[94, 41], en la Figura 4.18 se puede apreciar la tarjeta.

Figura 4.18: Tarjeta Beagleboard

La Beagleboard es un computador de baja potencia empotrado, de tarjeta única basado en un procesador de la familia OMAP de Texas Instruments. Entre las principales especificaciones de la Beagleboard, para la revisión C3 (usada en nuestra investigación), tenemos que: Es una tarjeta que posee un procesador OMAP3530 - 600MHz (system-on-a-chip ARM Cortex A-8 core), y su tamaño es tipo "bolsillo" (de 76 mm), con funcionalidades de un computador básico. Incluye un DSP TMS320C64x+ a 520MHz para la aceleración de vídeo y codificación de audio. El GPU PowerVR SGX 2D/3D, que permite renderizado en 2D y 3D, con soporte OpenGL ES 2.0. Tiene dos salida de vídeo, analógica en un conector de S-Video y digital sobre HDMI. Posee un slot para tarjeta SD/MMC, que emula un

[5]http://www.beagleboard.org/

disco duro. Posee 256MB de memoria RAM y 256MB de memoria NAND flash. Un puerto USB OTG (mini AB) y uno USB. Puertos RS232 y JTAG. Salida y entrada de audio en estéreo, y una fuente de alimentación es de 5 voltios.

Las características de la Beagleboard se pueden resumir en la Figura 4.19, que muestra los componentes principales, así como sus posibles periféricos de expansión.

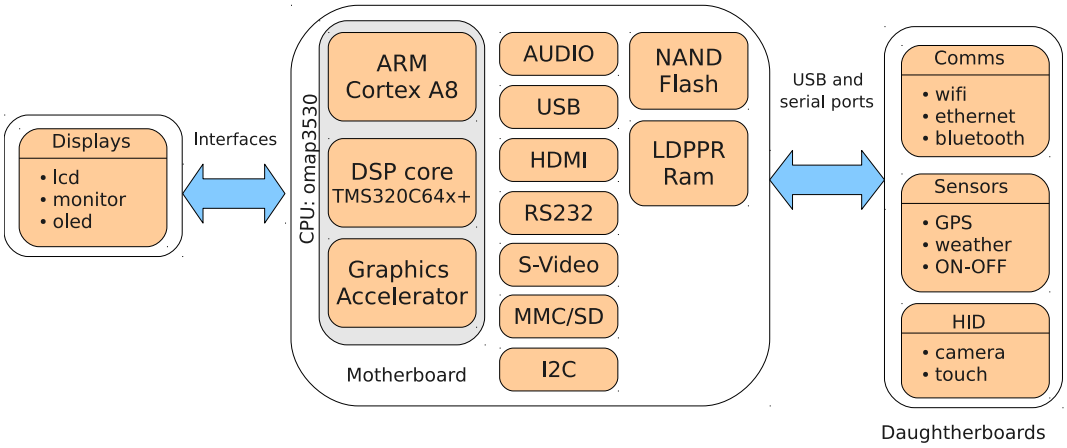

Figura 4.19: Principales componentes de hardware de Beagleboard y sus expansiones

Beagleboard es el computador empotrado donde se conectan los sensores y actuadores en nuestra propuesta, utilizando el enfoque clásico, donde los dispositivos tienen algunas o ninguna propiedades activas y necesitan de un *host* para su funcionamiento: esto es la lectura, escritura y configuración de los dispositivos. Además la Beagleboard ejecuta a los agentes empotrados, ubicados sobre el entorno ubícuo, por ejemplo en el hogar. Los agentes que ejecuta son los desarrollados, en nuestro caso, sobre la plataforma ANDROMEDA.

Debido a esto, se tuvo que portar *Android* para la Beagleboard. Este proceso de instalar *Android* sobre la Beagleboard fue posible gracias a que esté se ejecuta sobre una capa Linux. Esto se logró modificando, actualizando (y compilando) el kernel y el sistema de archivos para la Beagleboard a partir de trabajos previos desarrollados por los grupos: EMBINUX[6] y 0xLAB[7] con su proyecto 0xdroid. Así se obtuvo que *Android* fuese compatible con todos los componentes electrónicos que están sobre la Beagleboard y otros periféricos externos necesarios para la comunicación inalámbrica, como adaptadores USB para Bluetooth, Ethernet y WIFI. Para resumir lo anterior, la Figura 4.20 muestra la arquitectura de software que da soporte a *Android* sobre Beagleboard y como consecuencia permite

[6]http://labs.embinux.org/
[7]http://0xlab.org/

ejecutar agentes ANDROMEDA de forma similar a como se realiza en un teléfono o tableta basados en *Android*.

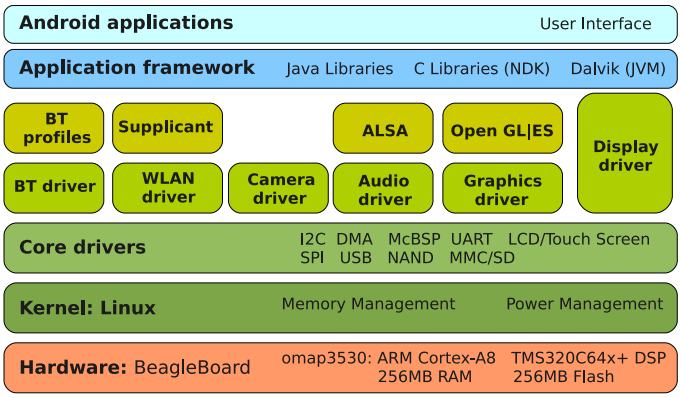

Figura 4.20: Arquitectura Android sobre Beagleboard

Es bueno destacar que después de los trabajos inícials del grupo 0xLAB, la empresa *Texas Instruments* oficialmente impulsó que *Android* se ejecutara eficientemente en su familia de procesadores *OMAP* y *SITARA* que tienen un núcleo ARM Cortex-A8 (que es la base de la Beagleboard y otras dispositivos empotrados), y por ello lanzó el proyecto ROWBOAT[8] [9], que actualmente es una de la versiones de *Android* más completa y estable para la familia de procesadores de *Texas Instruments*.

Finalmente, debido al interés de ejecutar Linux sobre procesadores con arquitectura basada en ARM, se creó LINARO[10] una empresa sin ánimo de lucro que distribuye versiones de *Android* para algunos computadores o tarjetas empotradas de gama alta.

4.7.2. Dispositivos móviles: teléfonos, tabletas, netbooks y PDAs

También debemos mencionar que otro hardware de importancia a incorporar en nuestra propuesta son los terminales móviles, que tienen la capacidad de ejecutar los agentes empotrados de los clientes. Estos terminales pueden ser principalmente: teléfonos, tabletas, netbooks y PDAs.

- Los teléfonos se suponen que pueden existir de dos tipos: el primero corresponde a teléfonos basados en el sistema Android, tales como HTC Desire

[8] http://www.ti.com/lsds/ti/tools-software/android.page
[9] https://code.google.com/p/rowboat/
[10] http://www.linaro.org/

o HTC One S (usados en los escenarios), y que ejecutan agentes sobre ANDROMEDA. El segundo teléfono son los dispositivos que tengan incrustado J2ME y por ello permiten ejecutar los agentes del tipo JADE-Leap.

- Las tabletas basadas en *Android*, y sobre ellas podemos instalar ANDROMEDA.

- Netbooks, con soporte Java permite la ejecución de agentes JADE. Sin embargo, en este tipo de ordenador portátil, por ejemplo el netbook modelo LG-X110, se le instaló *Android*, (y posteriormente ANDROMEDA), a partir de la modificación del trabajo hecho por el grupo Android-x86[11]. Este es un *Android* modificado para ejecutarse sobre los procesadores de la gama intel-X86 (específicamente Intel Atom), ya que originalmente el sistema operativo se ejecuta en procesadores con arquitectura del tipo ARM-V5 o superior.

- PDAs con una máquina virtual J2ME, y con esto permite ejecutar agentes basados en JADE-Leap (en desuso).

Todos estos terminales móviles poseen al menos un agente personal. Con este agente el usuario puede interactuar con los agentes empotrados en la Beagleboard, para usar servicios de entorno que prestan los dispositivos físicos, es decir los servicios ubícuos. Así como también con otros servicios Web ofrecidos por la organización virtual dentro de la plataforma global.

4.7.3. Sensores, actuadores y otros componentes

Los sensores y actuadores, a utilizar dependerán del escenario de uso. Los dispositivos dependen de si la aplicación es doméstica, es decir, el entorno inteligente está en el hogar o es un entorno industrial o urbano, etc. En el hogar, los dispositivos son comúnmente: persianas automáticas, lámparas inteligentes, detectores de presencia, de humo, humedad, controles de puertas y ventanas. Muchos de estos dispositivos se conectan al computador empotrado, la Beagleboard, o pueden trabajar independiente del *host*, con un protocolo de red.

Existen también dispositivos más adaptados a escenarios industriales o a aplicaciones urbanas (en el exterior). Para los escenarios industriales podemos contar con medidores de un sin número de variables físicas producidas por procesos industriales, tales como, presión, flujo o caudal, nivel de líquidos, el PH de sustancias, concentraciones, control sobre válvulas, etc. En entornos urbanos podemos medir las condiciones climáticas, el volumen vehicular (del tránsito), condiciones

[11] http://www.android-x86.org/

de peligro como atascos en el tránsito o usar cámaras para la seguridad o monitorizar a la población. También es posible controlar señales de tránsito, semáforos de peatones y vehículos, avisos de información y publicidad, etc.

Muchos de estos sensores están disponibles comercialmente, otras mediciones se pueden obtener mediante la composición de servicios entre varios dispositivos. Una línea de sensores o dispositivos importante a resaltar son los llamados Phidgets[12]. Los Phidgets son un sistema de componentes electrónicos de bajo costo y sensores que pueden ser controlados por un computador personal, utilizando una interfaz USB estándar, y esta interfaz en base para todos los Phidgets. Las aplicaciones se administran por el API del fabricante y pueden ser desarrolladas en variados sistemas operativos de escritorio e incluso móviles.

Su uso está enfocado principalmente para permitir la exploración de los fenómenos físicos a través de la interacción por el puerto USB con un host, y ha sido adoptado en algunos trabajos sobre AmI[102, 120, 103, 95], ya que simplifican enormemente la interacción. Los Phidgets son un intento de construir una analogía física de los *widgets* de software, lo que permite la construcción de sistemas físicos complejos a partir de componentes más simples. En la Figura 4.21 se muestra un sensor infrarrojo para temperatura del tipo Phidgets.

Figura 4.21: Sensor tipo Phidgets

Para finalizar, en la Figura 4.22 se observa en un esquema cómo los dispositivos (sensores) se conectan a las interfaces de la Beagleboard para su lectura y configuración. Los datos proporcionados por los sensores se encapsulan en un servicio (bajo nivel), que posteriormente se enlaza con un servicio Web semántico, que está registrado en el DF de THOMAS para que los agentes lo puedan descubrir y más tarde solicitar. De esta forma, cualquier tipo de dispositivo puede ser incorporado a nuestra propuesta de forma flexible y adaptable. La arquitectura presentada soporta la evolución de los dispositivos, los cambios, mejoras y

[12]http://www.phidgets.com/

actualizaciones, que puedan ocurrir con el transcurso del tiempo. La propuesta tiene características muy deseables cuando se desarrollan sistemas ubícuos.

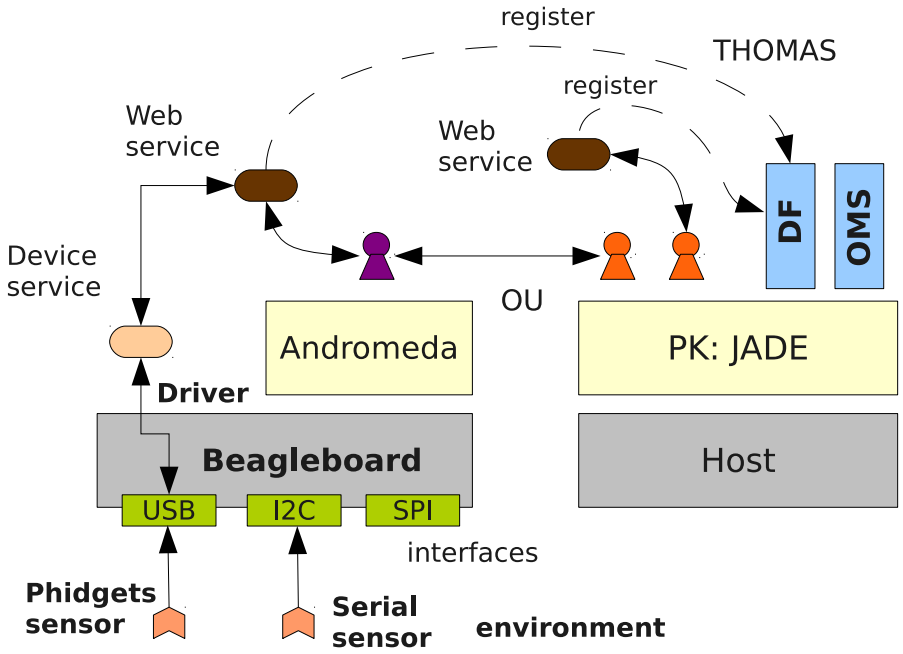

Figura 4.22: Dispositivos en la Beagleboard y su uso como servicio

4.8. Conclusiones

Este capítulo propone una arquitectura de soporte para la implantación de una *Organización Virtual Ubícua*. En él se han descrito los problemas de este tipo de sistemas y se han identificado los principales requisitos, y en respuesta a estos, se presenta una arquitectura abstracta para el desarrollo de *Organizaciones Virtuales*. Esta arquitectura permite la ejecución de una *Organización Virtual Ubícua* desarrollada usando las ideas propuestas por el desarrollo MDD. Dicho desarrollo, permite al desarrollador realizar un diseño más intuitivo usando estos métodos de "model-driven" que usando los enfoques clásicos, ya que se modela el *Sistema Ubícuo* como un modelo UML evitando los detalles técnicos.

Esta arquitectura se enfoca en dar soporte a los requerimientos para la implementación de sistemas multi-agente abiertos. Su objetivo es lograr la integración de los paradigmas de sistemas multi-agentes, y de los sistemas basados en servicios, la base para crear una *Organización Virtual* aplicada en los sistemas ubícuos. La aplicación a escenarios ubícuos en general, propone superar un conjunto de

grandes desafíos y que estos desafíos se acentúan por la apertura del entorno. La arquitectura propuesta permite soportar:

- El dinamismo debido a la disponibilidad del dispositivo, el contexto del dispositivo, la ubicación y la actividad del usuario.
- La distribución debido a la ubicación natural de dispositivos en el entorno.
- La heterogeneidad debido a la variedad de hardware, software y protocolos que depende de la evolución de los dispositivos creados por el mercado.
- Las restricciones empotradas derivadas de dispositivo de bajo costo elaborado por el mercado de la electrónica de consumo.

La arquitectura soporta que los servicios que ofrecen las entidades (unidades organizacionales, agentes y dispositivos), se enfoque en una computación con características inherentemente sociales, es decir, que se espera la colaboración de diferentes entidades para proporcionar uno o varios servicios. Estos servicios pueden ser verdaderamente complejos, ya que se generan de la composición de diferentes servicios Web, o de simples servicios como los que proporcionan generalmente los dispositivos físicos (encapsulado en un servicio OSGi). Este enfoque modela las aplicaciones como una composición de servicios de alto nivel, mientras que la implementación de cada entidad sigue el enfoque de componentes.

El enfoque de componentes se utiliza en un modo de integración de pre-facto, donde todos los componentes tienen que obedecer el mismo modelo de componentes. Por otro lado, se utiliza la orientación de servicio en un modo de integración post-facto, en la que cada entidad de software es integrada (combinada con otras) tal como es, con su propio comportamiento, para crear la funcionalidad de la aplicación. El uso de este enfoque proporciona a la arquitectura de una gran modularidad en las aplicaciones y en las entidades que participan en toda la aplicación.

Finalmente, los aportes de esta arquitectura en el desarrollo de *Sistemas Ubícuos* se pueden resumir en los siguientes:

- El *Sistema Ubícuo* se puede adaptar a nuevos dispositivos tecnológicos de una manera fácil, ya que el enfoque utiliza un alto nivel de abstracción que no es tan dependiente del hardware.
- Acoplamiento débil (*loose coupling*), los proveedores y los clientes sólo comparten una interfaz de servicio, donde los clientes son independientes de las implementaciones de los proveedores.
- Se dispone de la abstracción necesaria para que una entidad de terceros (externa) sea considera como una composición de servicios, lo que permite una

administración independiente y separada (donde pueden existir entidades con distintos ciclo de vida).

- La implementación del *Sistema Ubícuo* sobre una plataforma basada en servicios (como OSGi) usando ingeniería de software basada en agentes, permite que los agentes administren los servicios, el contexto, la adaptación del entorno, dando la posibilidad de crear modelos más avanzados y potentes.

Por consiguiente, la arquitectura presentada tiene la abstracción necesaria para la composición de dispositivos con características heterogéneas y la flexibilidad necesaria para que las aplicaciones sigan las características dinámicas de los sistemas ubícuos.

En el siguiente capítulo se presenta ANDROMEDA, una plataforma que permite ejecutar agentes sobre dispositivos basados en el sistema operativo Android (agentes en teléfonos móviles, tabletas y en computadores empotrados), que fue desarrollada durante el transcurso de esta Tesis, como un soporte adicional de la arquitectura de implantación para la ejecución de agentes empotrados. La arquitectura de implantación permite usar diferentes plataformas de implementación, como por ejemplo: Jade, Jade-Leap, THOMAS y E-Institutions. ANDROMEDA es una de las plataformas de implementación de la arquitectura propuesta, que ejecuta directamente el código de los agentes modelados en πVOM sobre dispositivos que soporten Android. La motivación de la plataforma es la ejecución de agentes empotrados con características ubícuas, que puedan acceder a su entorno usando dispositivos físicos.

5

Andromeda

Índice

5.1. Introducción

El diseño e implementación de MAS, es una área con variadas metodologías, arquitecturas, plataformas, lenguajes y escenarios de aplicación[200, 110, 172]. Sin embargo, cuando se necesita abordar un escenario, utilizando MAS, que integran diversas tecnologías y actores: como diferentes sensores, capacidades computacionales con recursos limitados (computadores empotrados), diferentes medios de comunicación inalámbricos o cableados, entidades que interactúan con los humanos en el entorno, etc. (como los escenarios que plantean los Ambientes

Inteligentes), entonces es necesario combinar varios enfoques, plataformas y herramientas para MAS. Este hecho trae complicaciones para el desarrollador, ya que no existe una forma uniforme de abordar este problema. Por ello se desarrolló ANDROMEDA, con la idea de contar con una plataforma de ejecución que pueda utilizarse en los escenarios mencionados anteriormente. ANDROMEDA puede ser vista como un soporte para la ejecución de agentes ubícuos en dispositivos con el sistema operativo *Android*.

Otra motivación para crear nuestra propia plataforma se debe a la gran diferencia encontrada entre los modelos conceptuales de agentes y su implementación en los sistemas empotrados. Por ejemplo, es conocido que Java es un lenguaje muy utilizado para desarrollar agentes, pero las diferencias entre Java para computadores personales (J2SE) y Java para dispositivos empotrados (J2ME), produce grandes cambios en la implementación del agente que es frecuentemente resuelto por la superposición o creación de nuevas capas de intermediación, reduciendo la funcionalidad del agente en muchas plataformas[64]. Con la aparición de *Android* como plataforma para el desarrollo de aplicaciones empotradas, se creó una nueva propuesta para diseñar agentes empotrados inteligentes, ya que *Android* es una plataforma abierta (*Open Source*) y la nueva librería de Java (Java *Android* library) es muy similar o más cercana al Java para computadores personales (J2SE) [17]. Además, *Android* posee ciertas características que facilitan su migración a dispositivos diferentes a los teléfonos móviles o tabletas (como ya ha ocurrido, ya que está basado en Linux), y con ello los agentes podrían estar ejecutándose en una variada gama de artefactos, tales como TV inteligentes (*Smart TV*), tarjetas empotradas, centros multimedios, GPS para vehículos, etc.

Por ello, en este capítulo se describe la plataforma de ejecución ANDROMEDA, sus componentes más importantes, su arquitectura y se muestra un ejemplo de un agente en esta plataforma. También se realiza una breve introducción de *Android*, se mencionan cuáles son sus principales atributos y características más relevantes.

5.2. Andromeda: plataforma de ejecución de agentes en Android

ANDROMEDA (ANDROid eMbeddED Agent platform)[2, 1, 3] es una plataforma de agentes específicamente orientada a agentes empotrados sobre el sistema operativo *Android*. *Android* puede ser visto como un sistema de software diseñado específicamente para dispositivos móviles que incluye un sistema operativo, un *middleware* y sus correspondientes bibliotecas. Las aplicaciones se escriben usando el lenguaje de programación Java y se ejecutan en Dalvik (la máquina

virtual de *Android*), una máquina virtual especialmente diseñada para el uso en entornos empotrados, que se ejecuta sobre un kernel Linux. La plataforma ANDROMEDA incluye todos los conceptos abstractos del meta-modelo de agente en π*VOM*. La implementación se realizó usando los principales componentes del API de *Android* (con el SDK 1.6 como base). Los agentes se ejecutan como cualquier aplicación de *Android*. En las siguientes secciones se realiza una breve introducción de *Android*, se mencionan cuáles son sus principales atributos y características más relevantes.

5.2.1. *Android* y su arquitectura

Android es un sistema de software para dispositivos móviles que incluye un sistema operativo, una capa de intermediación, bibliotecas y aplicaciones. Los desarrolladores pueden crear aplicaciones para la plataforma utilizando el SDK de *Android*. Las principales capas del sistema operativo[17] son mostradas en la Figura 5.1 donde se observan sus diferentes componentes, los cuales son explicados brevemente a continuación:

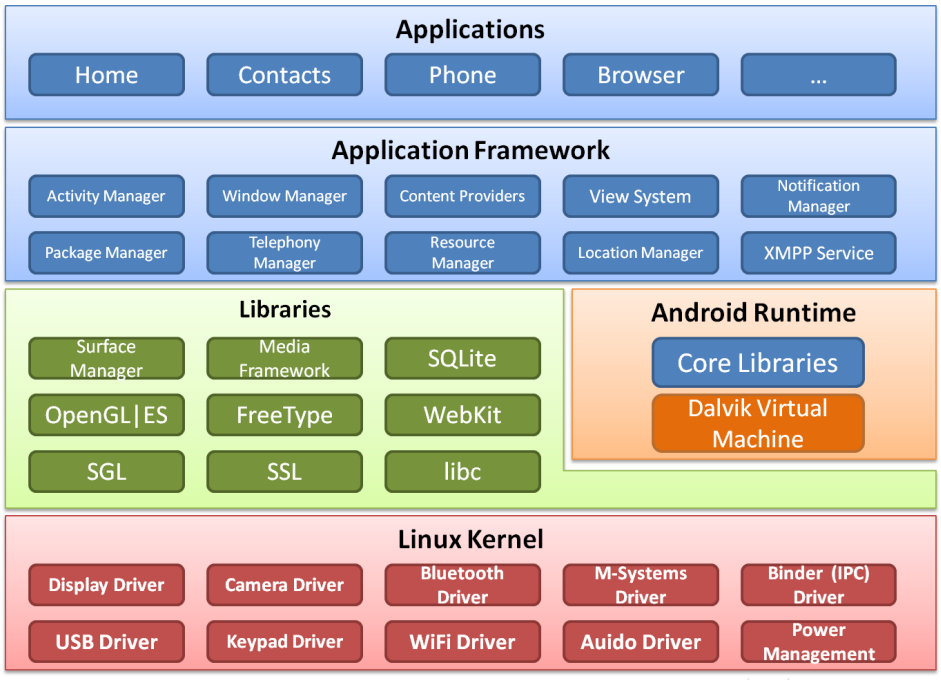

Figura 5.1: Arquitectura del sistema *Android*

- **Aplicaciones:** *Android* se suministra con un conjunto de aplicaciones básicas que incluye un cliente de correo electrónico, programa SMS, calendario,

mapas, navegador, libreta de contactos y otros. Todas las aplicaciones son escritas usando como lenguaje de programación Java. Cada aplicación de *Android* se ejecuta en su propio proceso, con su propia instancia de la máquina virtual Dalvik. Dalvik ha sido creada para que un dispositivo pueda ejecutar múltiples máquinas virtuales de una manera eficiente.

- **Infraestructura de Aplicaciones:** Los desarrolladores tienen total acceso al mismo API usado por las aplicaciones principales del sistema. La arquitectura de aplicaciones está diseñada para simplificar la reutilización de componentes; cualquier aplicación puede publicitar sus capacidades y cualquier otra aplicación puede hacer uso de esas capacidades (sujeto a las restricciones de seguridad de la plataforma).

- **Bibliotecas:** *Android* incluye un conjunto de bibliotecas utilizadas por los diversos componentes del sistema *Android*. Por ejemplo, algunas de las bibliotecas principales dan soporte a la reproducción y la grabación de populares formatos de audio y vídeo, también incluye un núcleo del motor del navegador Web y motor SQL (SQLite) para el mantenimiento de bases de datos.

- ***Android* Runtime:** *Android* incluye un conjunto básico de las bibliotecas que proporciona la mayor parte de la funcionalidad del lenguaje de programación Java. *Android* Runtime proporciona un conjunto de componentes abstractos para crear aplicaciones.

- **Núcleo Linux:** *Android* se basa en Linux, versión 2.6, como núcleo del sistema y da soporte o servicios tales como seguridad, gestión de memoria, la gestión de procesos, la pila de red y el conjunto de manejadores (*drivers*). El núcleo también actúa como una capa de abstracción entre el hardware y el resto de la pila de software.

5.2.2. Componentes de *Android*

Hay cuatro bloques constructores o bloques de componentes en una aplicación *Android*: la actividad (**Activity**), el receptor de intenciones (**BroadcastReceiver**), el servicio (**Service**) y el proveedor de contenidos (**Content Provider**). Una aplicación no necesita usar todos los componentes a la vez, pero pueden ser combinados de cualquier forma para desarrollar una aplicación. Cada aplicación tiene un fichero `AndroidManifest.xml`, llamado fichero de manifiesto, el cual lista todos los componentes usados en la aplicación. Éste es un fichero XML en donde se declaran los componentes, sus capacidades y sus requisitos.

- **Actividad:** Es el más común de los cuatro bloques constructores de *Android*. Una *Actividad* es usualmente un simple proceso en una aplicación. La *Actividad* es una pantalla que sirve como interfaz al usuario y que está compuesta por diferentes visores (**Views**) los cuales responden a eventos. Cada *Actividad* se implementa como una simple clase que extiende de la clase base `Activity`.

- **Receptor de Intenciones:** es un manejador de eventos, es decir, permite definir la reacción de la aplicación a los eventos (llamados *Intents*), por ejemplo, cuando suene el teléfono, cuando la red de datos se encuentra disponible o cuando es medianoche. Los *Receptores de Intenciones* no se muestran como una interfaz de usuario UI (User Interface), aunque se pueden utilizar las notificaciones para alertar al usuario si algo interesante sucede. La aplicación no tiene que estar en ejecución para que su *Receptor de intención* sea llamado; el sistema iniciará la aplicación, si es necesario, cuando el *Receptor de Intenciones* sea disparado.

- **Servicio:** Un servicio es un código de larga-vida que se ejecuta sin una UI. Es un proceso ejecutándose en *background* sin interactuar con el usuario por un periodo indeterminado de tiempo. Un buen ejemplo de este proceso es una aplicación que sirva como un reproductor multimedia, la reproducción de música en sí no debería ser manejada por una actividad porque el usuario espera seguir oyendo la música incluso después de desplazarse a una nueva pantalla. En este caso, un servicio permanecerá en funcionamiento para mantener la reproducción de la música.

- **Proveedor de contenidos:** Las aplicaciones pueden almacenar sus datos en ficheros, en una base de datos o en cualquier otro mecanismo de persistencia. El *Proveedor de Contenidos* es útil para compartir datos con otras aplicaciones *Android*. El *Proveedor de Contenidos* es una clase que implementa un conjunto de métodos estandarizados para permitir que varias aplicaciones puedan almacenar y recuperar datos que son manejados por el *Proveedor de contenidos*.

5.3. Componentes de Andromeda sobre Android

Está sección describe cómo se implementa un agente ANDROMEDA sobre el API de Android. Se describen como se mapean los conceptos del modelo de agente propuesto (*Agent*, *Behaviour*, *Capability*, *Task*, etc.) sobre los componentes principales de Android (*Activity*, *Service*, *BroadcastReceiver*, *Intent*, etc.), para

crear una plataforma de implementación particular de agentes en la arquitectura de implantación descrita en el capítulo anterior (capítulo 4).

ANDROMEDA (ANDROid eMbeddED Agent platform) es una plataforma de agentes empotrados desarrollada en el marco de este trabajo que implementa el modelo de agente πVOM en el sistema operativo conocido como *Android*. El objetivo es contar con una plataforma de agentes empotrados propia que permita ejecutar agentes que se basan con el modelo propuesto y también porque *Android* promete ser un nuevo sistema que permitirá implementar agentes con mayores capacidades debido a la nueva librería desarrollada por Google, y por el hecho que es un sistema abierto y puede migrarse fácilmente a otro tipos de dispositivos.

La plataforma presentada nos permite crear agentes con los requerimientos y necesidades del usuario. Los agentes se ejecutan como cualquiera aplicación del sistema *Android*, ya que la plataforma ANDROMEDA puede interpretarse como una nueva capa que es insertada en la arquitectura del sistema operativo *Android*, como se observa en la Figura 5.2, que es una modificación de la arquitectura original que es mostrada anteriormente en la Figura 5.1.

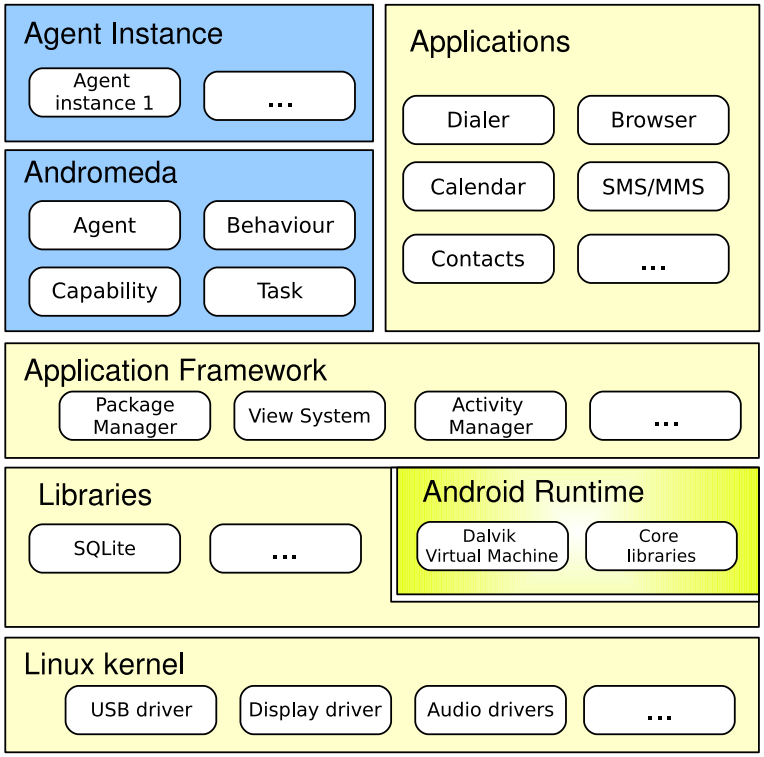

Figura 5.2: ANDROMEDA en la arquitectura de *Android*

ANDROMEDA implementa todos los conceptos abstractos del meta-modelo de agente πVOM. El desarrollo del agente fue realizado utilizando los principales

componentes del API de *Android* (la versión 1.6), y estos se muestran en la Tabla 5.1 que permiten la implementación del meta-modelo. En las secciones siguientes se describen estos componentes desarrollados.

Tabla 5.1: Los componentes de *Android* usados en el agente πVOM

Agente π VOM	Clases *Android*	Métodos sobrecargados
Agent	Activity + Service	onCreate(), onStart(), onDestroy()
Behaviour	BroadcastReceiver	registerReceiver(), onReceive()
Capability	BroadcastReceiver	registerReceiver(), onReceive()
Task	Service	onCreate(), onStart(), onDestroy()
Events	Intents	IntentFilter()
Believes	Contentprovider	–
Comunicación	HTTP	–

5.3.1. Implementando el agente Andromeda en Android

Un agente se implementa en ANDROMEDA como una instancia de la clase `Agent`. La clase `Agent` está diseñada para manejar fundamentalmente la llegada de eventos y soportar los otros componentes del meta-modelo. Por lo tanto, un agente tiene que considerar los cambios en su entorno (puede ser de interés para el agente) para determinar las acciones futuras activando y desactivando los *Comportamientos* (*Behaviours*) para responder a cualquier situación interna o externa. La clase `Agent` está implementada como una *Activity* en *Android*, y una vez que el agente se inicia, éste lanza una *Activity* que será la pantalla de interfaz con el usuario.

Para implementar el modelo de agente algunos métodos han sido sobrecargados de la clase `Activity`. El método `onCreate()` que se ejecuta al arrancar *Activity* permite inicializar las variables del agente. Posteriormente, el método `onStart()` es ejecutado a continuación por el *Activity* y ésto se aprovecha para habilitar los diferentes componentes del agente y al llegar a este punto el agente estará ejecutándose normalmente. El agente se ejecutará hasta que el usuario decide detenerlo. Para ello el agente llamará al método `selfstop()`, permitiendo terminar la ejecución del agente. Con ello cada componente del agente será detenido o eliminado.

El concepto agente diseñado tiene diferentes métodos que permiten al usuario implementar y ejecutar el modelo de agente sobre *Android*, pero existen dos métodos que son importantes de mencionar: el método `init()`, donde el usuario debe escribir el código necesario para iniciar el agente y el método `addbehav()`, que permite adicionar los distintos *Comportamientos* que son los diferentes roles

que el agente puede jugar. Debemos recordar que el agente debe lanzar una *Actividad* para interactuar con el usuario y mostrar el estado interno del agente y sus progresos. Los principales métodos de la clase agente en ANDROMEDA son los mostrados en la Figura 5.3.

```
public class Agent extends Activity {
    private AID myAID;
    private Goals mygoals;
    private List<Behaviour> myListBehaviours;
    . . .
    public void init()
    private void run()
    public boolean changestate(Behaviour behaviour, boolean cond)
    public void addbehav(Behaviour myBehaviour)
    public void destroy()
    protected void agentDestroy()
    . . .
}
```

Figura 5.3: Principales métodos de la clase *Agent*

Finalmente, el ciclo de ejecución básico de un agente en ANDROMEDA está orientado a la gestión de los eventos que se produzcan. Así un agente debe tener en cuenta la información de entrada para determinar sus futuras acciones, activando los mecanismos más adecuados para responder frente a cualquier situación que se produzca como consecuencia fundamentalmente de la lectura de su cola de eventos. Ésto puede ser ilustrado en el algoritmo de la Figura 5.4. Del código podemos interpretar que el agente siempre se mantiene a la espera de eventos que puedan activar o desactivar los *Comportamientos*, cambios en los *Objetivos* (*Goals*) o eventos que puedan disparar alguna nueva *Capacidad*.

Pero es oportuno hacer la salvedad que este código solo "ilustra" ese ciclo, ya que la recepción de eventos en *Android* la realizan los *Receptores de Intenciones* y estos funcionan de manera asíncrona, es decir, que no hace falta tener explícitamente un ciclo para verificar los eventos, ya que, al producirse alguno, el sistema operativo notifica automáticamente que se está produciendo un evento y este es re-direccionado al *Receptor de intenciones* correspondiente.

Las funciones utilizadas en el ciclo de vida de un agente en ANDROMEDA son las siguientes (ver Figura 5.4):

1. `UpdateBeliefs(environment)`, actualiza las creencias del agente a través de las estructuras que modelen el acceso a datos externos del entorno.

2. `ChangedGoals(behav)`, examina si ha habido algún cambio en los objetivos y las condiciones de mantenimiento del comportamiento.

```
while (true)
     /* UpdateBeliefs(environment); */

     forall( behav ) do
         if ChangedGoals( behav ) then
             UpdateBehaviour( behav );
         endif
     endforall

     while (e=pop(eventq))
         UpdateRelevantCapacities();   // puede limitarse en sistemas acotados
     endwhile

     forall( capac | IsRelevant( capac ) ) do
         if IsApplicable( capac ) then
             ExecuteCapacity( capac );
         endif
     endforall
 endwhile
```

Figura 5.4: Ciclo de ejecución de un agente

3. `UpdateBehaviour()`, actualiza el conjunto de comportamientos activos. En función del contexto asociado a cada comportamiento, se activarán los comportamientos cuyo contexto sea cierto y se desactivarán los que dejen de serlo.

4. `pop(eventq)`, lee el siguiente evento a tratar.

5. `UpdateRelevantCapacities()`, pasa a relevantes aquellas capacidades para las que se cumple su *Evento* de disparo y su *Condición* de activación.

6. `IsRelevant(capac)`, determina si la capacidad es relevante (ha llegado el evento de disparo y se debe evaluar su condición).

7. `IsApplicable(capac)`, determina si la capacidad es aplicable (se ha cumplido su condición y se puede ejecutar).

8. `ExecuteCapacity(capac)`, se cede el control al planificador de las capacidades para la ejecución correspondiente de las tareas asociadas.

5.3.2. Implementando los Comportamientos del agente

En cada agente se definen un conjunto de **Comportamientos** (*Behaviours*) para distinguir entre diferentes ambientes y focos de atención. Básicamente, los *Comportamientos* se utilizan para reducir y delimitar el conocimiento que tiene el agente para resolver un problema. Por lo tanto, los métodos, datos, eventos o mensajes que no estén relacionadas con el estado actual del agente no deben ser

considerados. De esta manera, se mejora la eficiencia del agente en el proceso de resolución de problemas.

Un *Comportamiento* tiene un *nombre* (*Name*) para su identificación. También tiene asociado un conjunto de *metas* (*Goals*) que pueden ser usadas indistintamente para su activación o como condición de mantenimiento (ver la Figura 5.5). Finalmente, el *estado* (*state*) indica la situación actual de activación. En un agente pueden coexistir varios *Comportamientos* activos al mismo tiempo.

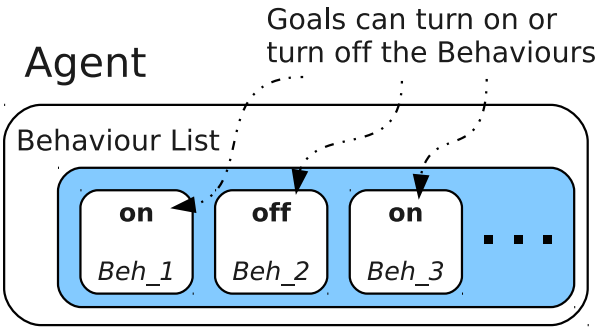

Figura 5.5: Los *Comportamientos* del meta-modelo del Agente

La clase `Behaviour` trabaja como un contenedor de las *Capacidades* de un agente y puede agrupar tantas *Capacidades* como el desarrollador desee implementar. Todas ellas pueden ser activadas o desactivadas con la llegada de eventos. Los *Comportamientos* son implementados por medio del *Receptor de Intenciones* que es la clase `BroadcastReceiver()` del API de *Android*. Esta clase recibe los eventos enviados por la plataforma *Android*.

El *Receptor de Intenciones* se registra dinámicamente para capturar las *intenciones*. Esto se realiza con el método `registerReceiver()` del API de *Android*. El *Receptor de Intenciones* será ejecutado cuando un evento llegue y éste coincida con el evento esperado, es decir, se une la *Intención* con el *Receptor de Intenciones*. Por ejemplo, el agente puede activar un *Comportamiento* especial cuando la batería del teléfono está a punto de agotarse. Para realizar esto el `BroadcastReceiver()` se configura para recibir la *intención* LOW_BATTERY.

La clase `Behaviour` diseñada posee distintos métodos que pueden ser utilizados por el usuario, pero hay dos métodos importantes que permiten añadir y borrar las distintas *Capacidades* que están contenidas en el *Comportamiento*, éstos son los métodos `add(capability)` y `remove(capability)` respectivamente. Cuando el usuario tiene que crear un nuevo *Comportamiento*, se debe proporcionar el nombre (*Behaviour name*) y su evento de activación. Parte de los principales métodos implementados para la clase `Behaviour` se muestran en la Figura 5.6.

```
public class Behaviour extends BroadcastReceiver{
    private List<Capability> myListCapability;
     . . .
    public void add(Capability mCapability)
    public boolean remove(Capability mCapability)
    public void activate()
    public void deactivate()
     . . .
}
```

Figura 5.6: Principales métodos de la clase *Behaviour*

5.3.3. Implementando las Capacidades del agente

Las *Capacidades* son almacenadas dentro de los *Comportamientos* y ellas modelan la respuesta del agente a determinado eventos. Una **Capacidad** se caracteriza por un *nombre* que la identifica, por su *Evento* de disparo, por una *Condición* (*Condition*) de activación y por la *Tarea* que será ejecutada cuando el evento es recibido y la condición correspondiente es completamente satisfecha (esto se ilustra en la Figura 5.7). La *Capacidad* también tiene un *Estado* que indica si está o no activa. Es de destacar que sólamente las *Capacidades* pertenecientes a un *Comportamiento* activo pueden ser ejecutadas.

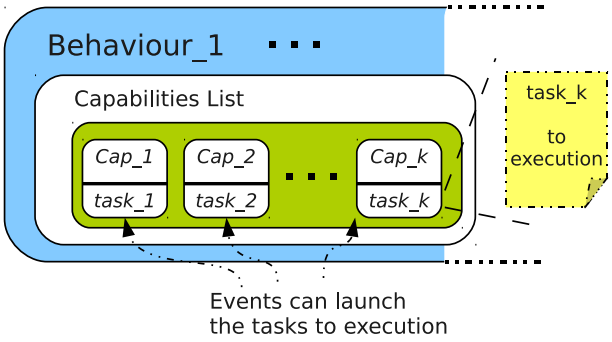

Figura 5.7: Las *Capacidades* del meta-modelo del agente

Un *Evento* (*event*) es cualquier notificación que llega al agente para informarle que algo que puede ser de su interés ha sucedido en el entorno (en el exterior) o en el interior del agente. Esto puede causar la activación de una nueva *Capacidad*.

Todas las *Capacidades* de un *Comportamiento* activo estarán en un estado marcado como *Activo*. Cuando un evento sea recibido, la *Capacidad* cambiará su estado a *Relevante* y su *Condición* de activación es evaluada. Si esta condición es completamente satisfecha, su estado pasará a *Aplicable* y la *Tarea* asociada comenzará su ejecución. Cuando esta *Tarea* finalice, la *Capacidad* retornará a su estado *Activo* de nuevo y se mantendrá a la espera de un nuevo evento. Cuando

un *Comportamiento* pasa a estar *Inactivo*, todas sus *Capacidades* detienen su ejecución y cambian su estado a inactivo. Asumimos que todas las *Capacidades* de un *Comportamiento* activo pueden ser ejecutadas concurrentemente, así que el sistema debe proporcionar los mecanismos necesarios para prevenir estancamientos e incoherencias durante su ejecución.

El concepto de *Capacidad* es implementado por medio de un *Receptor de Intenciones* del API de *Android*. Esta clase está a la espera de recibir algún evento lanzado por la plataforma de manera similar a los *Comportamientos*. Una *Capacidad* está siempre ejecutándose como un `BroadcastReceiver()`. Cuando una *Intención* es recibida y su condición se satisface, entonces el código en el método `onReceive()` será considerado como un proceso en primer plano y se mantendrá en ejecución por el sistema mientras la *Intención* es procesada, en este momento la *Tarea* asociada es lanzada.

La clase `Capability` tiene un método muy importante para crear una relación entre la *Tarea* y su correspondiente *Capacidad*, esté es el método `addTaskRun(task)`. Cuando el diseñador necesita crear una *Capacidad* se debe suministrar el nombre de la misma y su *Evento* de disparo. En la Figura 5.8 se muestran los principales métodos creados para la clase `Capability`.

```
public class Capability extends BroadcastReceiver{
     private Condition condition;
     private Boolean state;
     . . .
     public void activate()
     public void deactivate()
     public void setCondition(Condition condition)
     public boolean addTaskRun(Task nametask)
     . . .
}
```

Figura 5.8: Principales métodos de la clase *Capability*

5.3.4. Implementando las Tareas del agente

El último componente fundamental del meta-modelo del agente es la **Tarea** (*Task*). Las *Tareas* son elementos que contienen el código asociado a las *Capacidades* del agente. Una *Tarea* en ejecución pertenece sólamente a una *Capacidad* y se mantendrá en ejecución hasta su finalización o hasta que su *Capacidad* sea interrumpida porque su *Comportamiento* pasa a estar desactivado. Las *Tareas* no definen ningún método de recuperación ni de reanudación después de ser interrumpida. De otra forma, el agente debe tener mecanismos de "parada segura" para evitar caer en estados inconsistentes.

En la *Tarea* se ubica el código base de todas la acciones que el agente puede ejecutar. Dado que es la base de cómo se deben resolver los problemas, existen diversos tipos de tareas que pueden ser instanciadas y ejecutadas para resolver problemas específicos. Es decir, diferentes especializaciones del modelo genérico. Una *Tarea* puede ejecutar acciones simples como un "One Shot Task" (tarea de una sola ejecución), o puede realizar actividades un poco más complejas como cíclicas, secuenciales, tipo lazo, etc. En la Figura 5.9 se ilustran los diferentes tipos de *Tareas* que tiene el meta-modelo de agente πVOM y que han sido incorporados a ANDROMEDA.

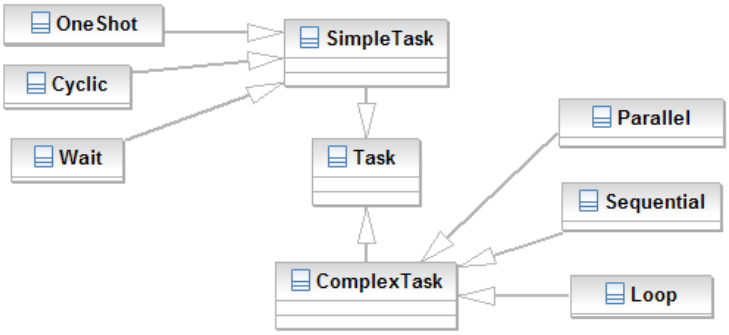

Figura 5.9: Meta-modelo de las *Tareas* en el agente πVOM

La clase `Task` se implementa como un proceso especial que se ejecuta como un `Servicio` en *Android*. Para desarrollar el concepto *Tarea*, algunos métodos de la clase `Servicio` han sido sobrecargados. En el método `onCreate()` de un `Service` se han inicializado las variables de la *Tarea*.

Posteriormente se ejecuta el método `onStart()` de un `Servicio` y en este punto se permitirá ejecutar el código del usuario que está asociado a la *Tarea*. Este código se debe escribir sobrecargando el método `doing()`. Con esto el programa Java escrito por el diseñador pasará a ejecución. En la Figura 5.10 se muestran los principales métodos diseñados para la clase *Task*.

5.3.5. El modelo de comunicación del agente Andromeda

La comunicación entre los agentes cumple con el estándar FIPA ACL (Agent Communication Language)[86]. Los mensajes están basados en la teoría de los *actos del habla* (speech acts) y representan la intención de un agente de realizar una acción (acto comunicativo). Estos mensajes se transportan utilizando las librerías de protocolo de transferencia de hipertexto (HTTP, por sus siglas en inglés, Hypertext Transfer Protocol) sobre el protocolo TCP/IP. Por ejemplo, uno de estos mensajes es el observado en la Figura 5.11, que muestra la comunicación

```
public class Task extends Service implements Runnable {
    public MsgQueue outputQ
    . . .
    public void doing() //contendrá el código del usuario
    public synchronized void pause()
    public synchronized void resume()
    public void taskDestroy()
    public final void send(Message msg)
    public final Message receive(MessageTemplate pattern)
    public final Message blkReceive(MessageTemplate pattern, long time)
    . . .
}
```

Figura 5.10: Principales métodos de la clase *Task*

de un agente ANDROMEDA con un agente en una Institución Electrónica. En el mensaje, el agente en `AndroClient@andromeda` (ANDROMEDA) solicita permiso para entrar en una subasta al agente `provider4@127.0.1.1:1099/JADE` (en una E-Institutions), utilizando un protocolo propietario llamado `START_BID`. El agente ANDROMEDA está en el teléfono móvil y sirve como agente interfaz a un usuario real (un pujador) en el proceso de subasta.

```
(INFORM
 :sender  ( agent-identifier :name AndroClient@andromeda
                             :addresses (sequence http://158.42.244.38:9090/ ))
 :receiver  (set ( agent-identifier :name provider4@127.0.1.1:1099/JADE
                             :addresses (sequence http://158.42.185.186:7778/acc )) )
 :content  "Start"
 :protocol  START_BID
)
```

Figura 5.11: Un mensaje FIPA en ANDROMEDA

Un mensaje FIPA ACL contiene un conjunto de uno o más parámetros. Precisamente los parámetros que son necesarios para la comunicación variarán de acuerdo con la situación. Estos parámetros del mensaje FIPA ACL pueden clasificase en

1. Tipo de acto comunicativo: *performative* (ilocución)
2. Participantes de la comunicación: *sender, receiver, reply-to*
3. Contenido del mensaje: *content*
4. Descripción del contenido: *language, encoding, ontology*
5. Control de la conversación: *protocol, conversation-id, reply-with, in-reply-to, reply-by*

El mensaje implementado en ANDROMEDA soporta todos los parámetros estándar del mensaje FIPA ACL. Estos parámetros se resumen en la Tabla 5.2

Tabla 5.2: Parámetros del mensaje ANDROMEDA

Parámetro	Descripción
:content	Contenido del mensaje
:sender	Identidad del Emisor del mensaje
:receiver	Identidad del Receptor del mensaje (un agente o una lista de agentes)
:language	El nombre del lenguaje de representación empleado en el atributo :content
:ontology	El nombre de la ontología utilizada en el atributo :content
:reply-with	Etiqueta para la respuesta (si es que el emisor la espera)
:in-reply-to	La etiqueta esperada en la respuesta
:protocol	Identificador del protocolo de interacción que se está utilizando
:conversation-id	Identificador de una secuencia de actos comunicativos que forman parte de una misma conversación
:reply-to	Agente al que han de ser enviadas las respuestas (si no es el emisor)
:reply-by	Indicación del tiempo en el que se quiere que se responda al mensaje

En los mensajes de ANDROMEDA se implementaron algunos protocolos de interacción, como también los actos de comunicación (*performatives*).

Las ***Performatives***, es una ilocución que conlleva un acto del habla. Esta identifica el acto del habla, la cual es imprescindible para una correcta comunicación. Los agentes puede realizar diferentes ilocuciones para: informar, preguntar, sugerir, prometer. En ANDROMEDA se pueden utilizar las ilocuciones que aparecen en la Tabla 5.3 y que son la mayoría de las definidas por FIPA.

El **protocolo de interacción**, es una descripción detallada del tipo y orden de los mensajes involucrados en una conversación entre agentes. Para establecer una conversación entre agentes es necesario definir previamente el protocolo que van a seguir durante la conversación. Para utilizar un protocolo en una conversación, los agentes deben definir el nombre del protocolo a utilizar en el parámetro: *protocol*. Finalmente, la versión actual de ANDROMEDA soporta dos de los más importantes protocolos de interacción. El protocolo FIPA-QUERY, que permite que un agente solicitar a otro agente que le envíe cierta información, como se

Tabla 5.3: Ilocuciones soportadas por ANDROMEDA

ACCEPT-PROPOSAL	AGREE	CANCEL
CFP	CONFIRM	DISCONFIRM
FAILURE	INFORM	INFORM-IF
INFORM-REF	NOT-UNDERSTOOD	PROPOSE
QUERY-IF	QUERY-REF	REFUSE
REJECT-PROPOSAL	REQUEST	REQUEST-WHEN
REQUEST-WHENEVER	SUBSCRIBE	PROXY

observa en la Figura 5.12 y el protocolo FIPA-REQUEST, el cual permite que una agente solicitar a otro agente que realice un tipo de acción, como observa en la Figura 5.13. Otros protocolos pueden ser implementados *ad-hoc* por parte de los desarrolladores.

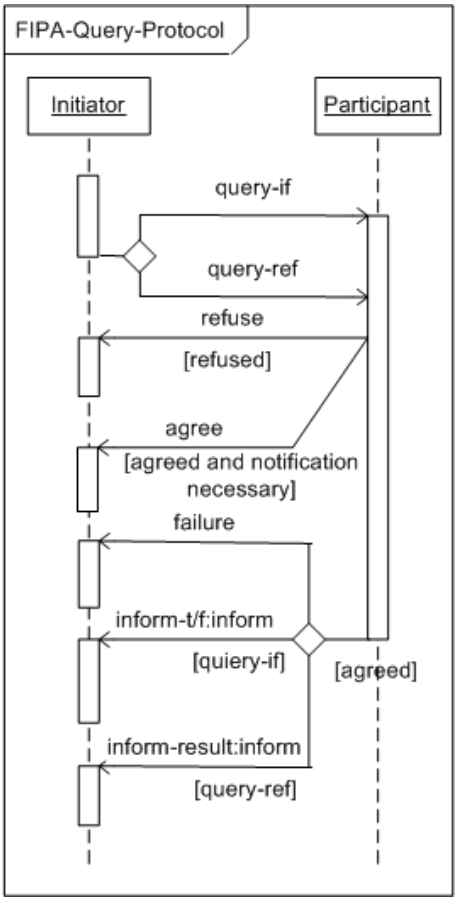

Figura 5.12: Protocolo FIPA-QUERY

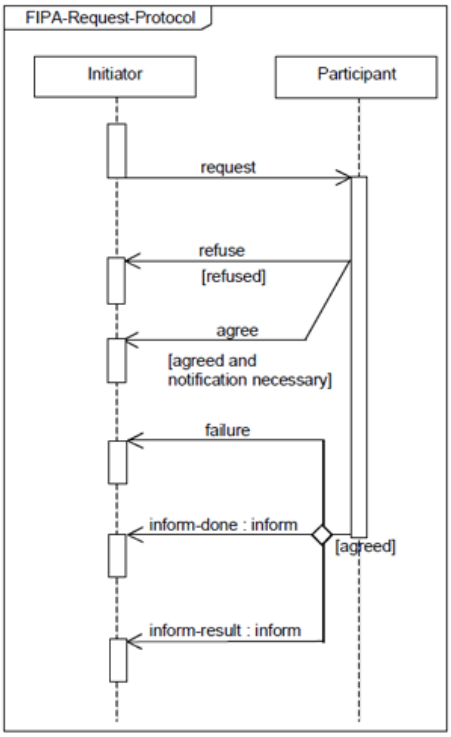

Figura 5.13: Protocolo FIPA-REQUEST

5.3.6. Otros componentes del agente Andromeda

Finalmente, existen otros conceptos del meta-modelo del agente πVOM que usan totalmente el API de *Android*. Como son:

- Los *Eventos*, son modelados con el componente *Intent* de *Android*. Las intenciones representan la "intención" o solicitud de que alguno de los componentes lleve a cabo una tarea. Las intenciones ofrecen un servicio de paso de mensajes que permite interconectar componentes de la misma o de distintas aplicaciones. Las intenciones se utilizan para: iniciar actividades o enviar eventos a múltiples destinatarios

- Las *creencias* del agente se implementan usando el elemento *ContentProvider*. El *ContentProvider* permite compartir información entre aplicaciones. Un agente que desee que toda o parte de su información almacenada esté disponible de una forma controlada para el resto de agentes del sistema deberá proporcionar un *ContentProvider*, a través del cual se puede realizar el acceso a dicha información. Este mecanismo es utilizado por muchas de las aplicaciones estándar de un dispositivo *Android*, como por ejemplo la lista de contactos, la aplicación de SMS, o el calendario/agenda.

- También existen algunos conceptos que usan parcialmente el API de *Android*, por ejemplo el concepto **Goal** tiene la capacidad de lanzar *Intenciones* para indicar que una meta se alcanzó. Por otro lado, hay algunos conceptos que son implementados con componentes propios de Java, por ejemplo, los protocolos de comunicación sobre HTTP, los mecanismos para el descubrimiento de otros agentes empotrados o de plataformas de agentes que utilizan Bonjour[1] o multicast DNS (mDNS).

5.4. Diseñando agentes con Andromeda

Como fue comentado anteriormente, los cuatro principales conceptos del metamodelo de agente de πVOM (*Tasks*, *Capabilities*, *Behaviours* y *Agent*) usan los componentes de *Android* de una forma completa, ya que las clases que implementan dichos conceptos se extienden directamente de los componentes *Android*. De esta forma, con propósitos ilustrativos se muestra en la Figura 5.14 un diagrama de clases que permite observar la relación directa entre los conceptos del agente πVOM y los componentes del API de *Android*.

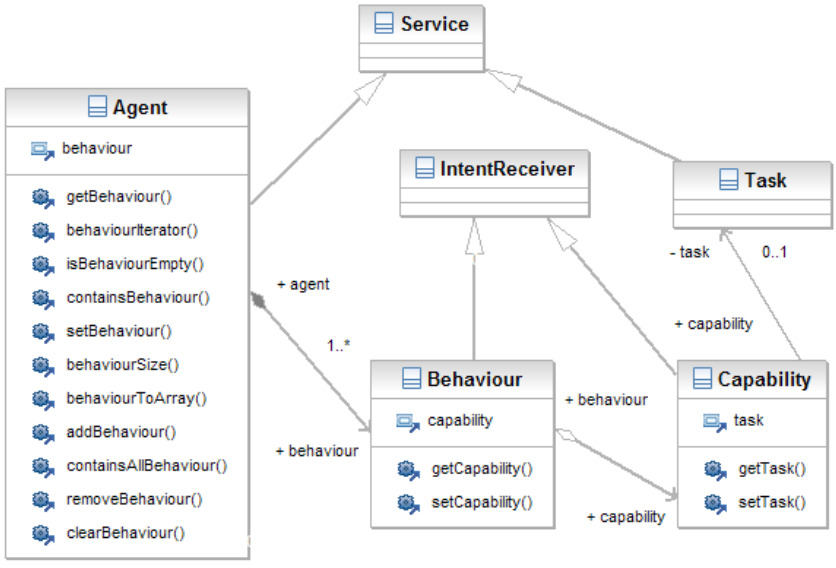

Figura 5.14: Relación entre los conceptos del agente πVOM y *Android*

De manera ilustrativa se propone un esqueleto o un esquema "típico" de cómo debe ser un agente sobre ANDROMEDA. El agente (en código) se muestra en la

[1]Implementación de *Zeroconf* realizada por Apple, https://developer.apple.com/opensource/

Figura 5.15. Esta plantilla nos indica las partes de código necesarias para que el agente lleve a cabo las actividades encomendadas.

```
public class MyAgent extends Agent {
  public void init(){
    . . .
       /*********************************************************************
       * Crear los comportamientos necesarios para el agente (mínimo uno)
       *********************************************************************/
        Behaviour myBehaviour_1 = new Behaviour("Comportamiento1");

        /************************************
        * Crear las capacidades necesarias
        ************************************/
        Capability myCapability_1 =  new Capability("capacidad1");
        Capability myCapability_2 =  new Capability("capacidad2");

        /*************************************************************************
        * Crear las condiciones de las capacidades y su evento de activación
        *************************************************************************/
        Condition mycondicion1 = new Condition() {
          @Override
          public boolean expression(Event event) {
             if (event.getAction() == "Android.intent.action.LOW\_BATTERY") {
                return true;
              }else { return false; }
          . . .

        /****************************************
        * Unir las condiciones a la capacidades
        ****************************************/
        myCapability_1.setCondition(mycondicion1);
        myCapability_2.setCondition(mycondicion5);

        /****************************************
        * Crear las tareas que ejecuta el agente
        ****************************************/
        Task myTask_1 = new Task();
        Task myTask_2 = new Task();

       /*********************************
        * Unir las tareas a la capacidad
        *********************************/
        myCapability_1.addTaskRun(myTask_1);
        myCapability_2.addTaskRun(myTask_2);

       /*********************************************
        * Añadir las capacidades al comportamiento
        *********************************************/
        myBehaviour_1.add(myCapability_1);
        myBehaviour_1.add(myCapability_2);

       /*********************************************************************
        * Añadir el comportamiento al agente y asi se inicia su ejecución
        *********************************************************************/
        addbehav(myBehaviour_1);
}
```

Figura 5.15: Esquema general de un agente ANDROMEDA

Como se muestra en el esqueleto, se han añadido comentarios para mejorar la comprensión por parte del lector. Se observa que cada uno de los componentes del agente se "enlaza" con los otros según como se específica en el modelo. Pero merece especial atención la parte del código que corresponde a la *Condición*, que es necesario especificar cuál será el atributo a evaluar para saber si la *Condición* se satisface o no. Esto se debe realizar dentro del método `expression(Event event)`. En este caso, se evalúa si el *Evento* recibido se corresponde a la "etiqueta" que indica si el dispositivo está a punto de agotar la batería (low_battery), es decir, se verifica la igualdad,

$$event.getAction() == Android.intent.action.LOW\_BATTERY \quad (5.1)$$

Si el dispositivo tiene el estado de la batería bajo, la `expression()` será verdadera y esto activará una *Tarea*. Pero en el caso contrario `expression()` retorna falso, esto se interpretará como que no se debe ejecutar ninguna tarea. En nuestra propuesta muchos agentes se ejecutan no sólo en teléfonos y tabletas, si no que existen escenarios donde hay la necesidad de contar con computadores empotrados para acceder al entorno. A continuación se describe como ejecutar los agentes en estos dispositivos.

5.5. Los sensores y actuadores en Andromeda

Como se comentó anteriormente, para ejecutar agentes tipo ANDROMEDA en un teléfono móvil o tableta, ésta debe utilizar como sistema operativo *Android* (se soportan las versiones antiguas, a partir de *release* 1.6 y las actuales). También necesitaríamos ejecutar agentes tipo ANDROMEDA en un computador empotrado o en otro tipo de artefacto o dispositivo, ya que nos interesa utilizar sus recursos de bajo nivel para acceder al entorno. El requerimiento principal que se tiene que cumplir es que el hardware debe tener instalado un *Android* empotrado, y con ello garantizar que el modelo de agente propuesto puede ejecutarse.

En el mercado existen algunos artefactos que poseen *Android* como sistema operativo, sin embargo, en este trabajo se seleccionó la tarjeta Beagleboard como hardware empotrado (por las distintas funcionalidades que proporciona), y con ello lograr que los agentes accedan directamente al entorno. Sin embargo, la tarjeta Beagleboard no viene de fábrica con *Android*, y por ello se debe migrar este sistema operativo a la tarjeta. El proceso de migración realizado en este trabajo, para soportar *Android* sobre la Beagleboard (Versión C3) se resume en el anexo I.

Para que los agentes y el marco OSGi (en el lenguaje Java de *Android*) puedan acceder a las funcionalidades de los sensores y actuadores, es necesario interactuar con los *drivers* del dispositivo. Sin embargo, algunos de los *drivers* de bajo

nivel de los dispositivos que están implementados en lenguaje C o C++ por el fabricante, y que es necesario de encapsular a lenguaje Java que es utilizado por los agentes en ANDROMEDA o por el marco OSGi. Es decir, la interacción o comunicación con cada *driver* dependen principalmente del fabricante del dispositivo, que generalmente suministra las funcionalidades del dispositivo especificado el lenguaje de programación, protocolos, interfaces de acceso, parámetros de configuración, etc.

Por ello, generalmente la interacción está muy marcada por las características suministradas por el fabricante del dispositivo. Es necesario proporcionar a los agentes una interfaz común y uniforme, independiente del fabricante del dispositivo y que sólo dependa del tipo específico (o familia) de sensor/actuador a usar. Una forma de lograr esto es encapsular las funciones dependientes del fabricante, para luego exportar y suministrar a los agentes una interfaz con características de más alto nivel.

Para implementar este encapsulado del *driver* como una interfaz común, es un paso que depende de varios factores, pero el primordial es el lenguaje en que están escritos. El lenguaje del *driver* por lo general es dado por los fabricantes, en ocasiones se cuenta con varias implementaciones en diferentes lenguajes o algunos dispositivos son muy simples de operar y no es necesario desarrollar un *driver* específico, como por ejemplo el encender una bombilla común. Para hacer el encapsulamiento de los dispositivos en Android diferenciamos entre dos lenguajes de programación: Java y el lenguaje nativo (código C o C++).

5.5.1. Dispositivos con drivers en lenguaje C

En este caso, que posiblemente es el más común, muchos de los *drivers* hacen uso del **sysfs**[2], que es un sistema de archivos virtual que proporciona el núcleo Linux a partir de la versión 2.6. `Sysfs` exporta información sobre el dispositivo y sus controladores desde el modelo del dispositivo en el núcleo hacia el espacio del usuario.

Para acceder al dispositivo, sus funcionalidades, parámetros y atributos (a su *driver*), se realiza accediendo a simples ficheros. Estos ficheros están incluidos en el subdirectorio del controlador correspondiente al dispositivo. Por lo general estos ficheros contienen un sólo valor y para cada dispositivo añadido se crea un directorio en `sysfs` (o un árbol de ficheros y directorios correspondientes al modelo del controlador y dispositivo). Se establece una relación padre/hijo que se refleja con subdirectorios bajo `/sys/devices/` (reflejando la capa física). En el subdirectorio `/sys/bus` se crean diferentes enlaces simbólicos, reflejando

[2]https://www.kernel.org/pub/linux/kernel/people/mochel/doc/papers/ols-2005/mochel.pdf

el modo en el que los dispositivos pertenecen a diferentes buses. En el directorio `/sys/class` se muestran los dispositivos agrupados de acuerdo a su clase, como por ejemplo red, de entrada/salida digital, etc., mientras que `/sys/block/` contiene los dispositivos de bloques.

Sin embargo, para que en *Android* estos dispositivos sean accedidos a través del `sysfs`, se debe habilitar su soporte. Esto se logra cuando se está configurando el kernel para antes de su compilación. Por ejemplo para habilitar el uso de los puertos digitales de propósito general de entrada y salida (GPIO, de sus siglas en inglés General Purpose Input/Output), en el comando de configuración del kernel `make ARCH=arm menuconfig`, realizamos lo que se muestra en la Figura 5.16

```
Device Drivers  ---> GPIO Support  ---> /sys/class/gpio/... (sysfs interface)
```

Figura 5.16: Habilitar para que GPIO esté soportado en **sysfs**

Esto permite que los puertos o pines del GPIO sean accedidos como ficheros por el usuario `root`. Para exportar estos ficheros al espacio de las aplicaciones *Android* (diferente al administrador), se debe asignar los permisos correspondientes en el fichero `init.rc`. Por ejemplo, para el puerto `gpio108` tenemos (ver Figura 5.17):

```
chown system system /sys/class/gpio/gpio108/direction
chown system system /sys/class/gpio/gpio108/value
chmod 0666 /sys/class/gpio/gpio108/direction
chmod 0666 /sys/class/gpio/gpio108/value
```

Figura 5.17: Ajuste del `init.rc` para usar el puerto `gpio108`

Con esto podemos decidir si `gpio108` es un puerto de entrada o salida, escribiendo en el fichero `/sys/class/gpio/gpio108/direction` un cero (0) o un uno (1). Si el puerto es de salida, el valor se envía al puerto escribiendo en `/sys/class/gpio/gpio108/value`. Si el puerto es de entrada, obtenemos el valor del puerto leyendo el fichero `/sys/class/gpio/gpio108/value`. Esto nos permite que aplicaciones escritas en C o C++ escriban y lean los puertos del GPIO. Por ejemplo, una vez que se configura el puerto `gpio108` como salida, podemos escribir un valor alto de voltaje (un 1 lógico, por lo general 5,0 Voltios o 3,3 Voltios) que sirve para activar el dispositivo en el entorno, con un código como el mostrado en la Figura 5.18.

Finalmente, como se explicó en el capítulo anterior, para poder usar los dispositivos desde la capa de agente, las funcionalidades de los dispositivos se convierten en servicios (OSGi y Web). Para que los agentes usen los servicios se debe

```
fp = fopen("/sys/class/gpio/gpio108/value", "rb+"));
strcpy(set_value,"1");
fwrite(&set_value, sizeof(char), 1, fp);
```

Figura 5.18: Se escribe un uno (1) lógico en el puerto `gpio108`

exportar las funcionalidades a un nivel más alto, a los lenguajes Java o Java Android. Esto se logra utilizando en *Android* (ANDROMEDA) la herramienta NDK (de sus siglas en inglés, Native Development Kit). El NDK es un conjunto de herramientas que permiten compilar, ejecutar y usar código nativo desde aplicaciones Android (el cual ofrece acceso a funciones específicas de las capas bajas de la arquitectura de Android, por ejemplo la capa de hardware). Para realizar esta función NDK utiliza el marco JNI (de sus siglas en inglés, Java Native Interface). JNI es un marco de programación que permite que un programa escrito en Java ejecutado en la máquina virtual pueda interactuar con programas escritos en otros lenguajes como C, C++ y ensamblador. En JNI, las funciones nativas se implementan en archivos `.c` o `.cpp` por separado. Cuando la máquina virtual invoca a la función, le pasa un puntero a `JNIEnv`, un puntero a `jobject`, y cualquier número de argumentos declarados por el método Java. Una función de JNI debería ser parecida al código de la Figura 5.19.

```
JNIEXPORT void JNICALL Java_ClassName_MethodName
   (JNIEnv *env, jobject obj)
   {
     //El método nativo se implementa aquí
   }
```

Figura 5.19: Cabecera JNI

5.5.2. Dispositivos con drivers en lenguaje Java

En este caso suponemos que el fabricante nos proporciona los *drivers* del dispositivo en lenguaje Java, y que muchos de éstos posiblemente sean externos al hardware base del computador empotrado, teléfono móvil o tableta. Aunque, hay algunos dispositivos propios del hardware que son parte del sistema *Android*: como las brújulas, giroscopios, cámaras, etc. de los teléfonos móviles y tabletas. Estos dispositivos están soportados por *Android* y se pueden usar directamente, ya que el fabricante del dispositivo móvil los incluyó en el kernel.

Nos interesa describir el proceso de los dispositivos externos al hardware del computador base. Sin embargo, como las aplicaciones *Android* (y los agentes ANDROMEDA) no pueden acceder directamente al `sysfs` del kernel (sólamente con

código nativo), las aplicaciones pueden usar dispositivos directamente si usan como interfaz el puerto USB, como es el caso de los dispositivos llamados phidgets[3].

Con estos dispositivos solo hay que incluir las librerías suministradas en el proyecto *Android*, como por ejemplo la librería `PhidgetsUSB.jar` para que la aplicación pueda importar y usar las diferentes funcionalidades del dispositivo (`import com.phidgets.*;`). Además, se debe especificar los permisos de uso del puerto USB, agregando la línea `<uses-feature android:name = android.hardware.usb.host/>` en el fichero `AndroidManifest.xml`.

5.5.2.1. Tarjetas de expansión sobre el puerto USB

En este caso, supone que se tiene un dispositivo *Android* y se conecta por el puerto USB una tarjeta de expansión. La tarjeta de expansión contiene, generalmente un conjunto de puertos de entrada-salida y un micro-controlador de recursos limitados y bajo coste, como lo ilustra la Figura 5.20, que muestra un teléfono móvil que controla una tarjeta por su puerto USB. Aunque esta configuración es similar al utilizar los sensores llamados phidgets, su diferencia radica es que un sensor phidgets, por lo general, puede medir sólo la variable específica para el cual fue diseñado (por ejemplo, sólo registra la temperatura del entorno). En cambio, una tarjeta de expansión permite conectar un conjunto de sensores y actuadores según la cantidad de puertos de entrada y salida disponibles.

Figura 5.20: Dispositivo *Android* usando una tarjeta de expansión

[3]http://www.phidgets.com

Esta configuración tiene como objetivo proporcionar sensores a equipos o artefactos que son cerrados, en el sentido, que el desarrollador no puede conectar nuevo hardware directamente sobre la placa base del equipo, si no que los sensores sólo se pueden conectar, si es posible, a través de una interfaz de uso general, como lo es el puerto USB. Por ello, esta configuración no tiene mucho sentido cuando ANDROMEDA se ejecuta sobre la Beagleboard, ya que en este caso, la tarjeta Beagleboard es el equipo con *Android*, y si permite agregar nuevo hardware sobre su sistema base.

Actualmente, esta configuración puede ser implementada con tres tipos de tecnologías, cada una tiene sus variantes y las describimos a continuación:

- Uso de ***Android* Open Accessory Protocol**[4]. Esta es la solución oficial de Google, es un estándar que Google creó y soporta, cuando lanzó al mercado el *Android Open Accessory Development Kit*. Básicamente al conectar una pieza de hardware en el puerto USB de un dispositivo *Android*, el nuevo hardware se convierte en un accesorio y como tal empieza a tomar energía para funcionar. En esta solución Google recomienda usar una tarjeta oficial, un hardware por ellos patrocinado, que lleva un firmware especial que permite una fácil interacción con el accesorio, ya que Kit de desarrollo enmascara todo el protocolo de bajo nivel. *Android* permite usar esta funcionalidad a partir de su versión 3.0.

- Uso de la **tarjeta IOIO**[5]. Esta solución fue realizada por un diseñador de sistemas empotrados, bajo la modalidad de software libre. El desarrollo incluye una tarjeta de expansión con un firmware específico y un conjunto de bibliotecas que permiten su uso bajo *Android*. La tarjeta de expansión es implementada como hardware libre por algunas empresas de electrónica. Como la tarjeta de expansión se le instala un firmware especial que se comunica directamente con las bibliotecas incluidas, esto facilita el la implementación de aplicaciones, ya que el desarrollador no debe programar el protocolo de bajo nivel (igual que la solución *Android* Open Accessory Protocol). Las bibliotecas se pueden en *Android* a partir de su versión 1.5.

- Uso de la **tarjeta Arduino**[6]. Esta solución fue propuesta por la comunidad de desarrollo de *Arduino*. La placa *Arduino* es una popular tarjeta de propósito general, usada en la creación rápida de prototipos en electrónica, con una amplia gama de aplicaciones. Esta propuesta consiste en crear un simple protocolo *ad-hoc*, definido por el desarrollador. El protocolo consiste

[4]http://developer.android.com/tools/adk/index.html
[5]https://github.com/ytai/ioio/wiki
[6]http://www.arduino.cc/es/

en el paso de mensajes a través del puerto USB, que la placa *Arduino* recibe y decodifica. El desarrollador decide qué hacer con cada tipo de mensaje. Este enfoque al ser una solución general, permite ampliar la gama de operaciones que se pueden realizar, sin embargo, el desarrollador debe tener habilidades de programación para la placa *Arduino*, ya que el desarrollador debe programar el protocolo de bajo nivel. Por ello, debe conocer la arquitectura del micro-controlador y lenguaje C o ensamblador.

Finalmente, podemos decir que la arquitectura de implantación y la plataforma de ejecución (ANDROMEDA) propuestas, permiten el desarrollo de MAS ubícuos, ya que soportan la incorporación de diversos dispositivos para interactuar sobre el entorno.

5.6. Conclusiones

Este capítulo ha presentado una nueva plataforma de agentes empotrados, ANDROMEDA, basada en sistema operativo *Android*. El objetivo de crear nuestra propia plataforma es porque *Android* es una poderosa y novedosa plataforma de soporte para la ejecución de aplicaciones empotradas. Esto se debe a que el API de Java de *Android* es potente, ya que es muy similar al Java para computadores personales (J2SE), permitiendo implementar agentes empotrados con mecanismo más avanzado. Adicionalmente como *Android* es un sistema Linux, puede migrase con cierta facilidad a otros dispositivos distintos a los teléfonos móviles o tabletas, lo que permite que los agentes puedan ejecutarse en variados aparatos. Se ha mostrado como nuestros agentes pueden ejecutarse en el computador empotrado Beagleboard.

En este capítulo también se ha descrito como utilizar dispositivos externos con ANDROMEDA, para dotar a los agentes (y a la arquitectura global del sistema) de características ubícuas. Estos dispositivos se pueden incorporar de distintas formas en la arquitectura de implantación, según la tecnología involucrada, y posteriormente, los agentes del sistema pueden percibir y controlar su entorno físico. Además, se describió como la plataforma ANDROMEDA ha implementado el meta-modelo de agente πVOM sobre *Android*. Por ello, se explicó brevemente el API de *Android*, sus características y componentes más relevantes. Finalmente, en el siguiente capítulo se desarrollarán y presentarán dos ejemplos que demuestran la aplicación de esta propuesta.

6

Aplicación de la propuesta: escenarios de uso

Índice

6.1. Introducción

Para evaluar la propuesta descrita en este trabajo se ha creído conveniente desarrollar dos casos de estudio, en concreto dos prototipos que valoren empíricamente la propuesta, con el fin de poder comparar las ventajas aportadas por el enfoque MDD al desarrollo de *Organizaciones Virtuales Ubícuas* con respecto al enfoque clásico. En este capítulo se verifica como diseñar una *Organización Virtual Ubícua* a partir de las vistas del modelo unificado, que permitan al desarrollador diseñar un MAS ubícuo de forma sencilla, utilizando componentes UML que representan los conceptos abstractos del sistema (agentes, servicios, dispositivos, entornos, etc.), tratando de obviar los detalles de la implementación, que no son necesarios en esta etapa del diseño. Posteriormente, usando las

transformaciones del MDD trasladar de forma automática o semi-automática estos modelos a plantillas de código específicas de la plataforma de ejecución. Estas transformaciones llevan consigo todos los detalles de implementación necesarios, pero que se encuentran ocultos al desarrollador.

Se muestran las transformaciones para diferentes plataformas de agentes, las que soportan organizaciones como E-Institutions[80] y THOMAS[53], al igual que la plataforma clásica de MAS, como lo es JADE[30]. Para mostrar la interoperabilidad los agentes podrán ejecutarse en dispositivos móviles (en teléfonos, tabletas, etc.) que usen las plataformas JADE-Leap[32] y Andromeda[3, 2]. Es necesario resaltar que el uso de organizaciones en estos escenarios, permite crear un marco regulador para los agentes que componen las organizaciones. Este marco regulador soporta la especificación de qué comportamientos individuales y colectivos (de la sociedad) son los deseados dentro de la organización, e igualmente qué acciones son consideradas indebidas y no deberían ser ejecutadas por los agentes. La organización permite que la especificación de lo que está permitido y prohibido se realice de manera más sencilla (el modelo del proceso regulatorio). Este proceso regulatorio puede ser realizado en algunas plataformas MAS convencionales, sin embargo, su descripción es mucho más complicada, ya que no se cuenta con los componentes necesarios para su modelado e implementación.

En el caso de THOMAS, se facilita la descripción de un entorno abierto, donde los agentes pueden entrar y salir de las unidades organizacionales, donde los agentes pueden adoptar diferentes roles que regulan parte de su comportamiento, donde los agentes pueden publicitar y usar servicios, donde se permite crear en tiempo de ejecución nuevas unidades organizacionales con nuevos servicios. Funcionalidades que en otras metodologías o plataformas no son permitidas o no son soportadas de forma directa. De forma similar, las E-Institutions permiten especificar fácilmente una organización (en este caso estática, cuya estructura no se modifica en tiempo de ejecución) con un entorno normativo que afecta los comportamientos individuales y colectivos de los agentes. Si se compara este enfoque con otras metodologías de agentes clásicas, muchas de ellas se enfocan en el diseño y descuidan la implantación, por otra parte, algunas plataformas de implementación carecen de modelos de diseño que faciliten el desarrollo de agentes. En este capítulo, de escenarios de uso, se muestran todas las etapas del desarrollo, desde el diseño de agentes hasta la generación de plantillas de código, mediante el uso del enfoque MDD.

En este capítulo también se presenta el uso de la arquitectura de implementación que permite dotar de características ubícuas al sistema, es decir, que los agentes pueden administrar su entorno, usando los dispositivos físicos del sistema (sensores y actuadores) para crear un entorno inteligente (o ubícuo), donde las funcionalidades de cada dispositivo se encapsulan en un servicio genérico (a través de la tecnología OSGi ya que las funcionalidades dependen de cada fabri-

cante de los dispositivos), donde cualquier agente o unidad organizacional podrá ofrecerlo o solicitarlo. Muchas de las funcionalidades del sistema (de cada plataforma) se exportan a través de servicios para permitir una fácil y transparente inter-relación.

Nuestro primer ejemplo es un escenario usado principalmente para verificar la interoperabilidad de las organizaciones de la propuesta. Este escenario, donde se modela un *club de lectura*, se basa en un agente que puede entrar a dos organizaciones mixtas, y que están enfocadas a modelar organizaciones muy distintas del mundo real. Una de ella representa una institución estática, que está soportada por la plataforma E-Institutions, mientras que la otra modela una organización dinámica y adaptable la cual está soportada con la plataforma THOMAS. En este escenario se permite que un agente pertenezca a las dos organizaciones y establezca relaciones con agentes de las dos organizaciones.

Además se ha desarrollado también un escenario más complejo, que incluye interacción con el usuario, servicios de alto y bajo nivel con dispositivos empotrados para modelar una aplicación ubícua estándar mediante el paradigma de agente expuesto. El ejemplo en cuestión es una organización ubícua que administra un sistema de transporte para obtener un *Sistema de Transporte Inteligente*.

El *Sistema de Transporte Inteligente* es una aplicación que facilita la interconexión entre los pasajeros (ciudadanos, turistas) y los proveedores de transporte (autobuses, metros, trenes, tranvías), delimitando los servicios que cada uno puede solicitar u ofrecer según sea el caso. El *Sistema de Transporte Inteligente* permitirá a los usuarios móviles (pasajeros dotados con dispositivos móviles: Teléfonos, Tabletas, PDAs) conocer las ofertas de servicios de la Agencia de Transporte Público sobre la base de las preferencias del usuario, es decir, el usuario puede recibir o solicitar información sobre las posibles rutas turísticas, servicios de noticias personalizados, el tiempo estimado de llegada (ETA, del inglés Estimate Time Arrival), las rutas más cortas hacia el destino, etc.

La organización ubícua administra el transporte público y es utilizada para mejorar la calidad de los servicios y por lo tanto hacerlos más atractivos para los pasajeros. Esta aplicación proporcionará a los operadores de transporte público información en tiempo real sobre el estado operativo del sistema de transporte público, lo que les permitiría gestionar más eficazmente su flota, y tomar acciones correctivas si se producen interrupciones. Además, esta aplicación proporcionará información en tiempo real a los pasajeros, tanto en el vehículo y en las paradas de autobús.

6.2. Club de lectura: integración de organizaciones mixtas.

El siguiente ejemplo tiene como objetivo mostrar la utilidad de los modelos y de las transformaciones en el diseño de MAS basado en organizaciones, propuesto en este trabajo de Tesis. Para ello se propone integrar o combinar dos sociedades heterogéneas (como dos empresas en el mundo real), una representada por una organización virtual (con THOMAS), y la otra con una institución electrónica (E-Institutions). Cada una de ellas está enfocada a satisfacer necesidades muy diferentes, la institución electrónica representa una organización estática, y la organización virtual representa a una sociedad dinámica.

En el ejemplo combinamos el enfoque de las instituciones electrónicas, con el enfoque de las organizaciones virtuales. La integración de ambas tecnologías muestra la aplicabilidad de esta propuesta para soportar soluciones de un sistema organizacional híbrido, que permite modelar problemas complejos, de una manera más novedosa que utilizando un único enfoque. Con la integración de estas tecnologías podemos obtener lo mejor de cada uno de estas aproximaciones, logrando una solución más potente y más adaptada al mundo real.

Representar sociedades del mundo real en organizaciones electrónicas es una tarea compleja[21, 173]. Como fue mencionado en el estado del arte de la memoria (capítulo 2), las organizaciones sirven para realizar diferentes acciones, con distintas características, tienen variados escenarios que debemos considerar para su representación. Por ejemplo, algunas organizaciones son rígidas e inmutables, mientras otras son flexibles y pueden variar en el tiempo. Con estas dos aproximaciones trabajando separadamente se puede modelar muchas organizaciones, sin embargo, al integrar ambos enfoques se logra incrementar el rango de escenarios a los que se puede dar solución.

De otra forma, se propone representar dos grupos o sociedades de personas (agentes) muy diferentes (implementado en [173]). El primer grupo representa un club de lectura privado. Los miembros de este club, hablan e intercambian opiniones sobre libros y novelas de su interés. Se conoce que es muy difícil ser miembro de este club, ya que es un club de tipo VIP. Para unirse a este club, el nuevo miembro debe ser respaldado por una buena suma de dinero. El segundo grupo no tiene acceso a la información o conversaciones que se generan en el club. Este grupo contiene a las personas sin garantías suficientes para unirse al club. El principal objetivo del segundo grupo es obtener acceso a la información interna del club de lectura. Para realizar este objetivo, los no miembros del club (los foráneos al club) crean un grupo de colaboración, donde ellos hacen una contribución económica para respaldar a un candidato del grupo foráneo, para que sea miembro del club. De esta forma, al unirse uno de ellos al club, puede

mantener informado al resto del grupo.

Cada grupo de personas puede ser representado electrónicamente como una organización. El club con reglas y objetivos fijos puede ser modelado como una institución electrónica, mientras el otro grupo, donde las personas pueden unirse o separarse sin ninguna restricción, y hacer contribuciones económicas al grupo, puede ser modelado en THOMAS (como una organización virtual abierta). De esta forma, se pueden representar las diferentes necesidades, aspectos y prerrequisitos de ambos grupos. En esta propuesta, existe un agente (el agente candidato) que es capaz de interactuar con la organización basada en THOMAS y con la institución electrónica al mismo tiempo, como se muestra en la Figura 6.1. Las acciones realizadas por el agente candidato en un grupo o sociedad tienen implicaciones en el otro.

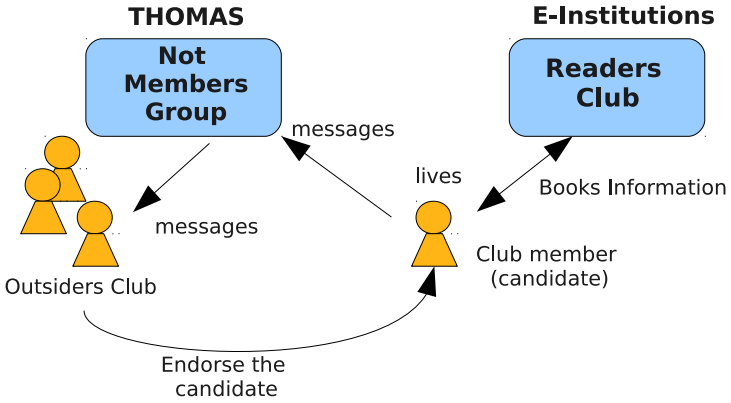

Figura 6.1: Esquema del sistema que integra la E-Institution y THOMAS

6.2.1. Diseño de la organización externa al club

El proceso de diseño comienza creando los modelos de la organización. Utilizando los modelos de πVOM se puede crear la institución electrónica y los agentes relacionados. Como en este ejemplo se quiere ilustrar el uso de la propuesta para crear dos modelos de organizaciones diferentes, uno para THOMAS y otro el de E-Institutions, los modelos principales a utilizar son el estructural, funcional y de agente, aunque hay más modelos que pueden complementar la propuesta, se usan estos tres para ilustrar el desarrollo del ejemplo. Para crear la integración entre los grupos, se ha diseñado un agente que llamaremos *BroadcastAgent*, este es el agente que puede ser miembro del club. Existe otro grupo de agentes, los *SubscriberAgent*, que no pueden ser miembro del club de lectura y por ello respaldan económicamente a *BroadcastAgent*, como se ilustra en la Figura 6.2. El

BroadcastAgent puede entrar como miembro al club de lectura (E-Institution) para participar en las conversaciones sobre libros, una vez que sale del club al mismo tiempo, se reúne con los *SubscriberAgent* (en THOMAS) para informarles sobre la reunión del club. Para soportar esta funcionalidad, se crea al menos un servicio al cual se deben subscribir los *SubscriberAgent* para recibir la información que requieren.

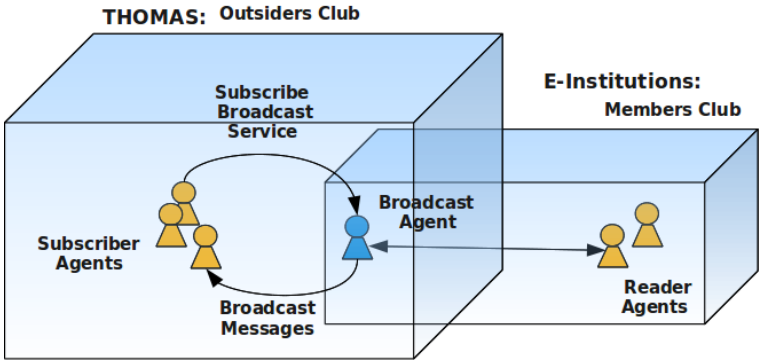

Figura 6.2: Agente Broadcast en la E-Institution y en THOMAS

6.2.1.1. Vista Estructural

El grupo de personas externas al club, están estructurados como se muestra en la Figura 6.3. Se tiene una unidad organizacional (OU), llamada *News*, que permite reunir a los agentes no miembros, recibir las contribuciones de los participantes y transmitir las noticias del club. La *News* posee dos roles, *Broadcast* y *Subscriber* que son adquiridos por los agentes participantes. Se supone que el agente que es miembro del club (*BroadcastAgent*) juega el rol de *Broadcast*) y tiene como objetivo transmitir las noticias a los demás agentes. Los *SubscriberAgent* deben adquirir el rol *Subscriber*, no pueden entrar a la institución electrónica y no reciben información directa del club, pero se les permite recibir las noticias del club de lectura.

6.2.1.2. Vista Funcional

En el modelo funcional se diseñan los servicios que proporcionan las unidades organizacionales. La unidad *News* tiene dos servicios *Broadcasting* y *Contributions*, como se observa en la Figura 6.4. El servicio *Broadcasting* permite distribuir las noticias del club de lectura a los no miembros, en cambio el servicio *Contributions* se encarga de recibir las contribuciones de los agentes de la unidad. También se definen los roles que pueden solicitar y proveer los servicios.

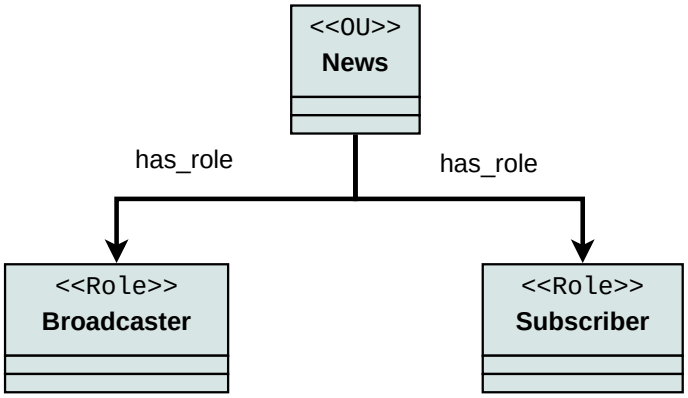

Figura 6.3: Modelo estructural del grupo externo al club

El rol *Subscriber* solicita el servicio de *Contributions* para dar su contribución monetaria para apoyar al agente candidato. Además, el rol *Subscriber* solicita el servicio *Broadcasting* que le permite recibir las noticias del club.

El rol *Broadcast* provee los dos servicios de la unidad *News*. Cuando proporciona el servicio *Contributions* conoce que agentes tienen autorización para recibir las noticias del club de lectura; y cuando provee el servicio *Broadcasting* periódicamente envía noticias a sus agentes subscriptores, es decir, envía todos los mensajes que aparecen en el club de lectura desde el último que fue enviado.

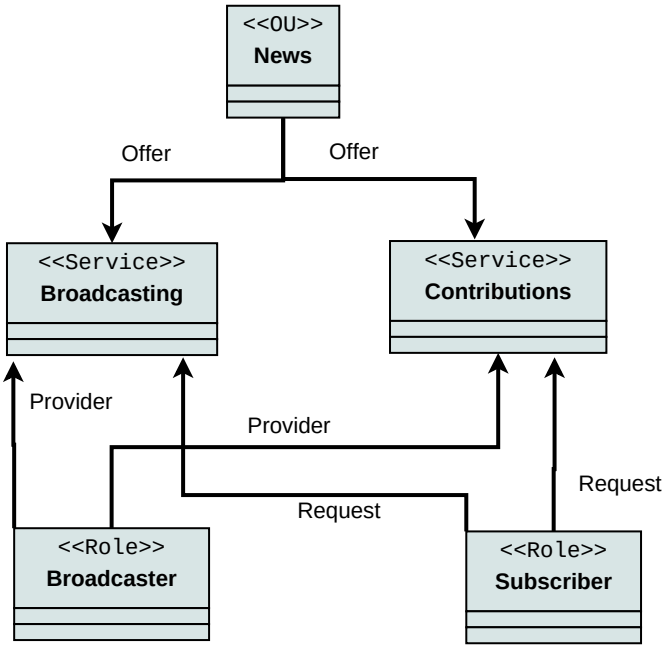

Figura 6.4: Modelo funcional del grupo externo al club

6.2.1.3. Vista de Agente

Los ***SubscriberAgent*** se diseñan para adquirir el rol *Subscriber*, como se muestra en la Figura 6.5. El *SubscriberAgent* posee un único comportamiento *SubscriberBehav* que le permite solicitar los servicios de la unidad *News*. El comportamiento *SubscriberBehav* tiene dos capacidades *Contributions* y *BroadcastMSG*. La capacidad *Contributions* es la que permite solicitar y manejar el servicio de contribuciones, es decir, permite que cada *SubscriberAgent* realice una contribución monetaria para respaldar a *BroadcastAgent* como candidato a ser miembro del club de lectura. Una vez que se realiza la contribución, puede solicitar que se le envíe los mensajes del club de lectura. La otra capacidad, *BroadcastMSG*, permite que el agente pueda solicitar el servicio de noticias, y si es autorizado (porque realizó la contribución económica), recibirá los mensajes periódicos que envía *BroadcastAgent* de lo que ocurre en el club de lectura.

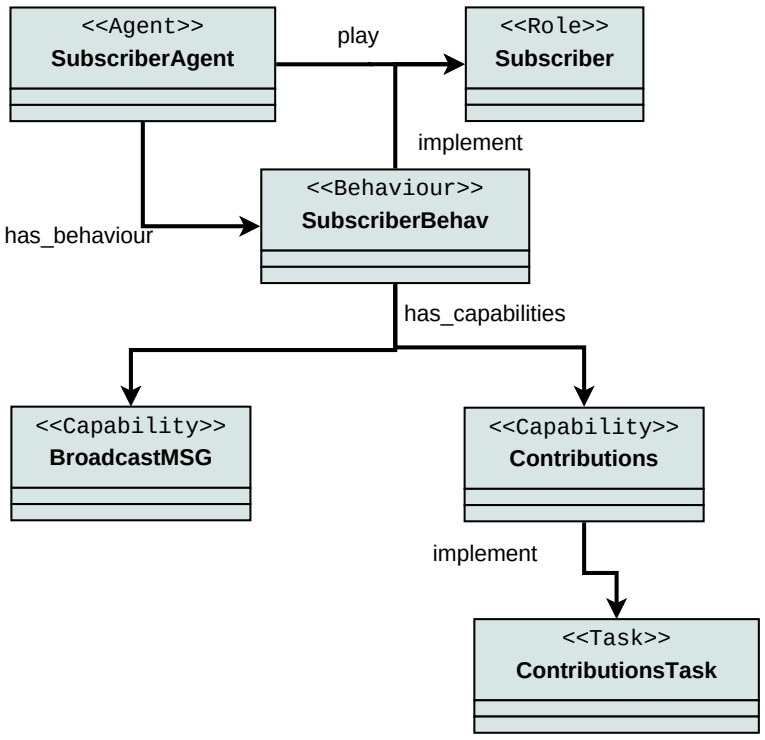

Figura 6.5: Modelo de los Agentes subscriptores (los no miembros)

El ***BroadcastAgent*** se diseña para jugar el rol de *Broadcast*, tal y como se muestra en la Figura 6.6. El *BroadcastAgent* posee un único comportamiento *BroadcastBehav* que le permite proveer los servicios que se modelan en la unidad *News*. El comportamiento *Broadcast* tiene tres capacidades *Contributions*, *BroadcastMSG* y *EInstitution*. La primera capacidad *Contributions* permite registrar

y proveer el servicio de contribuciones en la unidad *News*, así que los agentes que realicen las contribuciones, se les almacena su nombre como garantía de que efectivamente han colaborado monetariamente para apoyar la candidatura de un agente al club. Mientras, que la capacidad *BroadcastMSG* es la que se encarga de registrar y proveer el servicio de envío periódico de noticias del club a los *SubscriberAgent*. *BroadcastMSG* envía todos los mensajes que recoge del club de lectura, desde la última vez que hizo un envío. Finalmente, la capacidad *EInstitution* se utiliza para entrar en las salas de conversación del club de lectura. La capacidad *EInstitution* ejecuta los pasos necesarios para que el agente se pueda mover dentro de la institución electrónica: (i) ser admitido en la institución; (ii) entrar a las salas de conversación de libros y revistas, para almacenar todos los mensajes o conversaciones, que posteriormente serán enviados por la capacidad *BroadcastMSG*; (iii) salir de la institución, cuando las salas de conversaciones ya no tengan ninguna información.

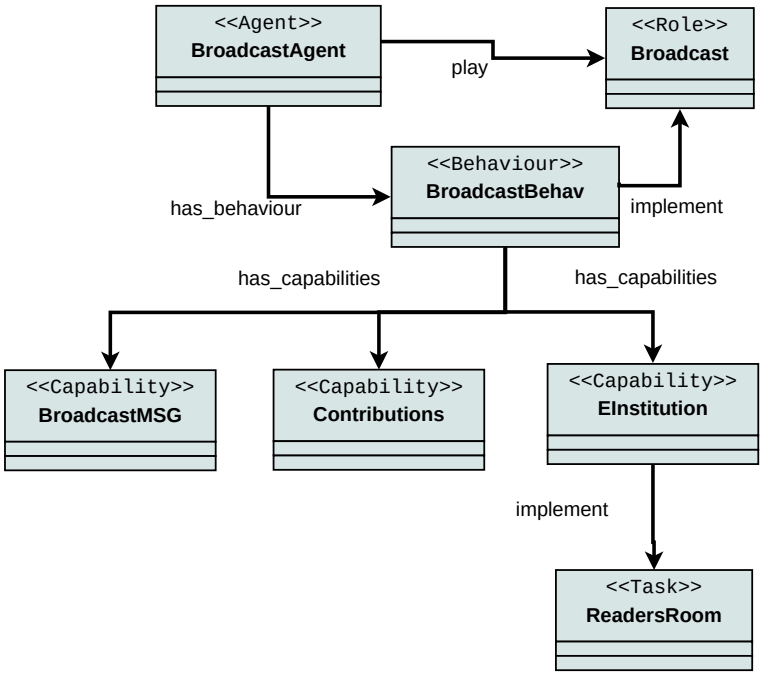

Figura 6.6: Modelo del Agente Broadcast (el miembro autorizado)

Una vez diseñada la organización de agentes foráneos al club de lectura, se debe diseñar la otra organización, el club de lectura, que está soportado por una institución electrónica, tal y como se describe a continuación.

6.2.2. Diseño del club de lectura

En el mundo real, un club de lectura es un grupo de personas que se reúnen periódicamente para hablar sobre un libro que han leído en común. Los encuentros suelen hacerse en torno a una buena cena o comida, aunque hay quienes prefieren quedar en una cafetería, en la biblioteca o en un parque. Cada miembro del club de lectura selecciona un libro y se encarga de coordinar la charla de la reunión, así como de hacer el papel de anfitrión. Existen clubes de lectura temáticos (de historietas, de ciencia ficción, de revistas, de ensayo, etc.) y clubes de lectura generalistas, en los cuales se leen todo tipo de libros.

Para la formación del club de lectura, no existe un número ideal de miembros, sin embargo, es deseable que cada persona tenga la oportunidad de escoger un libro y ser el organizador de una velada al menos una vez al año. Por otra parte, si existe un número muy grande de miembros, esto impediría que los debates y las discusiones se realicen con comodidad y que todo el mundo tuviese un turno para hablar. Conociendo su funcionamiento en el mundo real, se propone el modelo del club de lectura como sigue.

6.2.2.1. Vista Estructural

Nuestro modelo de club de lectura se compone de tres unidades: *ReadersClub*, *BookRoom* y *MagazineRoom*, tal y como se observa en la Figura 6.7. La *ReadersClub* se encarga de recibir a los agentes miembros que tienen un respaldo para entrar al club. Mientras que *BookRoom* y *MagazineRoom* son las salas donde se reúnen los agentes para conversar sobre los libros y revistas respectivamente.

En el club de lectura existen dos roles que los agentes podrán jugar: *Reader* y *Chairman*, como lo muestra la Figura 6.7. Se supone que el agente responsable de organizar la velada cumple el rol de *Chairman*, y éste lleva el control de la sala. Los demás agentes que asisten a la sala juegan el rol de *Reader*.

6.2.2.2. Vista Funcional

Para el funcionamiento (general) de un club de lectura, una vez que se desea formarlo, lo primero que hay que hacer es reunirse para que todos los miembros se conozcan y escojan de qué mes desean encargarse. Aunque todos pueden escoger en esa primera reunión de qué mes desean encargarse, no hace falta que digan qué libro quieren que se lea. Los libros se anunciarán siempre en la sesión del mes anterior, junto con la fecha exacta en que se reunirán al mes siguiente. Por supuesto, alguien se tiene que presentar voluntario para encargarse del primer mes. Deberá anunciar el día, hora y libro al resto del grupo. Este procedimiento debe realizarlo todos los meses la persona encargada de la lectura.

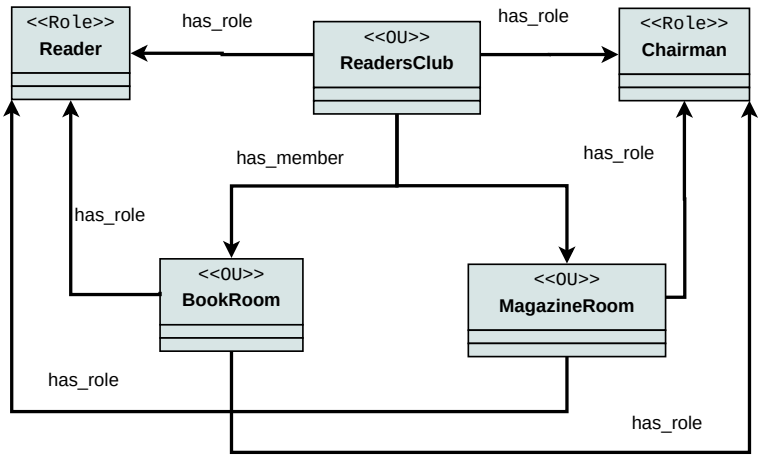

Figura 6.7: Modelo estructural del club de lectura

Para la elección del libro, cada miembro del club, como se ha mencionado, selecciona el libro que se va a leer en el mes que le corresponde a la persona organizar, y lo anuncia en la reunión anterior. Los criterios para seleccionar un libro puede ser como los siguientes: (i) que se encuentre en las tiendas y en las bibliotecas, (ii) el libro escogido tiene ser de interés para el grupo, y (iii) la una longitud del libro debe ser razonable para el tiempo entre cada reunión.

Para simular el funcionamiento del club de lectura, este ejemplo utiliza cuatro servicios compuestos: *Admision*, *SearchRoom*, *Book* y *Magazine* como se muestra en la Figura 6.8.

El servicio *Admision* le permite al club decidir que agentes son aceptados, es decir, cuales son los miembros del club. El servicio *SearchRoom* le permite al agente admitido saber cuáles de las salas de discusión están activas y aun tienen tiempo de entrar para la discusión. En nuestro caso, existen sólo dos salas de discusión por simplicidad para el ejemplo, las unidades *BookRoom* y *MagazineRoom*, pero se podrían replicar las salas si fuese necesario. Los servicios *Book* y *Magazine* permiten implementar los procesos de las salas de discusión para los agentes participantes.

En la Figura 6.8 también se muestran algunos servicios simples que forman parte de los servicios compuestos y las relaciones que existen entre ellos. El servicio *Book* está formado por los servicios *SelectBook* y *DiscussionsB*. En *SelectBook* se realiza la selección del agente responsable de la sala y del libro que será discutido en la próxima reunión del club, mientras, que *DiscussionsB* implementa el debate del libro seleccionado. La información que se genera en estos servicios, es la información y las noticias que transmite el *BroadcastAgent* a los *SubscriberAgent*. Un funcionamiento similar posee el servicio *Magazine*.

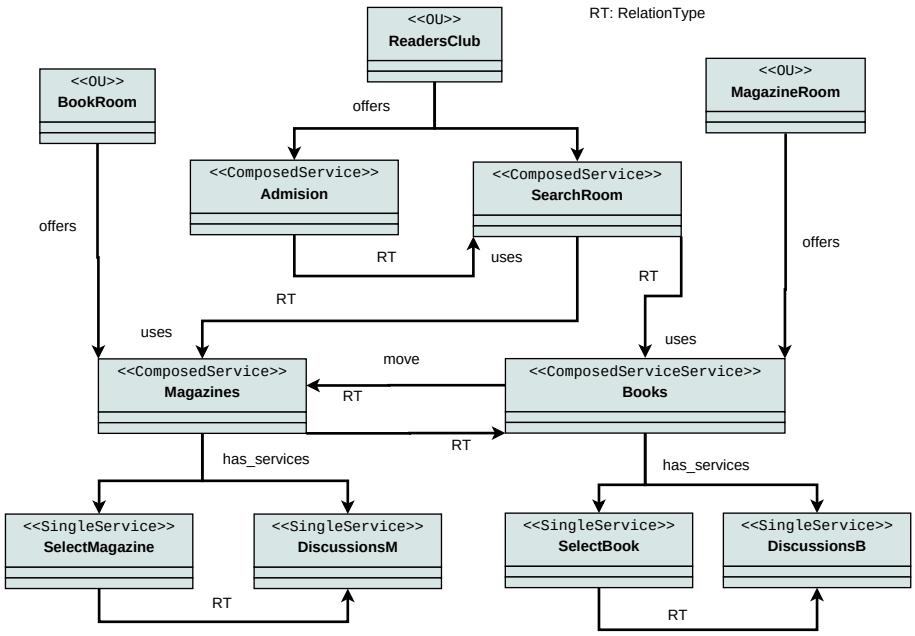

Figura 6.8: Modelo funcional del club de lectura

6.2.3. Transformación de modelos

Una vez que el proceso de modelado del *club de lectura* es realizado con las vistas estructurales, funcionales y de agente (véanse desde la Figura 6.1 hasta la Figura 6.8), es necesario aplicar las reglas de transformación con el fin de obtener las organizaciones en las plataformas de THOMAS y E-Institucions. Además, se deben obtener las transformaciones a nivel de agente, para obtener las plantillas de agentes JADE, que son la base de THOMAS y de la institución electrónica, como se describe en el capítulo 3 y en la bibliografía [11, 10, 9, 5].

6.2.3.1. Nivel de agente

Como ambas organizaciones soportan como base agentes del tipo JADE, las transformaciones a este nivel se destinan a esta plataforma. Si se selecciona como ejemplo, el modelo *SubscriberAgent*, que se observa en la Figura 6.5, que son los agentes que están dentro de THOMAS, se debe transformar cada concepto usado en πVOM al modelo de JADE. Primeramente, utilizamos las reglas de transformación para trasladar los componentes fundamentales del agente: *Agent*, *Behaviour*, *Capability* y *Task*, que están descritas en la sección 3.7.1. Específicamente, utilizamos las Reglas 17, 18, 19 y 20, tal y como se ilustran en la Figura 6.9.

Posteriormente, para traducir el componente *Role*, se debe utilizar la Regla 3 de la sección 3.6.1. Con estas reglas, podemos trasladar el modelo de *Subscri-*

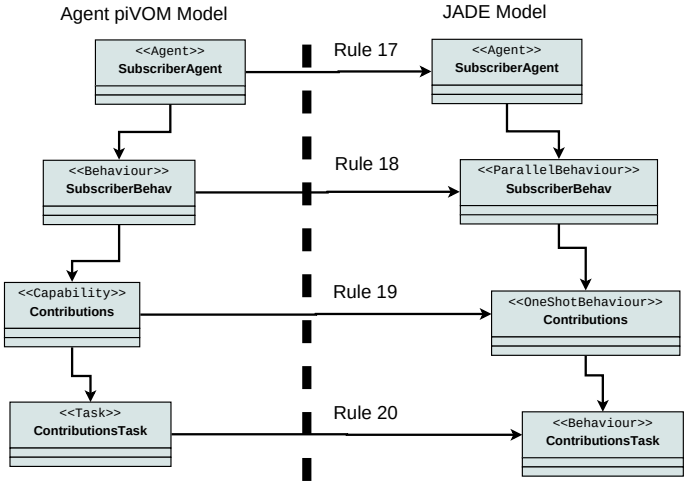

Figura 6.9: Forma de utilizar las reglas en *SubscriberAgent*

berAgent que se observa en la Figura 6.5, al modelo JADE que se muestra en la Figura 6.10.

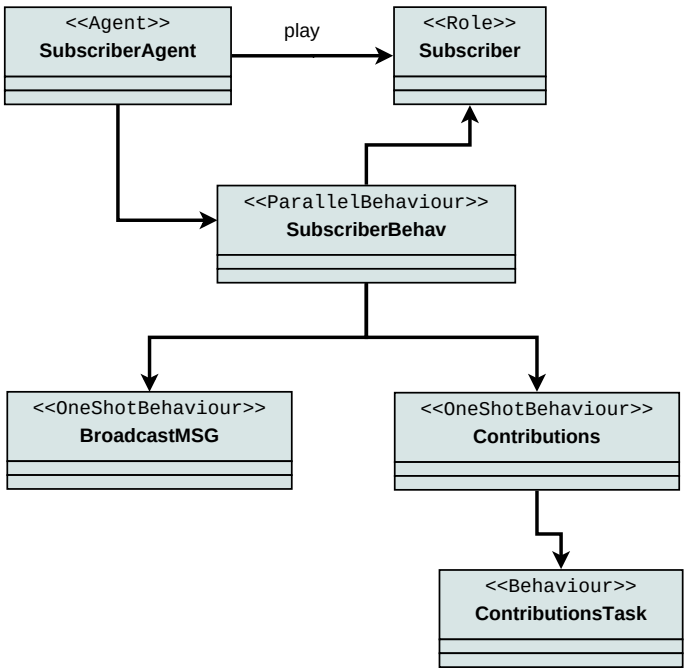

Figura 6.10: *SubscriberAgent* con componentes JADE

Un proceso de transformación idéntico se debe aplicar para el *BroadcastAgent*, obteniendo el modelo que se utiliza en JADE. Sin embargo, no sólo los

agentes se deben trasladar del modelo independiente de plataforma (πVOM) a los modelos específicos de plataforma. También, debemos hacer un proceso similar para obtener los modelos de las plataformas de las organizaciones, como se describe en la siguiente sección.

6.2.3.2. Nivel de organización

En el primer caso se necesita transformar el modelo del grupo que pertenece al club de lectura. Este caso corresponde a la organización que se implementará en la E-Institucions. El objetivo es obtener el modelo de la *PerformativeStructure*. Para ello se requiere la aplicación de una serie de reglas descritas en la sección 3.6.2. En este ejemplo las Reglas 9, 11, 12, y 13, son las más utilizadas.

Comenzamos por analizar los modelos del club de lectura diseñados en las Figuras 6.7 y 6.8. Se observa que en los modelos de entrada existen tres *OU*, que al utilizar la Regla 9, se transforma cada *OU* en una *Scene* de la *PerformativeStructure*, tal y como se observa en la Figura 6.11.

Posteriormente, para conocer como nos movemos entre cada *Scene* de la E-Institucions, utilizamos la Regla 13 (*RelationType*). Para ello, se observan las relaciones entre los diferentes *ComposedService* que ofrece cada *OU*, se analiza el flujo que hay entre los distintos *ComposedService*. Con ello, se obtiene la transición entre las *Scenes*, esta transición corresponde en este nivel de la *PerformativeStructure* a las *Transitions*, tal y como se observa en la Figura 6.11. Es importante comentar que dos *Scenes* de entrada y salida (*Root* y *Output*) deben ser añadidas a la *PerformativeStructure* para obtener un modelo que esté ajustado a la E-Institutions.

Como se mencionó anteriormente, este mapeo genera una plantilla básica de los componentes/conceptos utilizados en la *PerformativeStructure* de la E-Instituciones (Figura 6.11). Sin embargo, con la aplicación de las reglas restantes, se obtiene una descripción más detallada de la misma. Si utilizamos la Regla 11, es posible encontrar los Roles autorizados y permitidos en cada una de las *Scenes*.

Para refinar más el modelo obtenido, se observa en la Figura 6.11 que algunos *ComposedService* tienen servicios simples, como los servicios: *Books* y *Magazines*, cada uno formado por dos sub-servicios. En el meta-modelo πVOM los servicios compuestos están formados por servicios más sencillos, esto es, por servicios atómicos (*SingleService*). Si conocemos el flujo entre estos *SingleServices*, entonces se puede obtener un esqueleto básico de los *States* de cada *Scene*.

Si se observa la Figura 6.11, el concepto *Books* está compuesto de *SelectBooks* y *DiscussionsB*. Un mapeo sobre estos conceptos (aplicar la Regla 12 y la Regla 13) genera la plantilla básica de los *States* La Regla 12 genera cada *State* de la *Scene* a partir de *cada SingleService*. Al analizar el flujo existente entre los *SingleServices* y usando la Regla 13 (*RelationType*), se puede especificar la tran-

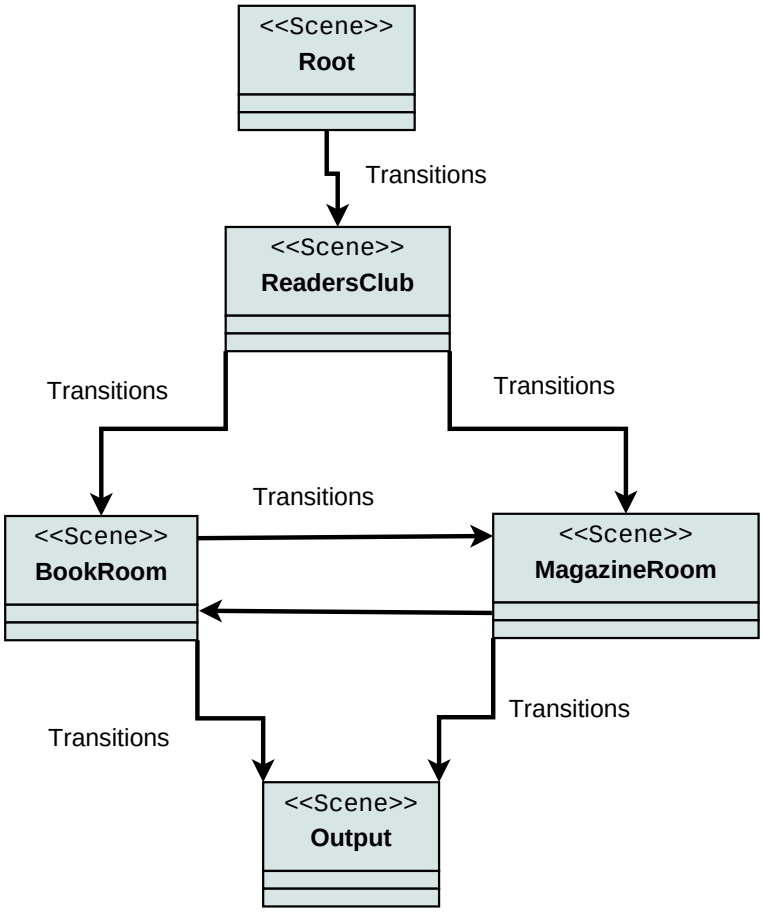

Figura 6.11: Modelo básico de la *PerformativeStructure* del club de lectura

sición entre los diferentes estados, que en este caso corresponden a *Illocutions*, tal y como se muestra en la Figura 6.12. Se deben agregar dos estados adicionales, de entrada y de salida en la máquina de estados para seguir con el esquema de una E-Institution.

Para obtener el modelo de la otra asociación de personas, se necesita transformar el modelo del grupo exterior al club de lectura. Este caso corresponde a la organización que se implementará en el marco de THOMAS, ya que los modelos son muy similares, sus transformaciones son directas (descritas en la sección 3.6.1), y los conceptos tienen una relación de uno a uno. Por lo tanto, los modelos destino de THOMAS son exactamente los mismos vistos en la Figura 6.3 y en la Figura 6.4.

Finalmente, en el último paso, se debe obtener el código de los agentes que son los que inician las organizaciones, y que son los que utilizan sus funcionalidades y servicios. Este código corresponde a plantillas o esqueletos básicos de los agentes,

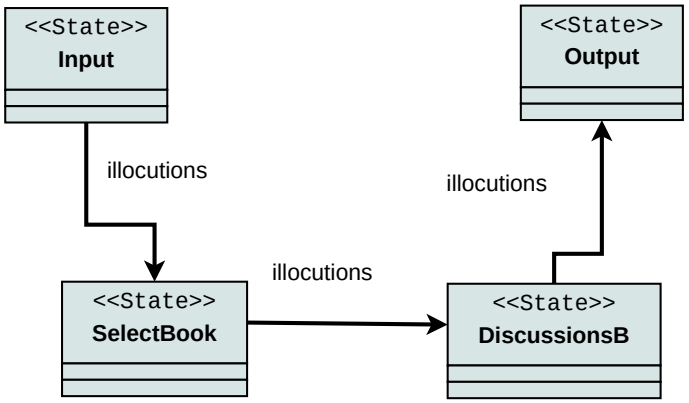

Figura 6.12: Modelo básico de la máquina de estado de una *Scene*

que el diseñador debe complementar con código del desarrollador, el cual se debe agregar manualmente. Para obtener las plantillas de código del *SubscriberAgent*, mostrado en la Figura 6.5, se parte del modelo PSM (basado en componentes de JADE) que se muestra en la Figura 6.10 (que fue transformado como se ilustró en la Figura 6.9). Después, cada componente del agente se puede convertir en trozos o patrones de código (usando la transformación de modelos a código). Así, al combinar los principales conceptos del *SubscriberAgent* se puede obtener la plantilla básica o esqueleto del agente, tal y como se muestra en la Figura 6.13. Un proceso similar se tiene que aplicar al otro modelo de agente, el *BroadcastAgent*.

Este escenario demuestra la viabilidad del meta-modelo propuesto y sus transformaciones para desarrollar MAS orientados a organizaciones. Sin embargo (en este ejemplo), no se han explorado todos los aspectos propuestos en este trabajo de tesis, como el uso de la arquitectura de implementación basada en servicios, el uso de agentes en plataformas que se ejecutan en dispositivos móviles o la utilización de dispositivos físicos (como sensores y actuadores) como servicios para crear ambientes ubícuos. Por esta razón, se presenta a continuación un segundo ejemplo, que permite utilizar con mayor amplitud los aspectos, componentes y enfoques propuestos en este trabajo de tesis.

6.3. Sistema de Transporte Inteligente.

Este escenario tiene como objetivo mostrar el enfoque completo de este trabajo, es decir, el uso de los modelos para diseñar organizaciones de agentes y cómo es su implementación en la arquitectura propuesta. El *Sistema de Transporte Inteligente* permite a los usuarios móviles (pasajeros) conocer los servicios que ofrece la Agencia de Transporte público basado en las preferencias del usua-

```
public class Subscriber extends Agent {
    ...
   protected void setup(){                                    Behaviour
      ParallelBehaviour SubscriberBehav =
         new ParallelBehaviour( ParallelBehaviour.WHEN_ALL );

      SubscriberBehav.addSubBehaviour( new Contributions(this) );
      ...
      addBehaviour(SubscriberBehav);
    }  // ----------- end setup()
                                                              Capability
      class Contributions extends OneShotBehaviour {
          public Contributions(Agent a) {
            super(a); }

          public void action() {
            ... //check condition == true
            addBehaviour(new ContributionsTask(this) );
          }
      } // ----------- end OneShotBehaviour

     ...                                                          Task
      class ContributionsTask extends Behaviour {
          public ContributionsTask(Agent a) {
             super(a); }

          public void action()  {
              //...this is where the code Task goes !!
          }
      } // ----------- end Behaviour

} // ----------- end class Agent
```

Figura 6.13: Plantilla básica de código de *SubscriberAgent*

rio, es decir, el usuario puede recibir/solicitar información para posibles rutas turísticas, servicios personalizados de noticias, el tiempo estimado de llegada (ETA) del transporte, rutas más cortas a un destino, etc.

El transporte público urbano moderno es cada vez más un importante servicio, utilizado por los ciudadanos en las grandes ciudades. Los objetivos de los gobiernos (locales y centrales), y de la Agencia de Transporte Público, es tomar medidas para persuadir positivamente a sus ciudadanos de dejar de usar sus coches particulares para llegar a los centros urbanos y fomentar el uso del transporte público. Por lo tanto, todos los proveedores de transporte público deben proporcionar un servicio cada vez mejor y de una alta calidad, con la esperanza de atraer a más pasajeros.

La tecnología actual puede ser utilizada para mejorar la calidad del servicio y por lo tanto su atractivo para los pasajeros. Esta tecnología proporciona a los operadores de transporte con información en tiempo real sobre el estado operacional del sistema de transporte público, el cual les permite gestionar más eficazmente su flota, y tomar acciones correctivas si llegasen a ocurrir interrupciones. Además, estos sistemas pueden proporcionar información en tiempo real a los pasajeros, tanto en el vehículo como en las paradas de autobús/trenes, para

formar la base de un *Sistema de Transporte Inteligente*.

6.3.1. Arquitectura del sistema

El *Sistema de Transporte Inteligente* es una aplicación que facilita la interconexión entre personas (ciudadanos, turistas) y proveedores de transporte (autobús, metro, trenes, tranvías), delimitando los servicios que cada uno puede solicitar u ofrecer. El sistema controla que servicios deben ser prestados por cada agente. La funcionalidad interna de estos servicios es responsabilidad de los agentes que lo proveen. Sin embargo, el sistema impone algunas restricciones sobre los perfiles y los resultados de los servicios.

En primer lugar el sistema se describe desde el punto de vista físico. El sistema propuesto proporciona servicios de datos inalámbricos, que permiten a los dispositivos móviles, teléfonos celulares y asistentes digitales personales (PDAs) comunicarse con los diferentes proveedores de servicio. La arquitectura de red proporciona acceso a servicios inalámbricos a los usuarios equipados con dispositivos móviles inalámbricos, a través de un conjunto de puntos de acceso desplegados en puntos clave alrededor de la ciudad (por lo general en paradas de autobús que ofrecen servicios de información). Esta arquitectura está soportada por una serie de estándares de comunicación inalámbrica (por ejemplo: WIFI, Bluetooth) que son utilizados para ofrecer estos servicios a los pasajeros. Este sistema (y la arquitectura de red) se muestra en la Figura 6.14.

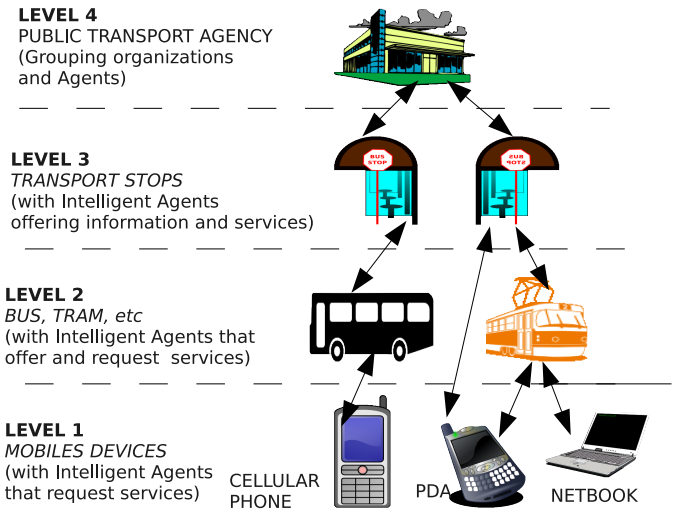

Figura 6.14: Arquitectura del Sistema de Transporte Inteligente

El **primer nivel** se compone de dispositivos móviles equipados con agentes inteligentes que actúan como asistentes personales (PA, de sus siglas en inglés

Personal Assistant) para el pasajero. El PA es utilizado para obtener información del transporte público. Esta información puede ser solicitada directamente por el pasajero, por ejemplo, el usuario puede pedirle a la parada del autobús, la mejor ruta para llegar a un destino específico. Además, el sistema puede hacer recomendaciones adaptadas a los perfiles de usuario, dependiendo de sus preferencias.

Por ejemplo, si el usuario está dentro del bus, el sistema puede ajustar automáticamente el entorno de bus, las noticias diarias mostradas en las pantallas, las cuales, se ajustan de acuerdo a las preferencias (promedio) de los pasajeros. En esta capa sólo pueden coexistir los agentes de los pasajeros (en los dispositivos móviles), como se muestra en la Figura 6.14. Este agente sólo podrá solicitar servicios a la capa superior (a las paradas, autobuses, trenes, etc.), pero ellos no pueden proporcionar servicios a la capa superior. Sin embargo, el sistema puede permitir (si se requiere en otro escenario) que los agentes de los usuarios (pasajeros en este caso) puedan proporcionar servicios.

El **segundo nivel** consiste en los agentes de los vehículos (autobús, metro, etc.), que actúan como un punto de acceso o puerta de enlace para el proveedor de servicio. El agente autobús controla todos los periféricos del vehículo, tales como: máquinas expendedora de boletos/billetes, pantallas para los pasajeros, pantallas de destino, etc., lo que permite al conductor concentrarse plenamente en el tráfico. Este agente autobús se encuentra en el interior del vehículo y sólo puede ser accedido por los pasajeros que se encuentran dentro del autobús.

Además, este agente almacena los datos de la ruta que el autobús está recorriendo, incluyendo la distancia entre cada parada, los nombres de las paradas y los horarios programados del recorrido. El agente autobús muestra en las pantallas el nombre de cada parada donde llega, las noticias del día, el tiempo estimado de llegada y el tiempo (temperatura exterior). Este agente tiene la capacidad de proporcionar servicios a los pasajeros y solicitar servicios de las paradas o de la agencia central.

El **tercer nivel** está ubicado en las paradas del autobús (paradas del transporte público), y muestra a los pasajeros el destino del próximo autobús y su ETA. Este agente puede mostrar información relevante para el viajero, como rutas, noticias diarias. Los pasajeros con su PA (en su teléfono celular o PDA) pueden descargar información relevante del agente de la parada, la cual puede ser leída con detenimiento una vez que se coge el autobús. Además, la pantalla muestra el nombre de cada parada, como el autobús que se acerca y señala en -tiempo real- cualquier desviación del horario programado. Además, este agente se utiliza para rastrear la ubicación verdadera del vehículo, y también puede compensar cualquier error de ubicación.

El **cuarto nivel** está compuesto por la Agencia de Transporte Público que actúa como "contenedor" de las diferentes organizaciones del transporte público

(pasajeros, autobuses, tranvías, paradas, etc.). Pero, su función lógica es permitir la creación y registro de los agentes de los pasajeros, de las unidades de transporte y sus servicios, además de controlar y sincronizar el flujo de información del sistema. Por lo tanto, la función principal del personal del centro de control es garantizar que los autobuses (los transportes) pasan a tiempo, y que los servicios sean eficientes y confiables. Deben asegurarse de que todas las interrupciones del servicio sean rápidamente identificadas y las acciones correctivas sean tomadas tan pronto como sea posible.

6.3.2. Escenarios del sistema de transporte

El sistema de transporte está organizado de tal manera que, cuando un usuario móvil entra dentro del rango de unos proveedores de transporte público (autobús y paradas), el agente inteligente instalado en el dispositivo empotrado permite al proveedor descubrirse mutuamente entre ellos. Esta interacción del agente con la estructura del sistema de transporte (desde el punto de vista físico) se ilustra en la Figura 6.15.

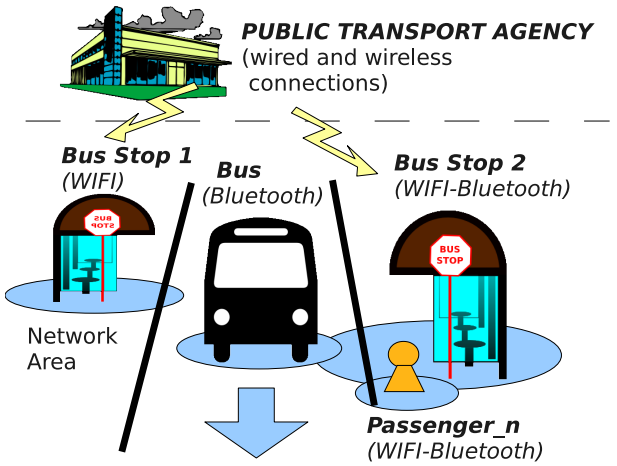

Figura 6.15: Estructura del Sistema de Transporte Inteligente

Esta interacción plantea por lo menos tres escenarios de uso, que dependen del rol jugado por los agentes en el sistema, es decir, existen casos donde un determinado agente juega el rol de proveedor de servicio, pero en otro momento ese mismo agente tiene el rol de cliente del servicio.

Un **primer escenario** supone que existe un pasajero que se dispone usar el transporte público y por ello se desplaza a una parada. En este caso, el pasajero utilizará los servicios (rol de cliente) que proporciona la parada (rol de proveedor) con el objetivo de mejorar su experiencia como usuario. Esta relación cliente-

servidor entre agentes típicamente está controlada por una secuencia de eventos y hechos, que puede ser resumida en los siguientes apartados:

- Un pasajero equipado con un teléfono inteligente está caminando hacia una zona o parada de transporte. El teléfono inteligente posee el software del sistema de transporte (basado en agentes), mientras que los proveedores de servicios (autobús, metro, tranvía, etc.) están equipados con la infraestructura necesaria para soportar a dicho pasajero, y forman nodos individuales en la red del sistema de transporte inteligente.

- Al entrar en la área de la red, un número de procesos arrancan. En primer lugar, la parada (su agente empotrado) detecta la presencia de un pasajero (PA o el agente de usuario). Como hay una lista de servicios que puede ofrecer la parada, el agente de la parada solicita al PA los perfiles de usuario, para comenzar primeramente un proceso de registro y, a continuación el análisis del perfil.

- El Asistente Personal (PA) envía la información del usuario. Esta información incluye una descripción del dispositivo móvil que se está utilizando y las preferencias del usuario. El agente de la parada solicita asistencia (o migra) al nodo de red de la Agencia de transporte, con el perfil del usuario.

- Cuando la Agencia de Transporte da su aprobación y el usuario es registrado con éxito. El perfil de usuario es analizado por la Agencia de Transporte para conocer las actuales preferencias del usuario y las capacidades del dispositivo. En ese momento la Agencia de Transportes compila una lista de servicios y productos aplicables a partir de su catálogo de servicios y productos.

- Posteriormente, la Agencia de Transporte envía los resultados al agente de parada para su conocimiento (o el agente de la parada migra de vuelta con los resultados de la parada donde salió originalmente).

- El Asistente Personal muestra la información sobre estos servicios y productos para que el usuario haga una elección y seleccione (genere una solicitud de) los servicios/productos que desea utilizar. El PA envía la petición de servicio del usuario a la parada, lo que crea una instancia del servicio.

El **segundo escenario** supone que el bus actúa como cliente y la parada es el proveedor del servicio. Los servicios usados en este escenario permiten la sincronización de los ETA del sistema, debido a que las paradas hacen un rastreo de los buses (o de los transportes del sistema: tranvías, trenes) para conocer su posición real. Así, se puede calcular el ETA verdadero y hacer las correcciones

correspondientes. Finalmente, el **tercer escenario** especifica que el usuario está dentro del transporte (autobús, metros, etc.) y está usando los servicios que proporciona la unidad de transporte, como los servicios de notificación de paradas, de información, noticias y sitios interés o turísticos, etc.

De forma similar al primer escenario existen una serie de hechos y eventos que disparan distintas actividades de operación de los agentes clientes y servidores. Pero se ha omitido debido a que su descripción es muy similar a la del primer escenario.

6.3.3. Transporte inteligente como una sociedad de agentes ubícuos

El entorno es una noción que es fundamental en AmI [25] o en computación ubícua. En este caso, en el transporte inteligente, el entorno físico está compuesto de una infraestructura extensa y rica de dispositivos y componentes electrónicos ocultos (y efectos) que pueden diferir significativamente. Así, podría ocurrir, que determinados entornos geográficos estén dotados de ciertas tecnologías, pero las aplicaciones que se utilicen no sean compatibles con ella. En esencia, se trata de una cuestión de interoperabilidad, pero cómo garantizar que todas las aplicaciones AmI sean interoperables es una tarea difícil, si no imposible[195].

Una primera manera de resolver los problemas interoperables, es la adopción de una rigurosa y formal normalización, sin embargo, aunque este enfoque puede parecer prometedor, hay algunas críticas[145, 62]. Este proceso posee su propia rigidez, que no le permite responder lo suficientemente rápido a la evolución de los avances tecnológicos y comerciales. Un segundo enfoque, es una solución más flexible, la interoperabilidad se trata de forma similar a los tradicionales servicios Web en Internet. Este enfoque permite utilizar diferentes tecnologías, sin embargo, con la adopción de tecnologías abiertas (y estándares, por ejemplo, protocolos de comunicación estándar) se reduce la gravedad de los problemas de interoperabilidad que puedan surgir en algún momento en el futuro.

Por lo tanto, el entorno del AmI soportará una serie de servicios, y una funcionalidad que puede cumplir con la composición de los servicios elementales en servicios sofisticados, que en última instancia, mejoraría la calidad de la experiencia de los pasajeros. Por último, existen tres características fundamentales del entorno AmI [47, 139], estas características son los mecanismos de promoción de los servicios, los mecanismos de acceso y los mecanismos inteligentes utilizados en el ambiente. La Figura 6.16, muestra como los mecanismos (y tecnologías) son integrados e interactúan en el entorno.

- El mecanismo de promoción, es un mecanismo donde el entorno se percibe como un mediador o facilitador del servicio de transporte. El entorno de

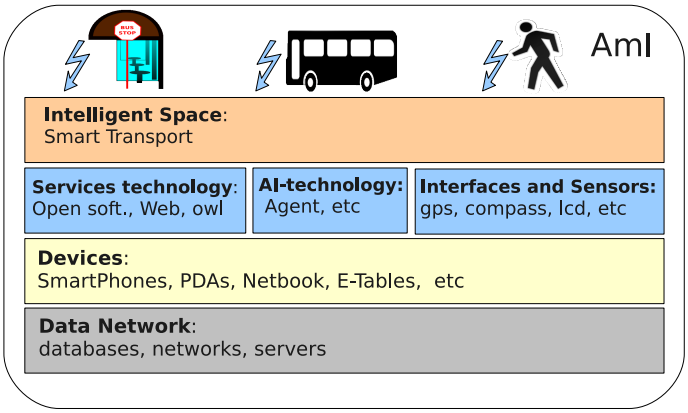

Figura 6.16: Sistema de Transporte Inteligente como un AmI

AmI mediaría entre los actores principales, normalmente, los pasajeros, paradas y vehículos de transporte. Las paradas pueden ser consideradas como un componente fijo del entorno AmI, los vehículos de transporte y los pasajeros forman un componente móvil o transitorio. El AmI tiene un modelo dinámico de los pasajeros y los vehículos de transporte, podemos buscar ajustar los requisitos del pasajero y a la infraestructura del transporte. Los pasajeros pueden suscribirse y usar los servicios. Sin embargo, el entorno AmI también tiene algunas características de proactividad, ya que no sólo es responsabilidad del pasajero iniciar todas las transacciones. Las paradas o vehículos de transporte pueden arrancar algunas operaciones para iniciar los servicios.

- El mecanismo de acceso, el AmI debe proporcionar una interfaz abierta para la infraestructura de transporte, así como facilidades para su uso. Se incluyen interfaces de sensorización, las cuales mejoran la monitorización del entorno. El pasajero tendría la opción de usar estas interfaces para comunicarse ya sea con las paradas o con los vehículos de transporte, que disponen de un servicio que el pasajero desea utilizar. Por lo tanto, el AmI puede proveer de interfaces inteligentes e intuitivas cuando sea necesario. Estas interfaces son naturalmente deseables, pero su implementación es una actividad no trivial:

  - Uso de PDAs estándar o teléfonos inteligentes, (muchos usuarios están familiarizados con ellos), la interfaz de software se ubica en el dispositivo del pasajero. La principal dificultad es: ¿cómo se puede garantizar que el software se ejecutará en la más amplia gama de dispositivos, dado la gran cantidad marcas y modelos de diferentes dispositivos que existen en el mercado.

  - El uso de interfaces estándar que están dispersas en todo el entorno, por ejemplo, mediante terminales fijos o pantallas LCD en el interior de los vehículos de transporte o en las paradas (en ambientes exteriores), que soportan la información multimedia. Estas interfaces realmente no complementan el carácter ubícuo del sistema, sino que son gestionadas por el software inteligente.
  - Utilizar sensores empotrados estandarizados, por ejemplo GPS, que podrían estar dispersos en todo el entorno, para habilitar una exitosa interacción intuitiva y transparente. La implementación de una buena interacción es, por supuesto, una función de la aplicación sistemática de la tecnología conocida como HCI (Human-Computer Interaction).

- Los mecanismos de inteligencia, el AmI implica el uso de software inteligente. Sin embargo, la tecnología clásica de inteligencia artificial, es computacionalmente costosa, por lo que requiere hardware bastante sofisticado. A pesar de que los avances en la tecnología móvil han sido significativos en los últimos años, no son todavía suficientes para ejecutar aplicaciones basadas en inteligencia artificial. Pero hay una excepción, los agentes inteligentes han demostrado que pueden ejecutarse en dispositivos móviles, PDA y teléfonos inteligentes[98, 97]. Esos agentes se conocen como agentes empotrados. Así, en el transporte inteligente (escenario AmI), la inteligencia puede ser distribuida en los componentes fijos de la arquitectura de red, por ejemplo, en las paradas (de autobús, metro, tren, etc), y en otro componente móvil, por ejemplo, vehículos de transporte (autobuses, tranvías, etc) y pasajeros. Estos agentes inteligentes tienen las características de autonomía, reactividad, proactividad y la movilidad que ha demostrado ser beneficiosa en sistemas AmI[9, 63].

6.3.4. Servicios móviles y ubícuos

Los tipos de servicios ofrecidos por el sistema se pueden agrupar en tres categorías[36, 70, 138], en función del tipo de servicio, del rol del cliente y del proveedor, que puede cambiar: (i) Servicios de localización, (ii) Servicios de información instantánea (información en tiempo real), y (iii) Servicios adaptativos.

- **Servicio de Localización**, es un servicio ofrecido por el sistema para la detección de la posición y el rastreo de los vehículos. El sistema está diseñado para rastrear los autobuses y tranvías que hacen recorridos regulares a lo largo de las rutas conocidas dentro de las áreas urbanas. Al comienzo de cada recorrido la posición del autobús se confirma automáticamente por

una baliza de localización. Una vez que el vehículo ha comenzado, su posición se rastrea a lo largo de la ruta mediante la medición de la distancia recorrida. En este caso, la parada de autobús (o la baliza en la carretera), actúa como proveedor de servicios, ya que pueden monitorizar la posición de los autobuses en tiempo real, y comparar sus posiciones reales con la planificada y tomar las acciones apropiadas en el caso de problemas. En contraste, el agente del Bus y el PA actúan como clientes del sistema, ya que reciben los cambios de itinerarios y rutas que son proporcionados por las paradas.

- **Servicio de Información en tiempo real**, muestra información continuamente para el usuario (a los pasajeros que esperan en las paradas o en el interior del transporte público). Esta información puede presentarse en dos formas, visibles y audibles, y es actualizada en tiempo real. Por lo general, las pantallas de las paradas de autobús muestran información de: número del autobús que se aproxima, el ETA u hora programada de llegada, las conexiones, los mensajes de interrupciones del sistema e información especial (noticias de la actualidad o de última hora).

- **Servicio de Adaptación**, es otro servicio que adapta el "entorno" de acuerdo con las preferencias del grupo, utilizando un análisis de los perfiles de los usuarios de los pasajeros. Por ejemplo, se puede adaptar el tipo de noticias que se muestran en la pantalla LCD (pantallas situadas en las paradas de metro o parada de autobús) de acuerdo con las preferencias de los pasajeros, a través de un proceso de votación, se pueden mostrar una mayor cantidad de noticias deportivas si el grupo de pasajeros tiene estas preferencias. Además, es posible que la temperatura en el interior de las unidades de autobús o metro pueda ser ajustada a un valor promedio (del valor deseado por el grupo), controlando el acondicionador de aire de la unidad.

Una vez que describimos los tipos generales de servicios del transporte inteligente, podemos centrarnos en los servicios específicos que se proporcionan a los pasajeros por parte de las paradas del transporte público o por las unidades de transporte (vehículos: autobuses, metros, trenes, tranvías, etc.).

6.3.4.1. Servicios en las unidades de transporte

En primer lugar describimos los servicios que ofrecen los vehículos de transporte cuando el pasajero está dentro del autobús, tranvía, etc. Estos servicios son: (i) Noticias personalizadas, (ii) Pagos, (iii) Notificación de paradas, (iv) Puntos de interés (POI) y (v) Climatización.

- **Noticias personalizadas**, proporcionan información (noticias principalmente) en las pantallas del transporte adaptadas a los perfiles de los usuarios, es decir, a sus preferencias y gustos que son conocidos. Este servicio se puede ilustrar con la Figura 6.17, donde el agente del transporte recibe la información de la preferencia de los usuarios, y por un proceso de "voting"[70], se seleccionan las noticias más relevantes a ser mostradas en las pantallas.

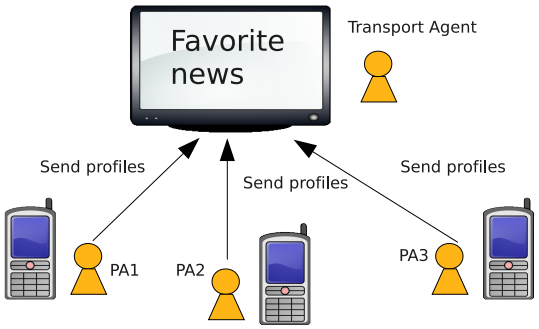

Figura 6.17: Servicio de Noticias personalizadas

Además, el servicio también permite descargar en el terminal móvil del usuario las noticias más relevantes, para que el usuario las pueda leer después con más calma y no en los LCD de la unidad de transporte. Finalmente, la Figura 6.18 muestra el protocolo (simplificado) utilizado por este servicio.

- **Pagos**, sirve para automatizar el pago del billete al subir al autobús o tranvía. Este servicio permite que el agente del usuario pague el importe necesario cuando el agente de la unidad de transporte lo solicita, usando su terminal móvil (por ejemplo su *smartphone*). Este servicio supone que el agente PA posee los billetes necesarios (billetes del transporte) para pagar el importe del autobús. El billete fue adquirido por el usuario del *smartphone* en la Agencia de Transporte (utilizando los servicios Web del sistema) y que los tiene almacenados en la base de datos del PA. La Figura 6.19 muestra el protocolo (simplificado) utilizado por este servicio.

- **Notificación de paradas**, le indica al PA que la parada de su destino es la próxima. Esta indicación es por un mensaje en pantalla del dispositivo móvil del usuario y con una indicación sonora o vibratoria (si el dispositivo lo permite) que permite alertar al usuario.

- **POI** (de sus siglas en inglés, Point Of Interest), es un servicio de notificación de los puntos de interés (por ejemplo sitios turísticos, restaurantes,

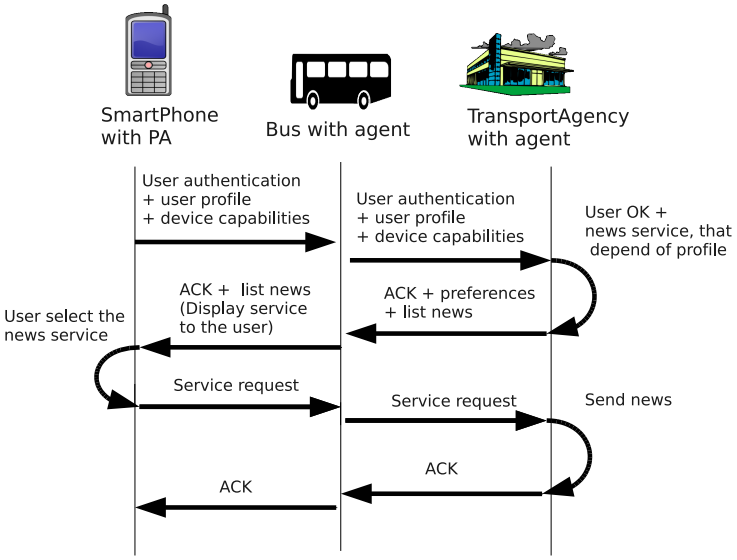

Figura 6.18: Protocolo del servicio de noticias personalizadas

sitios de entretenimiento, etc.) cercanos a cada parada o estación. Este servicio como otros es opcional de ser usado por PA, y permiten informar al usuario lugares interesantes de visitar en la ciudad. Estos POIs son de muy fácil acceso, al utilizar el transporte público y al bajar en una parada especifica.

- **Climatización** es un servicio novedoso ya que permite ajustar el sistema de climatización (aire acondicionado o calefacción) en función de las preferencias (gustos) de los usuarios dentro de la unidad. Esto se realiza de forma similar que el servicio *Noticias personalizadas*, leyendo los perfiles de los usuarios para conocer el gusto sobre la temperatura preferida. El agente del vehículo de transporte solicita la temperatura a los PA (en cada parada), y sacando un simple promedio se controla la temperatura a la que se ajusta el termostato.

6.3.4.2. Servicios de las paradas

En este punto se describen los servicios que brindan las paradas. La cantidad de servicios que se ofrecen pueden variar en función del entorno físico donde se ubique la misma, si es un entorno interno o externo. Estos servicios son proporcionados por el agente de la parada al PA, y son los siguientes: (i) Noticias personalizadas, (ii) ETA, (iii) Buscar ruta, y (iv) POI.

- **Noticias personalizadas**, ajusta la información en las pantallas de las

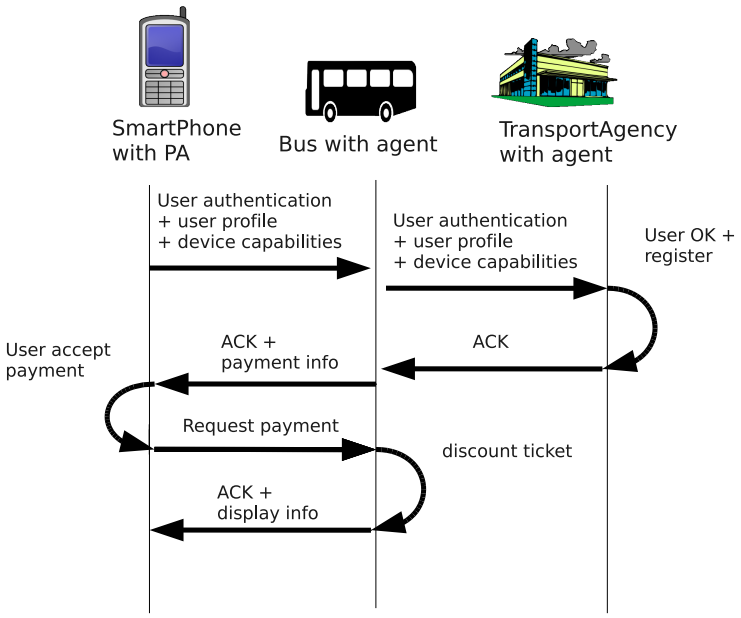

Figura 6.19: Protocolo del servicio de pagos

estaciones en función de la perfiles de los usuarios, de manera similar al servicio que se presta dentro de los vehículos de transporte. Además que permite descargar las noticias preferidas (según el perfil del usuario) al terminal móvil.

- **ETA**, informa que unidades de transporte se están acercando a la parada y además muestra los tiempos aproximados de llegada de las unidades.

- **Buscar ruta**, permite calcular una ruta de transporte al usuario, desde la parada donde se encuentra hasta su punto de destino, indicando los posibles transbordos que pueda necesitar. Esto lo realiza planificando una posible ruta que el usuario deberá seguir para alcanzar su destino. Este servicio utiliza técnicas de planificación[138]. Donde se ofrece un rastreo y una Planning/Scheduling de los recorridos del transporte, y que fue adaptado a nuestro enfoque.

- **POI** es un servicio que informa al usuario de los puntos de interés (por ejemplo sitios turísticos, restaurantes) cercanos a cada parada o estación. Este los indica en el dispositivo móvil del pasajero, describe la dirección a seguir (la ruta) para llegar al punto de interés.

6.3.5. Modelos del sistema de transporte inteligente

Nuestra propuesta definió en el capítulo 3 un conjunto de modelos que proporcionan los conceptos y componentes necesarios para describir el transporte inteligente como una organización virtual. Este conjunto de modelos se basa en π**VOM**[5] y está dividido en cinco vistas diferentes: la estructural (topología), Funcionalidad (Dinámica), Entorno, Agentes y los comportamientos normativos de la organización. Por lo tanto, la definición de Transporte Inteligente corresponde a la identificación de agentes, funciones, unidades organizacionales, normas, etc., de acuerdo a las vistas propuestas.

6.3.5.1. Vista Estructural

Este caso de estudio se modela como una organización (*TransportAgency*) en cuyo interior hay dos unidades organizacionales (*StopUnit* y *TransportUnit*) que representan el grupo de agentes. Cada unidad está dedicada a representar las *Paradas* o *Estaciones* y los *vehículos de transportes*, respectivamente. Dos clases de roles pueden interactuar dentro de la Agencia de Transporte, estos son: *Client*, *Provider*. El rol *Client* solicita los servicios al sistema y el rol *Provider* tiene como misión proporcionar los servicios.

El modelo de organización (modelo de la agencia de transporte), que permitirá la implementación del Sistema de Transporte Inteligente (como una estructura organizacional que ofrece servicios) es mostrado en la Figura 6.20, con sus unidades, roles y las relaciones entre ellos (se describen las entidades como componentes UML y sus correspondientes estereotipos).

6.3.5.2. Vista Funcional

Este modelo especifica los objetivos globales de la organización, los servicios y funciones que ofrece, los objetivos perseguidos por las diferentes entidades de la organización, así como las tareas y planes que se deben seguir para alcanzarlos.

El transporte inteligente puede ofrecer varios tipos de servicios para mejorar la calidad del sistema de transporte público (como se explicó anteriormente en la sección 6.3.4). Estos servicios se utilizan para alcanzar los objetivos de la Agencia de Transportes, entre los cuales están, mejorar las rutas del transporte que operan actualmente y aumentar el uso del transporte público. La Agencia de Transporte ofrece servicios generales, como: búsquedas, información, rastreos de unidades, *Scheduling/Planning* del sistema y venta de boletos. Estos servicios están especializados para cada unidad, de acuerdo con el tipo de proveedor (parada o estaciones y unidades de transporte).

Debido a lo extenso del problema, esta sección se centrará en los servicios que se proporcionan al pasajero. En este caso sólo se muestra la vista funcional de

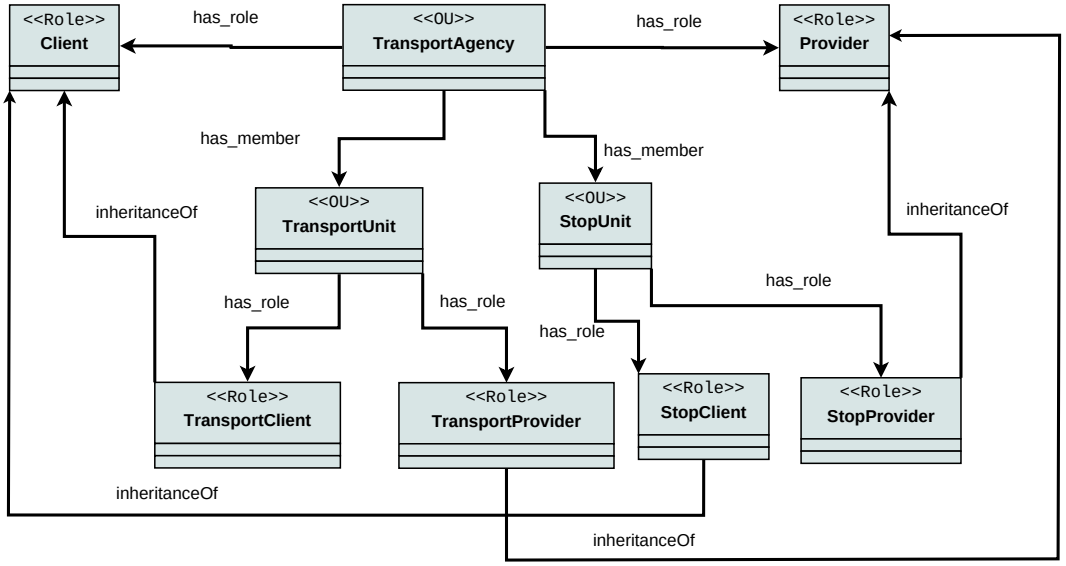

Figura 6.20: Modelo estructural de TransportAgency

la *StopUnit*, se describen los servicios que ofrece y también se indican que roles solicitan y proporcionan estos servicios. La Figura 6.21 muestra dicha unidad con la notación UML. En el modelo de la Figura se observa que la parada proporciona tres servicios: *SearchRoute*, *DisplayInfo*, y *NotificationETA* a los pasajeros o clientes (una vez que toman el rol *StopClient*).

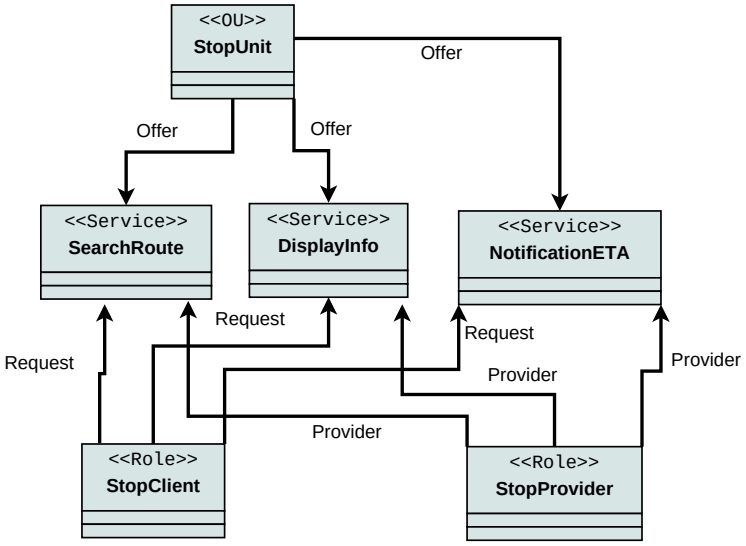

Figura 6.21: Modelo funcional de StopUnit

6.3.5.3. Vista de Entorno

Esta vista incluye los recursos físicos y de software que utiliza la organización, por ejemplo, los proveedores de insumo. También permite describir diferentes ambientes donde las entidades interactúan y donde existen elementos o partes interesadas de la organización. Como también permite acceso a los dispositivos electrónicos en el entorno, como sensores y actuadores. Esta vista detalla los elementos que representa un punto de interacción entre la entidad y otros elementos del modelo, ya que sirve como una interfaz con el mundo real. La Figura 6.22 muestra el modelo de entorno usado en la *TransportAgency*.

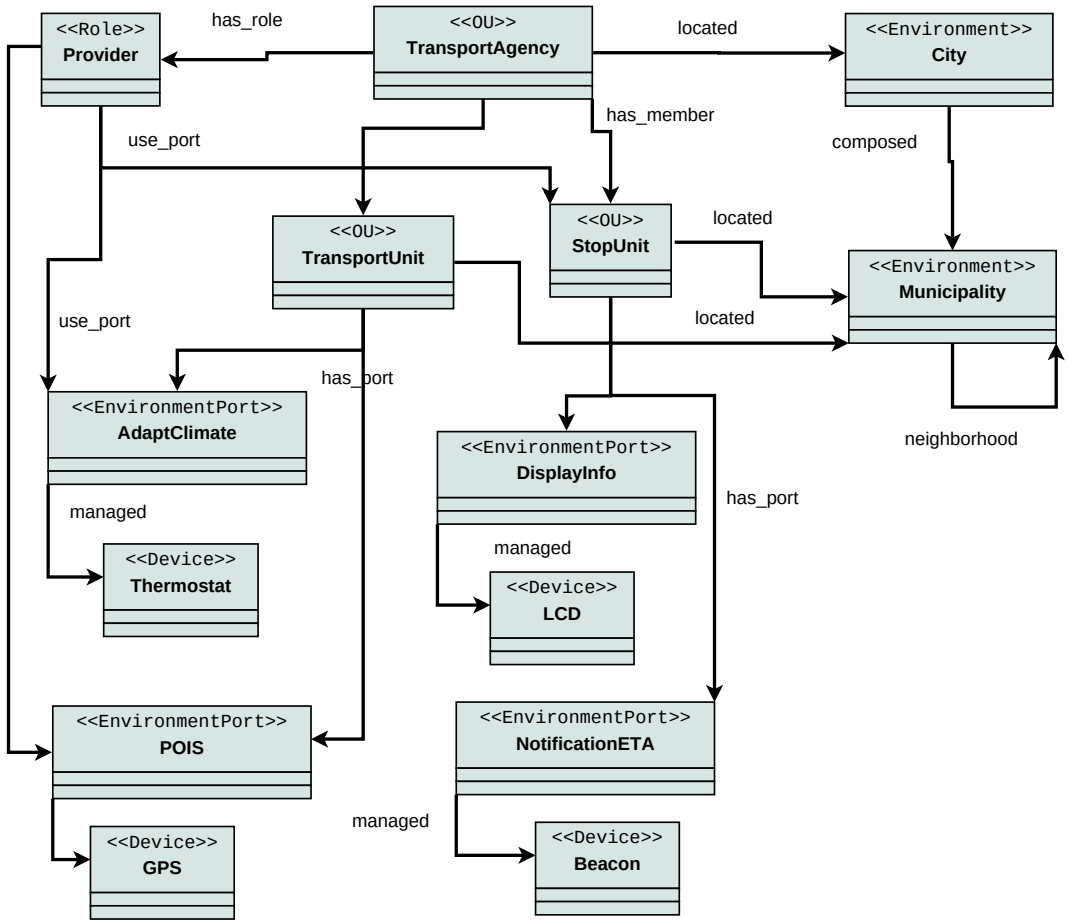

Figura 6.22: Modelo de entorno de TransportAgency

En este modelo las unidades organizacionales estan ubicadas en entornos físicos: en la ciudad o en el municipio (*City* y *Municipality*). Esto permite que los agentes conozcan en que sitio geográficos están. Además, la Figura 6.22 muestra los dispositivos con los que cuenta la parada y el autobús. La parada (*StopUnit*)

acciona una pantalla (*LCD*) que es usada para el servicio *DisplayInfo*. La parada también manipula una baliza (*Beacon*), usada para prestar calcular el tiempo estimado de llegada de los autobuses (servicio *NotificationETA*. De manera similar, el autobús (*TransportUnit*) cuenta con un actuador, un termostato (*Thermostat*) para ajustar el aire acondicionado, y con un sensor *GPS*, utilizado para ubicar los puntos de interés de la zona (servicio *POIS*).

6.3.5.4. Vista normativa

Este modelo asume que la coordinación entre agentes es realizada a través de *normas sociales*. Estas describen el comportamiento esperado de los miembros, es decir, las acciones que están permitidas, las requeridas o necesarias y cuáles evitar. Las *normas* se utilizan como mecanismo para limitar la autonomía de los agentes en los sistemas grandes y para resolver problemas complejos de coordinación. La vista se especifica el conjunto de reglas y acciones definidas para controlar el comportamiento de los miembros de la organización, específicamente para los *Roles* de la organización. La Figura 6.23 muestra el modelo normativo utilizado en *TransportAgency*.

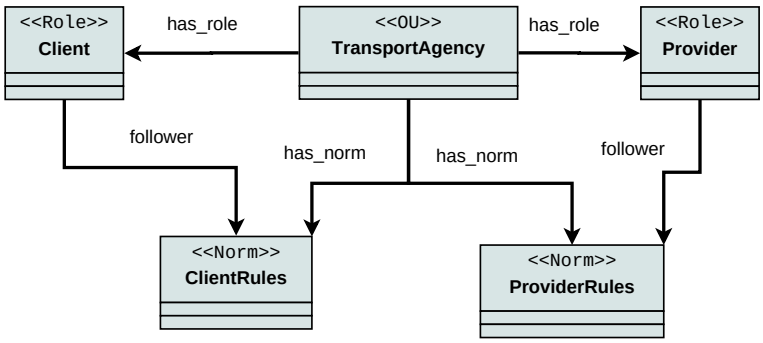

Figura 6.23: Modelo Normativo de TransportAgency

6.3.5.5. Vista de Agente

Un *Agente* es la entidad básica de un MAS, el cual está dentro de la organización y utiliza una serie de protocolos de interacción. La vista del agente tiene un conjunto de componentes relacionados entre sí, cada uno cumple una función específica para la definición del agente. Los principales componentes son: *Behaviours*, *Capabilities*, y *Tasks*[2, 5]. El desarrollo del agente comienza combinando los distintos componentes del modelo del agente, estos componentes son parte de π**VOM**[5]. En esta sección se describen brevemente los principales agentes de

la propuesta: Asistente Personal o agente del cliente, agente de la parada y el agente de la unidad de transporte.

I.) Asistente Personal

El *Agente Personal*, es un modelo de agente que representa a un pasajero en el sistema. Este agente tiene varios comportamientos que permiten la gestión de los servicios ofrecidos por el sistema de transporte. Sin embargo, existen tres comportamientos que son fundamentales, ya que permiten que el agente pueda asumir el rol de cliente, descubrir los servicios que se ofrecen y enviar perfiles. En la Figura 6.24 pueden ser vistos esos tres comportamientos. El *Behaviour*: `NetDiscover`, permite al agente realizar las actividades necesarias para encontrar el agente proveedor de transporte (autobús, tranvía, tren, etc) o agente proveedor de la parada, y comenzar el proceso de registro del agente y, luego, enviar los perfiles del usuario. Mientras que el *Behaviour* (`StopClient`) hace las tareas necesarias para utilizar y solicitar los servicios ofrecidos por la parada (*StopUnit*). De manera muy similar, existe el *Behaviour* (`TransportClient`) que permite manejar los servicios que ofrece el autobús (*TransportUnit*).

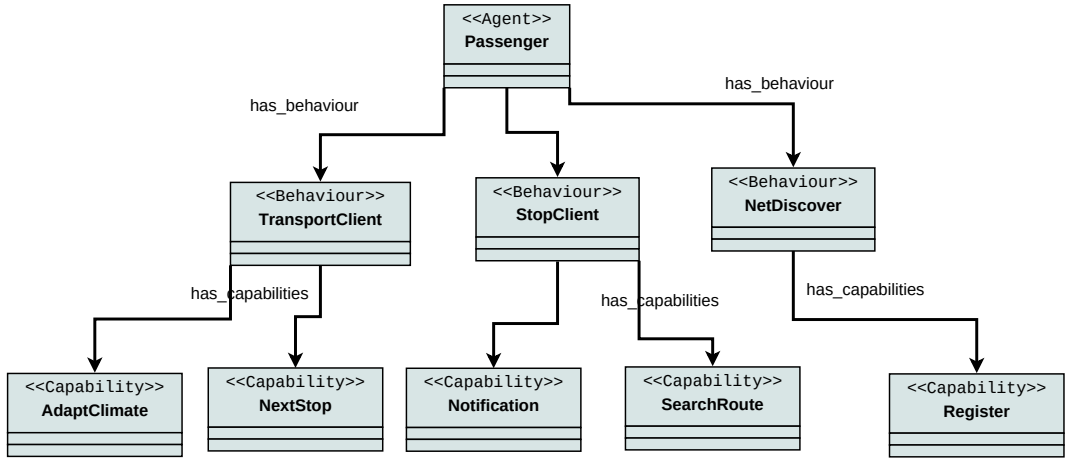

Figura 6.24: Modelo del agente del pasajero (Asistente Personal)

Estos comportamientos están compuestos por una o varias capacidades (ver Figura 6.24), y de ellas dependen los servicios ofrecidos o solicitados por los agentes de la unidad organizacional. Como ejemplo, el *Behaviour* `NetDiscover` posee la *Capability* `Register`. Esta capacidad tiene la responsabilidad de hacer las acciones de buscar a los agentes de los vehículos de transporte (o paradas) y habilitar el registro del agente. Así mismo, el *Behaviour* `StopClient` incluye las capacidades para soportar diferentes servicios de las paradas o estaciones. Esas capacidades pueden ser vistas en la Figura 6.24, donde podemos comentar

la *Capability* `SearchRoute` que permite a los pasajeros solicitar una ruta a un destino en particular usando el transporte público.

II.) Agente de la unidad de transporte

Este es uno de los agentes más completos de este ejemplo, ya que debe cumplir con varios roles al mismo tiempo a lo largo de su ciclo de vida. El agente *Transport* debe comportarse como un agente cliente para actualizar la información que le puedan suministrar las paradas (o también la estación central), como por ejemplo cambio de rutas, notificación de imprevistos en la vía, etc. Además debe proporcionar servicios a los usuarios que viajan dentro de la unidad de transporte.

Entre los comportamientos más importantes cabe mencionar el *Behaviour* `NetDiscover` que le permite encontrar a los clientes y otros proveedores de servicio. Este comportamiento se observa en la Figura 6.25. El *Behaviour* `StopClient` se activa cuando el transporte llega a una estación o una parada y en este momento el transporte adquiere el rol de cliente. Las capacidades de este comportamiento dependen de los servicios que el transporte pueda solicitar. Un servicio importante a solicitar lo implementa la *Capability* `Notification` que permite recibir información relevante para la unidad de transporte y para los pasajeros que transporta.

El *Behaviour* `TransportProvider` activa el rol de proveedor de servicio del transporte. Si observamos en la Figura 6.25 se ven al menos dos *Capabilities*: *NotificationT* y *POIS* (que son capacidades para manejar esos servicios, aunque existen más). La capacidad *NotificationT* sirve para notificarle a los pasajeros a que parada se está aproximando. La capacidad *POIS* permite informar a los pasajeros del autobús de los puntos de interés que existen en la zona.

III.) Agente de la parada

Este agente es el que se encarga de proveer los servicios en las paradas o estaciones (StopUnit) una vez que los usuarios (PA Agent) o los transportes (Transport Agent) establecen comunicación. El modelo del agente en cuestión se observa en la Figura 6.26.

Este agente tiene varios comportamientos pero los más importantes a comentar son los que muestra la Figura 6.26. El *Behaviour* `NetDiscover` que se utiliza para encontrar a los clientes o usuarios (pasajeros y transportes). Otro comportamiento fundamental es *Behaviour* `StopProvider` que le permite al agente jugar el rol de proveedor. Las tres *Capabilities* denotan los servicios que puede ofrecer este tipo de agente, que son *Notification* para notificar los diferentes rutas de transportes que llegan a la parada como su ETA, *Display* que permite ajustar la información que se muestra por pantalla a los perfiles de usuarios y *SearchRoute*

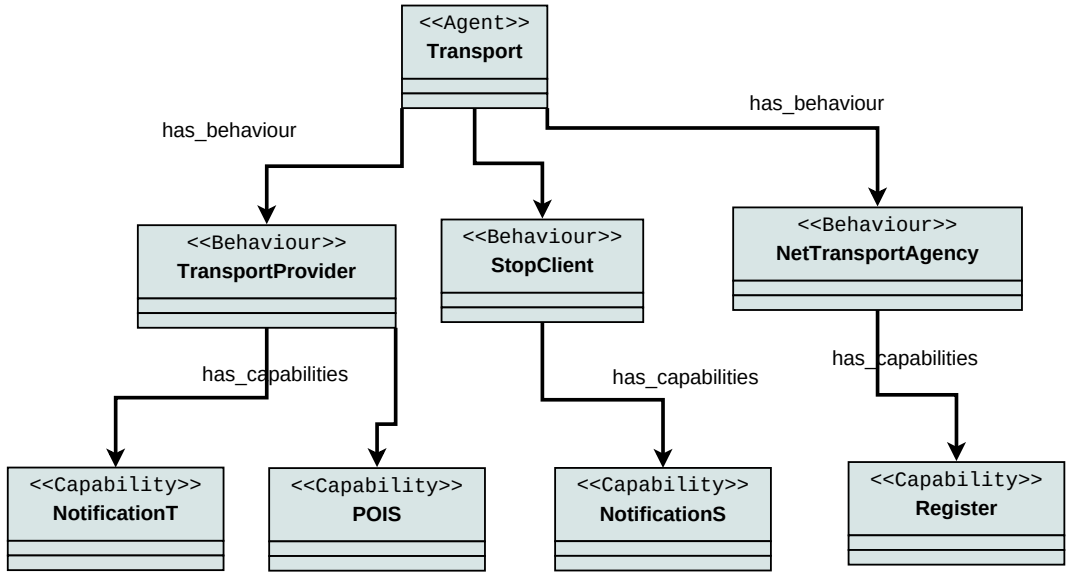

Figura 6.25: Modelo del agente de la unidad de transporte

que permite calcular/planificar una ruta desde la parada a un punto de destino.

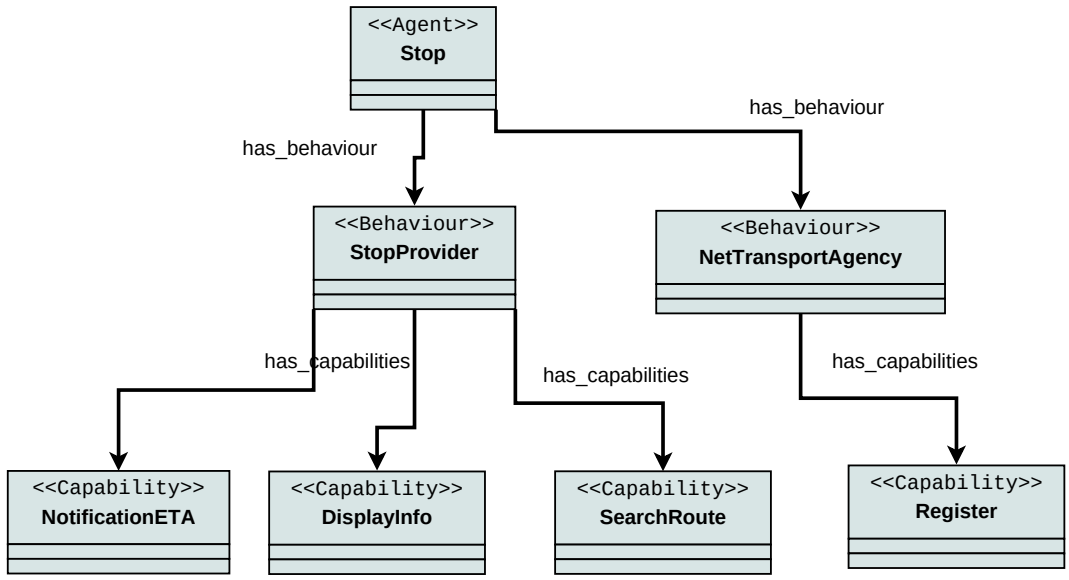

Figura 6.26: Modelo del agente de la parada

Finalmente, antes de terminar esta sección es bueno resaltar que pueden existir uno o varios agentes dentro de cada unidad de la organización y que cada uno puede ofrecer uno o varios de los servicios que proporciona la unidad. Es por ello que un agente no necesariamente tiene todos los servicios de su unidad (a menos

que esté sólo) y es por esto que el agente puede tener más o menos *Behaviours* y *Capabilities*, según los roles que juegue y los servicios que ofrezca o solicite.

6.3.6. Arquitectura de implantación del transporte inteligente

Después de describir el sistema de transporte inteligente, los servicios que puede ofrecer, y los modelos necesarios para su diseño, es necesario explicar algunas plataformas utilizadas para implementar el presente ejemplo. La implementación del transporte inteligente como una organización virtual ubícua, debe permitir la fácil integración de los servicios móviles y ubícuos, que son ofrecidos y solicitados por agentes.

Este sistema necesita de diferentes tecnologías de hardware y de agentes que son descritas sobre la arquitectura de implantación descrita en el capítulo 4. Las paradas y las unidades de transporte como autobuses, tranvías, etc., están diseñados como unidades de una organización virtual que soporta una sociedad abierta de agentes. Los agentes pueden entrar y salir de la organización en cualquier momento. La Figura 6.27 muestra donde se ubican las plataformas de agentes, servicios y demás tecnologías sobre la arquitectura de implantación. A continuación se describen cada una de las capas de dicha arquitectura.

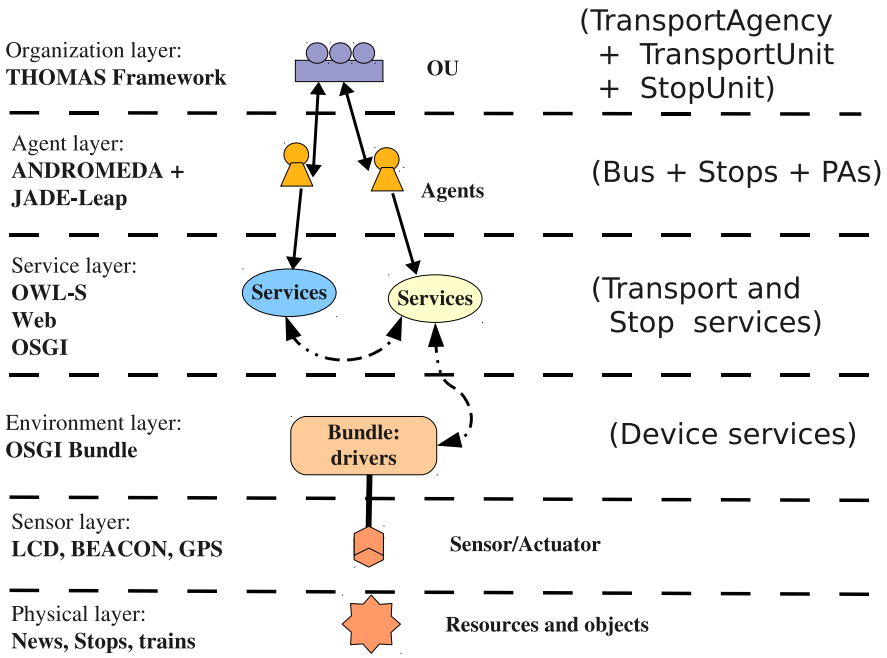

Figura 6.27: Sistema de transporte inteligente sobre la arquitectura de implantación

6.3.6.1. Capa Organizacional

La capa organizacional se implementa con el "framework" THOMAS, el cual fue descrito en el capítulo 2. El uso de THOMAS nos dará las herramientas necesarias para la implementación de las entidades de las paradas, las unidades de transporte y la agencia de transporte en una infraestructura de organización virtual. Esta infraestructura permite a los agentes entrar o salir de forma dinámica de la organización, asignar roles a cada agente e incluir las reglas o normas que los agentes deben cumplir.

6.3.6.2. Capa Agente: plataformas de agentes empotrados

Con respecto a las plataformas de agentes utilizadas, nuestra propuesta utiliza dos, que ya fueron descritas en los capítulos 4 y 5:

- ANDROMEDA que es una plataforma de agentes específicamente orientada a agentes empotrados sobre el sistema operativo *Android*.

- JADE que es una de las plataformas más ampliamente utilizada en la ejecución de agentes, sobre JAVA. También se usa JADE-Leap para implementar agentes empotrados. La Figura 6.28 ilustra las tecnologías de agentes utilizadas y el posible proceso de diálogo e interacciones entre los agentes de los distintos actores del sistema de transporte, que son los pasajeros, las unidades de transporte (tranvía, bus, etc.) y las paradas.

6.3.6.3. Capa de Servicio

Esta capa puede ofrecer varios tipos de servicios para facilitar y mejorar la experiencia del pasajero en el sistema de transporte. Los tipos de servicios que ofrece el sistema de transporte son muy variados, pero se pueden agrupar en tres categorías (dependiendo del tipo de servicio, el rol del cliente y del proveedor): (i) Servicio de Recomendación; (ii) Servicio de Información al instante; (iii) Servicio de adaptación, que son suministrados por las unidades de transporte y las paradas del sistema. La descripción de estos servicios ya fue descrita en la sección 6.3.4.

La implementación de estos servicios está basada en el uso de la tecnología OWL-S, que permite a los pasajeros (PAs), transportes (agentes del autobús) y paradas (agentes de las paradas) automáticamente ser descubiertos, invocar, componer, y monitorear recursos Web que ofrecen otros servicios, bajo ciertas condiciones.

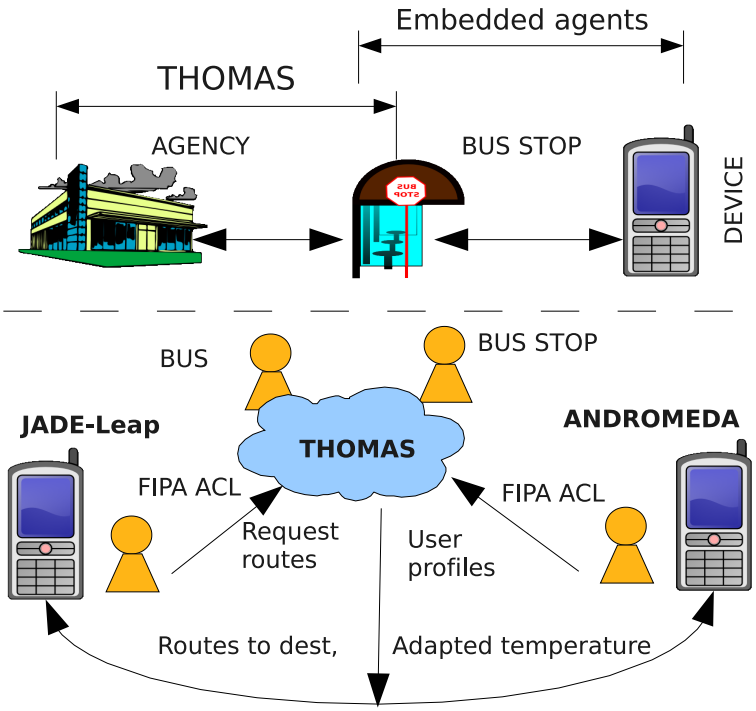

Figura 6.28: Interacción de los agentes en el caso de estudio

6.3.6.4. Capa de Entorno

Esta capa tiene la responsabilidad de encapsular los *drivers* de los dispositivos (que son dependientes del fabricante) en un servicio único y genérico. Esta encapsulación se realiza con la tecnología de OSGi (usando *bundles*). Un *bundle* implementa una interfaz común, con el fin de proporcionar una manera uniforme de comunicación dentro del *driver* de los dispositivos. El *bundle* se ejecuta sobre una tarjeta empotrada (la tarjeta Beagleboard es usada en este trabajo). La Beagleboard es un ordenador de placa única, de bajo consumo y hardware de código abierto. La tarjeta Beagleboard es un ordenador empotrado basado en un procesador ARM de Texas Instruments. Es importante resaltar que las Beagleboard ejecutan a los agentes empotrados desarrollados sobre ANDROMEDA, ya que se pudo portar *Android* a Beagleboard. La Figura 6.29 muestra algunos de los componentes de software/hardware usados en esta aplicación y además donde se utilizan las Beagleboard sobre el ejemplo.

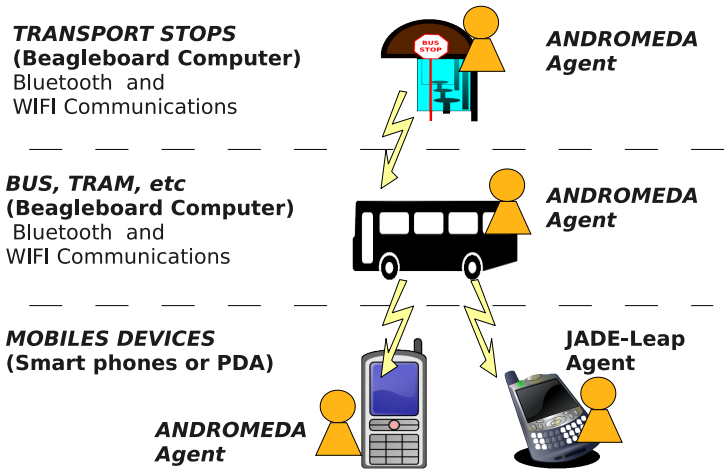

Figura 6.29: Tecnologías en el transporte inteligente

6.3.6.5. Capa de Sensores y física

Estas capas contienen los sensores y actuadores usados en el sistema de transporte, tales como: dispositivos para el cobro del pasaje en el bus o metro, pantallas LCD para mostrar información, GPS, sensores de temperatura, unidades de aire acondicionado/calefacción, balizas de localización, tarjetas NFC o RFID, puertas o cerraduras automáticas, indicadores sonoros, etc. Las unidades de transporte y las paradas poseen una o más tarjetas empotradas que pueden ejecutar un agente empotrado (con ANDROMEDA sobre Beagleboard), cuya responsabilidad es la gestión de sensores. También debemos mencionar que otro componente de importancia son los teléfonos móviles, que ejecutan los agentes empotrados de los clientes, como se muestra en la Figura 6.29. Estos terminales son de dos tipos principalmente, el primero corresponde a teléfonos que están basados en el sistema *Android* (como el HTC One) y por ello pueden ejecutar agentes ANDROMEDA. El segundo terminal que soporte J2ME, y por ello permite ejecutar los agentes del tipo JADE-Leap.

6.3.7. Implementando el sistema de transporte inteligente

Una vez que conocemos los diferentes componentes del sistema, y los modelos de diseño que permiten implementar el sistema de transporte inteligente como organización virtual, el paso siguiente es generar las plantillas de código por medio de las transformaciones de los modelos UML. Estas plantillas le permiten al desarrollador obtener un esquema donde escribir el código final de la aplicación. Para ilustrar este proceso usaremos el agente del pasajero, que es el agente que se ejecuta sobre los teléfonos móviles del pasajero y cuyo objetivo principal es

solicitar los servicios del sistema.

El diseñador debe decidir en qué plataforma móvil se ejecutará cada agente (como se describió en el capítulo 3). En este caso, se selecciona el agente del pasajero a ser ejecutado a través de una plataforma de agentes: ANDROMEDA o JADE-Leap. El marco ANDROMEDA ha sido diseñada utilizando los mismos conceptos del meta-modelo abstracto. Este diseño simplifica en gran medida la transformación entre el modelo PIM y el modelo PSM.

En este caso, las reglas de transformación necesarias son principalmente de modelo-a-texto, obteniendo directamente el código necesario para ANDRÓMEDA, el cual puede ser combinado con código adicional escrito por el usuario. Un ejemplo de la plantilla de código que se puede obtener de las transformaciones se muestra en la Figura 6.30.

```
public class Passenger extends Agent {
     public void init(){
         . . .
      //define agent components
                                                    Behaviour
      Behaviour StopClient= new Behaviour("StopClient");
                                                    Capability
      Capability SearchRoute =  new Capability("Route");

       Task TaskSearch =new Task("TaskSearch");

        . . .
        //add objects into of the agent components
        SearchRoute.addTaskRun(TaskRoute);
        StopClient.add(SearchRoute);
        addbehav(StopClient);
    }  // ----------- end init()
                                                    Task
    class TaskSearch extends Task{
       doIt {
         //here the user write the  code
       }  // ----------- end doIt
       . . .
    }  // ----------- end Class Task

     . . .
}  // ----------- end Class Agent
```

Figura 6.30: Plantilla de código para el agente ANDROMEDA

La transformación del modelo de agente al modelo de código de JADE-Leap debe hacerse usando las dos fases descritas en el capítulo 3: (i) la primera fase se traduce a partir del modelo PIM al modelo PSM de JADE-Leap, y permite obtener una correspondencia entre la conceptos abstractos de los modelos; (ii) la segunda fase permite traducir los modelos PSM de JADE-Leap obtenidos de la

fase anterior a código ejecutable JADE-Leap.

Estas fases se pueden resumir como, que el proceso de transformación traduce el `Behaviour` del agente del pasajero diseñado a un `ParallelBehaviour` de JADE-Leap. Además, convierte cada `Capability` a un `OneShotBehaviour` y cada `Task` a un `Behaviour`. La Figura 6.31 muestra una plantilla ejemplo de esta transformación.

```
public class Passenger extends Agent {
    ...
   protected void setup(){                                    Behaviour
      ParallelBehaviour StopClient =
         new ParallelBehaviour( ParallelBehaviour.WHEN_ALL );

      StopClient.addSubBehaviour( new SearchRoute(this) );
      ...
      addBehaviour(StopClient);
    }  // ----------- end setup()                             Capability
      class SearchRoute extends OneShotBehaviour {
          public SearchRoute(Agent a) {
            super(a); }

          public void action() {
            ... //check condition == true
            addBehaviour(new TaskRoute(this) );
          }
      } // ----------- end OneShotBehaviour
      ...                                                     Task
      class TaskRoute extends Behaviour {
          public TaskRoute(Agent a) {
             super(a); }

          public void action()  {
              //...this is where the code Task goes !!
          }
      } // ----------- end Behaviour

} // ----------- end class Agent
```

Figura 6.31: Plantilla de código para el agente JADE-Leap

Las plantillas de código que se pueden obtener de esta transformación pueden deben ser editadas por el usuario para completar el código del agente pasajero con los detalles necesarios para su ejecución sobre la plataforma final.

Finalmente, una vez descritos los dos ejemplos: el club de lectores y el sistema de transporte inteligente, cabe destacar que nos han servido para mostrar la utilidad de la propuesta presentada en esta Tesis. Los escenarios se han modelado como MAS abiertos utilizando los modelos independientes de plataforma de πVOM. Posteriormente, los modelos unificados son transformados a las plataformas de implementación específicas. Los ejemplos han mostrado como la arquitectura de implantación permite integrar en sus capas distintos dispositivos

de entorno y diferentes plataformas o marcos de ejecución MAS, para que estas tecnologías de naturaleza distribuida puedan combinarse de manera uniforme. A continuación se presentan las conclusiones que se desprenden de este trabajo y las posibles líneas de investigación que pueden ser abordadas en futuros trabajos.

7

Conclusiones y Trabajos Futuros

En este capítulo se presentan las conclusiones que se han obtenido a lo largo de esta investigación. Para ello se presentan las principales aportaciones realizadas, así como, las posibles líneas de trabajo futuras, las cuales, constituyen aspectos que quedan abiertos para posteriores investigaciones.

7.1. Contribuciones Destacadas

Las principales contribuciones obtenidas a lo largo de esta investigación pueden resumirse en una *propuesta para desarrollar organizaciones virtuales ubícuas utilizando el desarrollo dirigido por modelos*. No obstante, el trabajo consta de tres partes bien diferenciadas:

1. La especificación de los meta-modelos abstractos, que permiten diseñar MAS basados en organizaciones de forma general, y transformarlos a modelos específicos de plataforma.

2. La especificación de una arquitectura general de implantación basada en servicios, la cual permite implementar MAS con característica ubícuas.

3. La implementación de un marco de ejecución de agentes sobre dispositivos móviles basado en el sistema operativo *Android*.

En la primera parte (capítulo 3), se analizaron las diferentes metodologías de agentes que soportan sociedades, instituciones y organizaciones, e igualmente metodologías de sistemas ubícuos. Los cuales permiten proponer un conjunto de conceptos y componentes, que se agrupan en cinco meta-modelos, llamado

πVOM: *Estructural*, *Funcional*, *Normativo*, de *Agente* y del *Entorno*. Dichos meta-modelos permiten al desarrollador, modelar una organización virtual basada en agentes usando conceptos abstractos que son independientes de la plataforma de implementación. El diseño se realiza con modelos unificados, intuitivos y visuales, es decir, con un alto nivel de abstracción, ya que se usan componentes UML, siguiendo un enfoque MDD. El meta-modelo de *Entorno* permite que la organización virtual basada en agentes tenga cualidades ubícuas, ya que modela dispositivos físicos del entorno como servicios que pueden utilizar los agentes y las organizaciones, permitiendo que πVOM pueda modelar organizaciones virtuales ubícuas. Además, siguiendo la filosofía MDD, se proponen un conjunto de reglas de transformación que permiten trasladar los modelos independientes de plataforma (πVOM) a plataformas específicas de implementación. A nivel de organización se proponen reglas de transformación para plataformas que soportan organizaciones. A nivel de agente se específica la transformación a dos modelos de agentes, el modelo de agente de JADE y el modelo de agente de Andromeda.

En la segunda parte (capítulo 4), se estudió la problemática de implementación de los sistemas ubícuos y de las organizaciones virtuales, para proponer una arquitectura de implantación o despliegue para la organización virtual ubícua basada en agentes. Después del análisis, se ha propuesto una arquitectura por capas, donde los elementos y entidades del sistema se organizan en diferentes niveles con responsabilidades y funcionalidades bien definidas. Esta arquitectura de despliegue (de implantación) está basada en la tecnología OSGi. Esto proporciona que las organizaciones y agentes soporten la integración de los servicios prestados por los dispositivos externos y por los sistemas de software (como otro de servicios). Además, la arquitectura soporta el aislamiento de la abstracción de bajo nivel de los dispositivos. Con estas características se logra: (i) que los sistemas ubícuos puedan adaptarse de manera fácil a diferentes dispositivos, ya que las dependencias de hardware del fabricante quedan encapsuladas, y (ii) que las funcionalidades del sistema (la organización ubícua) sean más potentes, ya que se basan en la composición de servicios generales y unificados.

En la tercera parte (capítulo 5), se ha descrito una plataforma de implantación de agentes empotrados que recibe el nombre de Andromeda. Andromeda se diseñó con los conceptos del modelo de agente de πVOM, sobre el sistema operativo *Android*. Esto permite que los agentes diseñados en esta investigación puedan ejecutarse sobre una gran gama de artefactos de uso comercial como: teléfonos, tabletas, netbooks, etc., e igualmente en aquellos dispositivos (hardware con capacidades de computación) donde se pueda migrar *Android*, como: computadores o tarjetas empotradas, televisores inteligentes, sistemas multimedia, etc. Además, la plataforma permite que los sensores y actuadores puedan ser añadidos al sistema. Con ello, los agentes se ejecutan en dispositivos empotrados y pueden manipular los sensores del sistema, logrando que los agentes puedan

interactuar y percibir el entorno físico. Esto facilita la implantación de ambientes inteligentes o sistemas ubícuos basados en agentes.

Por lo tanto, podemos resumir las aportaciones de este trabajo en los siguientes puntos:

- Una propuesta para diseñar organizaciones virtuales con meta-modelos independientes de la plataforma de implementación, que es más intuitiva que usando los enfoques clásicos, porque se usan métodos de "model-driven" que permiten que el desarrollador diseñe con herramientas UML evitando los detalles técnicos.

- Una propuesta que permite modelar la organización virtual a través de cinco meta-modelos complementarios, llamado πVOM, que incluyen los componentes fundamentales de las metodologías más usadas en esta área, y que soporta sus funcionalidad en el uso de los servicios.

- Un conjunto de reglas de transformación, que permiten traducir los modelos abstractos de las organizaciones virtuales a modelos específicos (y hasta plantillas de código) de la plataforma final de ejecución, basado en el enfoque de desarrollo dirigido por modelos,

- Un meta-modelo de entorno que permite que las entidades de la organización virtual interactúen y perciban su entorno real, a través del uso de dispositivos físicos, como sensores y actuadores.

- Una arquitectura dividida en capas funcionales que sirve como un marco de implementación de los componentes de la organización virtual ubícua. Lo que facilita la implantación de sistemas ubícuos, ya que las unidades organizacionales y los agentes pueden percibir y controlar su entorno, usando dispositivos físicos, lo que permite crear ambientes con inteligencia basado en MAS.

- Una arquitectura que basa sus funcionalidades en servicios, que son implementados usando la tecnología OSGi, que facilita la composición, búsqueda y resolución de los mismos. Lo que facilita diseñar sistemas ubícuos más novedosos, ya que usa ingeniería de software basada en agentes sobre una arquitectura que soporta servicios (OSGi), que permite que los agentes y organizaciones administren los servicios, el contexto, y la adaptación al entorno.

- Una arquitectura adaptable a nuevos dispositivos tecnológicos de una manera fácil, ya que las funcionalidades de los dispositivos (sensores y actuadores) que son dependientes del fabricante, son encapsulados en servicios genéricos, que no son tan dependiente del hardware.

- El diseño y desarrollo de una plataforma para la ejecución de agentes basados en el modelo de πVOM sobre el sistema *Android*, que llamamos ANDROMEDA. ANDROMEDA nos permite de ejecutar los agentes en una variedad de computadores embebidos, interactuando con dispositivos inmersos en el entorno.

7.2. Trabajos Futuros

A lo largo de este trabajo no se han tratado ciertas ideas que son consideradas interesantes para posteriores investigaciones.

- Debido a la dinámica de los MAS, nuevos modelos y conceptos pueden aparecer con el transcurso del tiempo. Mantener los meta-modelos con conceptos generales y genéricos, es una buena práctica. Sin embargo, actualmente se investiga en diversas tecnologías del acuerdo[156] ("Agreement"), que es un componente muy útil en sociedades de agentes abiertos, su inclusión en los meta-modelos propuestos puede potenciar los problemas que pueden abordarse.

- En el meta-modelo de entorno se puede estudiar y analizar una especificación para soportar la sensibilidad al contexto ("Context-Awareness"), y en el uso de ontologías para describir la información contextual, lo que permite inferir nuevos conocimientos del entorno, componentes muy utilizados en sistemas ubícuos[27, 181].

- La arquitectura propuesta se especificó usando la tecnología OSGi, que es un marco muy potente de ejecución basados en servicios, sin embargo, en este último año se han realizado muchos progresos en la tecnología que facilita la implementación de servicios distribuidos, es decir un OSGi distribuido (DOSGi). DOSGi está soportado por un marco de servicios abiertos de Apache (Apache CXF: An Open-Source Services Framework)[18]. Su utilización en la arquitectura puede potenciar la capacidad de los servicios usados por los agentes y la organización virtual.

Además, destacamos dos líneas futuras de investigación: (i) herramienta gráfica de soporte de la transformación de la organización, y (ii) el uso de la tecnología propuesta en esta Tesis, para el desarrollo de sociedades humano-agentes, iHAS: *Intelligent Social Computing for Human-Agent Societies*, en el marco del proyecto TIN2012-36586-C03-01

La primera línea de investigación abierta, consiste en la creación de una herramienta gráfica que de soporte al proceso de transformación de modelos de la

organización. En este trabajo se proponen un conjunto las reglas de transformación, y de ellas se realizaron varias pruebas de concepto para evaluarla esta propuesta, sin embargo, no se implementó una herramienta como tal para el desarrollador del sistema, que genere en forma integrada las plantillas a nivel de la organización y a nivel del agente para una implementación más rápida y sencilla. Esta herramienta se puede crear desde cero o incluir estas reglas de transformación en alguna herramienta ya existente. En este sentido, en el proyecto THOMAS posee una herramienta gráfica basada en la metodología GORMAS y que está implementada sobre el software de Eclipse. Dicha herramienta permite crear diferentes diagramas de la metodología GORMAS[90]. Por ello, se puede pensar en crear una herramienta complementaria que permita hacer las transformaciones propuestas.

En la segunda línea de investigación abierta, se pretende desarrollar sistemas que permitan la inclusión de los humanos, es decir, donde los humanos sean considerados como una entidad más de la organización virtual, de una forma homogénea con los agentes del sistema.

7.3. Publicaciones Relacionadas con la Tesis

En esta sección presentamos las publicaciones generadas en el marco de la Tesis Doctoral. Estas publicaciones las hemos clasificado en artículos en revistas, en congresos internacionales y en congresos nacionales.

Artículos en revistas.

- J. Agüero, M. Rebollo, C. Carrascosa, and V. Julián. Agent Reactive Capabilities in Dynamic Environments. Neurocomputing, Elsevier, ISSN: 0925-2312, Volume In Press, 2014.
  (factor de impacto JCR 2014: 2.005)

- J. Agüero, M. Rebollo, C. Carrascosa, and V. Julián. Towards the development of agent-based organizations through MDD. International Journal on Artificial Intelligence Tools (IJAIT). DOI: 10.1142/S0218213013500024, Volume 2, No 2, pp. 1350002_1–1350002_35, 2013.
  (factor de impacto JCR 2012: 0.453)

- J. Agüero, M. Rebollo, C. Carrascosa, and V. Julián. MDD-Approach for developing Pervasive Systems based on Service-Oriented Multi-Agent Systems. ADCAIJ: Advances in Distributed Computing and Artificial Intelligence Journal. ISSN: 2255-2863, Volume 1, No 6, pp. 55–64, 2013.

Artículos en congresos internacionales.

- J. Agüero, M. Rebollo, C. Carrascosa, and V. Julián. Developing Pervasive Systems as Service-oriented Multi-Agent Systems. 7th International ICST Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services (MobiQuitous 2010), ISBN 978-963-9995-20-8, LNICST, Volume 73, pages 78-89, 2010.
  (Core ranking: A)

- J. Agüero, M. Rebollo, C. Carrascosa, and V. Julián. Towards on embedded agent model for android mobiles. In The Fifth Annual International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services (Mobiquitous 2008), ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), ISBN: 978-963-9799-21-9, pages 37:1-37:4, 2008.
  (Core ranking: A)

- J. Agüero, M. Rebollo, C. Carrascosa, and V. Julián. MDD-based agent-oriented software engineering for ubiquitous deployment. In The Sixth Annual International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services (Mobiquitous 2009), IEEE press, ISBN: 978-963-9799-59-2, pages 1-2, 2009.
  (Core ranking: A)

- J. Agüero, M. Rebollo, C. Carrascosa, and V. Julián. Agent design using Model Driven Development. In 7th International Conference on Practical Applications of Agents and Multi-Agent Systems (PAAMS2009). Volume 55, ISBN: 978-3-642-00486-5, pages 60-69, 2009.
  (Core ranking: C)

- J. Agüero, M. Rebollo, C. Carrascosa, and V. Julián. MDD for Virtual Organization design. In Proceedings of Trends in 8th International conference on Practical Applications of agents and multiagent systems(PAAMS2010). Volume 71, ISBN 978-3-642-12432-7, pages 9-17, 2010.
  (Core ranking: C)

- J. Agüero, M. Rebollo, C. Carrascosa, and V. Julián. Agent Capability Taxonomy for Dynamic Environments. 7th International Conference on Hybrid Artificial Intelligence Systems (HAIS 2012), E. Corchado et al. (Eds.): HAIS 2012, Part I, LNCS 7208, pp. 37–48. Springer, Heidelberg, 2012.
  (Core ranking: C)

- J. Agüero, M. Rebollo, C. Carrascosa, and V. Julián. Applying Model Driven Development in MAS for limited devices. In Seventh European Workshop on Multi-Agent Systems (EUMAS 2009), CD Press, pages 1-15, 2009. (Core ranking: C)

- J. Agüero, M. Rebollo, C. Carrascosa, and V. Julián. Developing intelligent agents in the Android platform. In Sixth European Workshop on Multi-Agent Systems (EUMAS 2008), CD press, pages 1-14, 2008.
(Core ranking: C)

- J. Agüero, M. Rebollo, C. Carrascosa, and V. Julián. Model-driven development for ubiquitous MAS. Ambient Intelligence and Future Trends-International Symposium on Ambient Intelligence (ISAmI 2010). Volume 72, ISBN 978-3-642-13267-4, pages 87-95, 2010.

- J. Agüero, M. Rebollo, C. Carrascosa, and V. Julián. Does android dream with intelligent agents?. In International Symposium on Distributed Computing and Artificial Intelligence 2008 (DCAI 2008), Volume 50, ISBN: 978-3-540-85862-1, pages 194-204, 2008.

Artículos en congresos nacionales.

- J. Agüero, M. Rebollo, C. Carrascosa, and V. Julián. Service-oriented Multi-Agent Systems for Ubiquitous Domains. In Proceedings of CAEPIA13 Conferences on Agent and Multiagent Systems: from theory to practice (ASMas 2013). ISBN: 978-84-695-8348-7, pp. 1503–1512, 2013

- J. Agüero, M. Rebollo, C. Carrascosa, and V. Julián. Developing Virtual Organizations Using MDD. In Proceedings of CAEPIA09 Workshop on Agreement Technologies(WAT 2009), Volume 635, ISSN 1613-0073, pages 130-141, 2009.

Bibliografía

[1] J. Agüero, M. Rebollo, C. Carrascosa, and V. Julián. Developing intelligent agents in the Android platform. In *Sixth European Workshop on Multi-Agent Systems (EUMAS 2008)*, volume CD Press, pages 1–14, 2008.

[2] J. Agüero, M. Rebollo, C. Carrascosa, and V. Julián. Does Android Dream with Intelligent Agents? In *International Symposium on Distributed Computing and Artificial Intelligence 2008 (DCAI 2008)*, volume 50, ISBN: 978-3-540-85862-1, pages 194–204, 2008.

[3] J. Agüero, M. Rebollo, C. Carrascosa, and V. Julián. Towards on embedded agent model for Android mobiles. In *The Fifth Annual International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services (Mobiquitous 2008)*, ISBN: 978-963-9799-21-9, pages 37:1–37:4, 2008.

[4] J. Agüero, M. Rebollo, C. Carrascosa, and V. Julián. Agent design using Model Driven Development. In *7th International Conference on Practical Applications of Agents and Multi-Agent Systems (PAAMS2009)*, volume 55, ISBN 978-3-642-00486-5, pages 60–69, 2009.

[5] J. Agüero, M. Rebollo, C. Carrascosa, and V. Julián. Developing Virtual Organizations Using MDD. In CEUR-WS, editor, *CAEPIA09 Workshop on Agreement Technologies(WAT 2009)*, volume 635, ISSN: 1613-0073, pages 130–141, 2009.

[6] J. Agüero, M. Rebollo, C. Carrascosa, and V. Julián. MDD-based agent-oriented software engineering for ubiquitous deployment. In *The Sixth Annual International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services(MobiQuitous 2009)*, volume IEEE press, ISBN: 978-963-9799-59-2, pages 1–2, 2009.

[7] J. Agüero, M. Rebollo, C. Carrascosa, and V. Julián. MDD for Virtual Organization design. In Springer-Verlag, editor, *Trends in Interna-*

tional conference on Practical Applications of agents and multiagent systems(PAAMS2010), volume 71, ISBN 978-3-642-12432-7, pages 9–17, 2010.

[8] J. Agüero, M. Rebollo, C. Carrascosa, and V. Julián. Model-driven development for ubiquitous MAS. In Springer-Verlag, editor, *Ambient Intelligence and Future Trends-International Symposium on Ambient Intelligence (ISAmI 2010)*, volume 72, ISBN: 978-3-642-13267-4, pages 87–95, 2010.

[9] J. Agüero, M. Rebollo, C. Carrascosa, and V. Julián. Developing Pervasive Systems as Service-oriented Multi-Agent Systems. In *7th International ICST Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services (MobiQuitous 2010)*, volume 73, ISBN 978-963-9995-20-8, pages 78–89, 2012.

[10] J. Agüero, M. Rebollo, C. Carrascosa, and V. Julián. MDD-Approach for developing Pervasive Systems based on Service-Oriented Multi-Agent Systems. *ADCAIJ: Advances in Distributed Computing and Artificial Intelligence Journal*, 1(6):55–64, 2013.

[11] J. Agüero, M. Rebollo, C. Carrascosa, and V. Julián. Towards the development of agent-based organizations through MDD. *International Journal on Artificial Intelligence Tools (IJAIT)*, 2(2):1350002:1–1350002:35, 2013.

[12] J. Agüero, M. Rebollo, C. Carrascosa, and V. Julián. Agent Reactive Capabilities in Dynamic Environments. *Neurocomputing*, In press, 2014.

[13] K. Akkaya and M. Younis. A survey on routing protocols for wireless sensor networks. *Ad hoc networks*, 3(3):325–349, 2005.

[14] I. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci. Wireless sensor networks: a survey. *Computer networks*, 38(4):393–422, 2002.

[15] F. Allilaire, J. Bézivin, F. Jouault, and I. Kurtev. ATL: Eclipse Support for Model Transformation. In *European Conference on Object-Oriented Programming (ECOOP2006)*, 2006.

[16] M. Amor and L. Fuentes. Malaca: A component and aspect-oriented agent architecture. *Information and Software Technology*, 51(6):1052–1065, 2009.

[17] Android. The Android Software Development Kit (SDK), May 2014.

[18] Apache-CXF. Distributed OSGi Reference Guide. http://cxf.apache.org/distributed-osgi-reference.html, June 2014.

[19] E. Argente, V. Botti, C. Carrascosa, A. Giret, V. Julian, and M. Rebollo. An abstract architecture for virtual organizations: The THOMAS approach. *Knowledge and Information Systems*, 29(2):379–403, 2011.

[20] E. Argente, A. Giret, S. Valero, V. Julian, and V. Botti. Survey of MAS Methods and Platforms focusing on organizational concepts. *Recent advances in Artificial Intelligence Research and Development*, 113:309–316, 2004.

[21] E. Argente, V. Julian, and V. Botti. Multi-Agent System Development based on Organizations. *Electronic Notes in Theoretical Computer Science*, 150:55–71, 2006.

[22] E. Argente, V. Julian, and V. Botti. MAS Modelling based on Organizations. In *9th Int. Workshop on Agent Oriented Software Engineering (AOSE08)*, pages 1–12, 2008.

[23] C. Atkinson and T. Kuhne. Model-driven development: a metamodeling foundation. *IEEE software*, 20(5):36–41, 2003.

[24] J. Augusto. Ambient intelligence: the confluence of ubiquitous/pervasive computing and artificial intelligence. *Intelligent Computing Everywhere*, pages 213–234, 2007.

[25] J. Augusto, H. Nakashima, and H. Aghajan. Ambient intelligence and smart environments: A state of the art. *Handbook of Ambient Intelligence and Smart Environments*, pages 3–31, 2010.

[26] I. Ayala, M. Amor, and L. Fuentes. A model driven engineering process of platform neutral agents for ambient intelligence devices. *Autonomous agents and multi-agent systems*, 28(2):214–255, 2014.

[27] M. Baldauf, S. Dustdar, and F. Rosenberg. A survey on context-aware systems. *International Journal of Ad Hoc and Ubiquitous Computing*, 2(4):263–277, 2007.

[28] B. Bauer. UML Class Diagrams Revisited in the Context of Agent-Based Systems. *Proceedings Agent-Oriented Software Engineering*, pages 101 – 118, 2002.

[29] B. Bauer, J. Mueller, and J. Odell. Agent UML: A Formalism for Specifying Multiagent Software Systems. *International Journal of Software Engineering and Knowledge Engineering*, 11(3):207–230, 2001.

[30] F. Bellifemine, A. Poggi, and G. Rimassa. JADE - A FIPA -compliant agent framework. In *Proceedings of the Practical Applications of Intelligent Agents*, 1999.

[31] A. Beresford and F. Stajano. Location privacy in pervasive computing. *Pervasive Computing, IEEE*, 2(1):46–55, 2003.

[32] F. Bergenti and A. Poggi. LEAP: A FIPA Platform for Handheld and Mobile Devices. In *Intelligent Agents VIII, Proceedings of the Eighth International Workshop on Agent Theories, Architectures, and Languages (ATAL-2001)*, number 436–446, 2001.

[33] C. Bernon, M. Cossentino, M.-P. Gleizes, P. Turci, and F. Zambonelli. A Study of some Multi-agent Meta-models. *Agent-Oriented Software Engineering V: 5th International Workshop, AOSE 2004. Lecture Notes in Computer Science*, 3382:62–77, 2004.

[34] G. Beydoun, G. Low, B. Henderson-Sellers, H. Mouratidis, J. Gomez-Sanz, J. Pavón, and C. Gonzalez-Perez. FAML: A Generic Metamodel for MAS Development. *IEEE Transactions on Software Engineering*, pages 841–863, 2009.

[35] J. Bézivin. On the unification power of models. *Software and Systems Modeling*, 4(2):171–188, 2005.

[36] N. Bhouri, S. Haciane, and F. Balbo. A multi-agent system to regulate urban traffic: Private vehicles and public transport. In *Intelligent Transportation Systems (ITSC), 2010 13th International IEEE Conference on*, pages 1575–1581. IEEE, 2010.

[37] A. Bikakis, T. Patkos, G. Antoniou, and D. Plexousakis. A survey of semantics-based approaches for context reasoning in ambient intelligence. pages 14–23, 2008.

[38] G. Boella, J. Hulstijn, and L. van der Torre. Virtual Organizations as Normative Multiagent Systems. *Hawaii International Conference on System Sciences (HICSS)*, 7:192–201, 2005.

[39] O. Boissier, J. Hübner, and J. Sichman. Organization oriented programming: From closed to open organizations. *Engineering Societies in the Agents World VII*, pages 86–105, 2007.

[40] C. Bolchini, C. A. Curino, E. Quintarelli, F. A. Schreiber, and L. Tanca. A data-oriented survey of context models. *ACM Sigmod Record*, 36(4):19–26, 2007.

[41] M. Bonfe, A. Boldrin, and E. Mainardi. Open-source technologies for embedded control systems: from robotics to home/building automation. *European Industrial Ethernet Award. IEEE*, 2011.

[42] R. H. Bordini, J. F. Hübner, and M. Wooldridge. *Programming multi-agent systems in AgentSpeak using Jason*, volume 8. John Wiley & Sons, 2007.

[43] A. Brando, V. Silva, and C. Lucena. A model driven approach to develop multi-agent systems. Technical report, Technical Report, Departmento de InformaticaPontificia Universidade Catolica do Rio de JaneiroPUCRio, 2005.

[44] R. F. Brena, J. L. Aguirre, C. Chesñevar, E. H. Ramírez, and L. Garrido. Knowledge and information distribution leveraged by intelligent agents. *Knowledge and Information Systems*, 12(2):203–227, 2007.

[45] P. Bresciani, A. Perini, P. Giorgini, F. Giunchiglia, and J. Mylopoulos. Tropos: An agent-oriented software development methodology. *Autonomous Agents and Multi-Agent Systems*, 8(3):203–236, 2004.

[46] P. Brézillon. Context in Artificial Intelligence: I. A survey of the literature. *Computers and artificial intelligence*, 18:321–340, 1999.

[47] J. Brønsted, K. Hansen, and M. Ingstrup. A survey of service composition mechanisms in ubiquitous computing. In *Second Workshop on Requirements and Solutions for Pervasive Software Infrastructures (RSPSI)'at Ubicomp*, 2007.

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

[49] F. Budinsky, S. A. Brodsky, and E. Merks. *Eclipse Modeling Framework*. Pearson Education, 2003.

[50] G. Caire, W. Coulier, F. Garijo, J. Gomez, J. Pavon, F. Leal, P. Chainho, P. Kearney, J. Stark, R. Evans, et al. Agent oriented analysis using MESSAGE/UML. *Agent-Oriented Software Engineering II*, pages 119–135, 2001.

[51] G. Canfora and M. Di Penta. Service-oriented architectures testing: A survey. In *Software Engineering*, pages 78–105. Springer, 2009.

[52] J. Cano, N. Madrid, R. Seepold, and F. Aguilar. Model-driven development of embedded systems on OSGi platforms. In *Forum on Specification and Design Languages (FDL07), Barcelona, Spain*, number 1–6, 2007.

[53] C. Carrascosa, A. Giret, V. Julian, M. Rebollo, E. Argente, and V. Botti. Service Oriented Multi-agent Systems: An open architecture. In *Autonomous Agents and Multiagent Systems (AAMAS)*, pages 1–2, 2009.

[54] C. Carrascosa Casamayor. *Meta-razonamiento en Agentes con Restricciones Temporales Críticas*. PhD thesis, Departamento de Sistemas Informá ticos y Computación. Universidad Politécnica de Valencia, 2003.

[55] J. Castro, M. Kolp, and J. Mylopoulos. A Requirements-Driven Development Methodology. *Conference on Advanced Information Systems Engineering*, pages 108 – 123, 2001.

[56] R. Cervenka and I. Trencansky. *The Agent Modeling Language – AML*, volume ISBN: 978-3-7643-8395-4. Whitestein Series in Software Agent Technologies and Autonomic Computing, 2007.

[57] H. Chen, T. Finin, and A. Joshi. An ontology for context-aware pervasive computing environments. *The Knowledge Engineering Review*, 18(03):197–207, 2004.

[58] M.-C. Chen, J.-L. Chen, and T.-W. Chang. Android/OSGi-based vehicular network management system. *Computer Communications*, 34(2):169–183, 2011.

[59] C. Cheng and L. Bonanni. Intelligent Spoon, 2006.

[60] S.-T. Cheng, C.-H. Wang, and G.-J. Horng. OSGi-based smart home architecture for heterogeneous network. *Expert Systems with Applications: An International Journal*, 39(16):12418–12429, 2012.

[61] C. Chong and S. Kumar. Sensor networks: evolution, opportunities, and challenges. *Proceedings of the IEEE*, 91(8):1247–1256, 2003.

[62] D. Cook, J. Augusto, and V. Jakkula. Ambient intelligence: Technologies, applications, and opportunities. *Pervasive and Mobile Computing*, 5(4):277–298, 2009.

[63] D. Cook, M. Youngblood, E. Heierman III, K. Gopalratnam, S. Rao, A. Litvin, and F. Khawaja. MavHome: An agent-based smart home. In *Pervasive Computing and Communications, 2003.(PerCom 2003). Proceedings of the First IEEE International Conference on*, pages 521–524. IEEE, 2003.

[64] D. J. Cook and S. K. Das. How smart are our environments? An updated look at the state of the art. *Pervasive Mob. Comput.*, 3(2):53–73, 2007.

[65] M. Cossentino. From requirements to code with the PASSI methodology. In *Agent-oriented Methodologies*, 2005.

[66] M. Cossentino and C. Potts. PASSI: A process for specifying and implementing multi-agent systems using UML. Technical report, Technical report, University of Palermo, 2001.

[67] N. Criado, E. Argente, V. Julián, and V. Botti. Designing Virtual Organizations. In *7th International Conference on Practical Applications of Agents and Multi-Agent Systems (PAAMS2009)*, volume 55, ISBN 978-3-642-00486-5, pages 440–449, Salamanca, Spain, 2009.

[68] S. Das and D. Cook. Health monitoring in an agent-based smart home by activity prediction. In *Toward a human-friendly assistive environment: ICOST'2004, 2nd International Conference on Smart Homes and Health Telematics*, page 3. Ios Pr Inc, 2004.

[69] P. Davidsson and M. Boman. Distributed monitoring and control of office buildings by embedded agents. *Inf. Sci. Inf. Comput. Sci.*, 171(4):293–307, 2005.

[70] P. Davidsson, L. Henesey, L. Ramstedt, J. Törnquist, and F. Wernstedt. An analysis of agent-based approaches to transport logistics. *Transportation Research part C: emerging technologies*, 13(4):255–271, 2005.

[71] V. Dignum. *A model for organizational interaction: based on agents, founded in logic*. Phd dissertation, Utrecht University, 2003.

[72] V. Dignum, J. Vázquez-Salceda, and F. Dignum. Omni: Introducing social structure, norms and ontologies into agent organizations. *Lecture Notes Artificial Intelligent*, 3346:181–198, 2005.

[73] F. Doctor, H. Hagras, and V. Callaghan. A type-2 fuzzy embedded agent to realise ambient intelligence in ubiquitous computing environments. *Inf. Sci. Inf. Comput. Sci.*, 171(4):309–334, 2005.

[74] C. Doukas and I. Maglogiannis. Intelligent pervasive healthcare systems. *Advanced Computational Intelligence Paradigms in Healthcare-3*, pages 95–115, 2008.

[75] D. D'Souza. Model-driven architecture and integration — opportunities and challenges. In *Version 1.1, Kineticum*, 2001.

[76] B. Elvesæter, A. Hahn, A. Berre, and T. Neple. Towards an interoperability framework for model-driven development of software systems. *Interoperability of Enterprise Software and Applications*, pages 409–420, 2006.

[77] C. Endres, A. Butz, and A. MacWilliams. A survey of software infrastructures and frameworks for ubiquitous computing. *Mobile Information Systems*, 1(1):41–80, 2005.

[78] M. Escrivá, J. Palanca, G. Aranda, A. García-Fornes, V. Julián, and V. Botti. A jabber-based multi-agent system platform. In *AAMAS '06: Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems*, pages 1282–1284, New York, NY, USA, 2006. ACM.

[79] M. Esteva, J. A. Rodríguez-Aguilar, J. L. Arcos, C. Sierra, and P. Garcia. Institutionalising open multi-agent systems. In *proceedings of the Fourth International Conference on MultiAgent Systems (ICMAS'2000)*, pages 381–83, 2000.

[80] M. Esteva, J. A. Rodríguez-Aguilar, C. Sierra, and J. L. Arcos. On the formal specifications of electronic institutions. *Agent mediated electronic commerce. Lecture Notes in Computer Science*, 1991:126–147, 2001.

[81] J. Ferber and O. Gutknecht. A meta-model for the analysis and design of organizations in multi-agent systems. In *Multi Agent Systems, 1998. Proceedings. International Conference on*, pages 128–135. IEEE, 1998.

[82] J. Ferber, O. Gutknecht, and F. Michel. From agents to organizations: an organizational view of multi-agent systems. *Proceedings AOSE, Lecture Notes in Computer Science*, 2935:214–230, 2003.

[83] J. Ferber, F. Michel, and J. Baez. AGRE: Integrating environments with organizations. *Environments for multi-agent systems: first international workshop, E4MAS*, pages 48–56, 2005.

[84] I. A. Ferguson. *Touring Machines: An architecture for dynamic, rational, mobile agents*. PhD thesis, University of Cambridge Cambridge, 1992.

[85] C. Fiehe, A. Litvina, I. Luck, O. Dohndorf, J. Kattwinkel, F.-J. Stewing, J. Kruger, and H. Krumm. Location-transparent integration of distributed osgi frameworks and web services. In *Advanced Information Networking and Applications Workshops, 2009. WAINA'09. International Conference on*, pages 464–469. IEEE, 2009.

[86] FIPA. Agent ACL message structure specification. Technical report, XC00061E, http://www.fipa.org, 2001.

[87] FIPA. Technical Report SC00001L. Technical report, Foundation for Intelligent Physical Agents, http://www.fipa.org, 2002.

[88] J. Flinn, S. Park, and M. Satyanarayanan. Balancing performance, energy, and quality in pervasive computing. pages 217–226, 2002.

[89] I. Foster, C. Kesselman, and S. Tuecke. The Anatomy of the Grid: Enabling Scalable Virtual Organizations. *International Journal Supercomputer Applications*, 15(3):200–222, 2001.

[90] E. Garcia, E. Argente, and A. Giret. A modeling tool for service-oriented Open Multiagent Systems. In *The 12th International Conference on Principles of Practice in Multi-Agent Systems. PRIMA 2009*, volume 5925 of *LNAI*, pages 345–360. Springer-Verlag, 2009.

[91] I. García-Magariño, J. Gómez-Sanz, and R. Fuentes. INGENIAS Development Assisted with Model Transformation By-Example: A Practical Case. In *7th International Conference on Practical Applications of Agents and Multi-Agent Systems (PAAMS 2009)*, pages 40 – 49, 2009.

[92] B. Gateau, O. Boissier, D. Khadraoui, and E. Dubois. Moiseinst: An organizational model for specifying rights and duties of autonomous agents. In *Proceedings of Workshop Coordination and Organisation (CoORG 2005)*, pages 484–485, 2005.

[93] J. Gomez Sanz. *Modelado de Sistemas Multi-Agente*. Phd thesis, Universidad Complutense de Madrid, Spain., 2002.

[94] P. Gowtham. An Interactive Hand Gesture Recognition System on the BeagleBoard. *International Proceedings of Computer Science and Information Technology*, 20:113–118, 2011.

[95] S. Greenberg and C. Fitchett. Phidgets: easy development of physical interfaces through physical widgets. In *Proceedings of the 14th annual ACM symposium on User interface software and technology*, pages 209–218. ACM, 2001.

[96] Z. Guessoum and T. Jarraya. Meta-models & model-driven architectures. In CEEMAS, editor, *Contribution to the AOSE TFG AgentLink3 meeting, Ljubljana*, volume 4696, pages 256–265. Springer, 2005.

[97] H. Hagras, V. Callaghan, and M. Colley. Intelligent Embedded Agents. *Information Sciences*, 171(4):289 – 292, 05 2005.

[98] H. Hagras, V. Callaghan, M. Colley, G. Clarke, A. Pounds-Cornish, and H. Duman. Creating an ambient-intelligence environment using embedded agents. *Intelligent Systems, IEEE*, 19(6):12–20, 2004.

[99] C. Hahn, C. Madrigal-Mora, and K. Fischer. A platform-independent metamodel for multiagent systems. *Autonomous Agents and Multi-Agent Systems*, 18(2):239–266, 2008.

[100] R. Hall, K. Pauls, S. McCulloch, and D. Savage. *OSGi in action: Creating modular applications in Java*. Manning Publications Co., 2011.

[101] R. S. Hall and H. Cervantes. Challenges in building service-oriented applications for OSGi. *Communications Magazine, IEEE*, 42(5):144–149, 2004.

[102] S. Helal. The Landscape of Pervasive Computing Standards. *Synthesis Lectures on Mobile & Pervasive Computing*, 5(1):1–89, 2010.

[103] S. Helal, W. Mann, H. El-Zabadani, J. King, Y. Kaddoura, and E. Jansen. The gator tech smart house: A programmable pervasive space. *Computer*, 38(3):50–60, 2005.

[104] K. Henricksen and J. Indulska. Developing context-aware pervasive computing applications: Models and approach. *Pervasive and Mobile Computing*, 2(1):37–64, 2006.

[105] J. Hightower and G. Borriello. Location systems for ubiquitous computing. *Computer*, 34(8):57–66, 2001.

[106] B. Horling and V. Lesser. A survey of multi-agent organizational paradigms. *The Knowledge Engineering Review*, 19(04):281–316, 2005.

[107] J. Hübner, J. Simao Sichman, and O. Boissier. S-Moise+: A middleware for developing organised multi-agent systems. *Lecture Notes in Computer Science*, 3913:64–77, 2006.

[108] M. Huhns and M. Singh. Service-oriented computing: key concepts and principles. *IEEE Internet Computing*, 9(1):75–81, Jan-Feb 2005.

[109] M. Huhns, M. Singh, M. Burstein, K. Decker, K. Durfee, T. Finin, T. Gasser, H. Goradia, P. Jennings, K. Lakkaraju, H. Nakashima, H. Van Dyke Parunak, J. Rosenschein, A. Ruvinsky, G. Sukthankar, S. Swarup, K. Sycara, M. Tambe, T. Wagner, and L. Zavafa. Research directions for

service-oriented multiagent systems. *IEEE Internet Computing*, 9(6):65–70, Nov.-Dec. 2005.

[110] C. Iglesias, M. Garijo, and J. Gonzalez. A survey of agent-oriented methodologies. *Lecture notes in computer science*, 1555:317–330, 2000.

[111] C. A. Iglesias Fernández. *Definición de una metodología para el desarrollo de sistemas multiagente*. PhD thesis, Departamento de ingeniería de sistemas telemáticos. Universidad Politécnica de Madrid, 1998.

[112] T. Jarraya and Z. Guessoum. Towards a model driven process for multiagent system. In CEEMAS, editor, *Multi-Agent Systems and Applications V*, volume 4696, pages 256–265. Springer, 2007.

[113] G. Jayatilleke, L. Padgham, and M. Winikoff. A model driven component-based development framework for agents. *International Journal of Computer Systems Science & Engineering*, 20(4):273–282, 2005.

[114] N. Jennings and M. Wooldridge. Applications of intelligent agents. In *Agent Technology: Foundations, Applications, and Markets*, pages 3–28, 1998.

[115] N. Jennings and M. Wooldridge. Agent-oriented software engineering. *Lecture notes in computer science*, 1647:4–10, 1999.

[116] V. Julian, M. Rebollo, E. Argente, V. Botti, C. Carrascosa, and A. Giret. *Using THOMAS for Service Oriented Open MAS*, pages 56–70. Springer, 2009.

[117] V. Julián Inglada. *RT-MESSAGE: Desarrollo de Sistemas Multiagente de Tiempo Real*. PhD thesis, Departamento de Sistemas Informá ticos y Computación. Universidad Politécnica de Valencia, 2002.

[118] C. Ke and Z. Huang. Self-adaptive semantic web service matching method. *Knowledge-Based Systems*, 35:41–48, 2012.

[119] C. Kidd, R. Orr, G. Abowd, C. Atkeson, I. Essa, B. MacIntyre, E. Mynatt, T. Starner, and W. Newstetter. The aware home: A living laboratory for ubiquitous computing research. *Cooperative buildings. Integrating information, organizations, and architecture*, pages 191–198, 1999.

[120] J. King, R. Bose, H.-I. Yang, S. Pickles, and A. Helal. Atlas: A service-oriented sensor platform: Hardware and middleware to enable programmable pervasive spaces. In *Local Computer Networks, Proceedings 2006 31st IEEE Conference on*, pages 630–638. IEEE, 2006.

[121] D. Kinny and M. Georgeff. Modelling and design of multi-agent systems. Technical report, 59, Australian Artificial Intelligence Institute, Melbourne, 1996.

[122] J. Kirby. Model-Driven Agile Development of Reactive Multi-Agent Systems. In *Computer Software and Applications Conference, 2006. COMPSAC'06. 30th Annual International*, volume 2, 2006.

[123] A. Kleppe, J. B. Warmer, and W. Bast. *MDA Explained: The Model Driven Architecture: Practice and Promise*. Addison-Wesley Professional, 2003.

[124] M. Knoll, T. Weis, A. Ulbrich, and A. Brändle. Scripting your home. *Location and Context-Awareness*, pages 274–288, 2006.

[125] M. Kolp, P. Giorgini, and J. Mylopoulos. Multi-agent architectures as organizational structures. *Autonomous Agents and Multi-Agent Systems*, 13(1):3–25, 2006.

[126] Y. Labrou and T. Finin. KQML as an agent communication language. In *Proccedings 3rd International. Conference. On Information and Knowledge Management*. MIT Press, 1994.

[127] T. C. Lech and L. W. M. Wienhofen. AmbieAgents: a scalable infrastructure for mobile and context-aware information services. In *AAMAS '05: Proceedings of the fourth international joint conference on Autonomous agents and multiagent systems*, pages 625–631, New York, NY, USA, 2005. ACM.

[128] C. Longbing, C. Zhang, and D. Ruwei. Organization-oriented analysis of open complex agent systems. *International Journal of Intelligent Control and Systems*, 10(2):114–122, 2005.

[129] M. Luck and P. McBurney. Computing as interaction: agent and agreement technologies. In *Proc. of the 2008 IEEE International Conference on Distributed Human-Machine Systems*, pages 1–6, 2008.

[130] M. Luck, P. McBurney, O. Shehory, and S. Willmott. Agent Technology: Computing as Interaction. A Roadmap for Agent Based Computing. 2005.

[131] Z. Maamar, S. Mostefaoui, and H. Yahyaoui. Toward an agent-based and context-oriented approach for Web services composition. *IEEE Transactions on Knowledge and Data Engineering*, pages 686–697, 2005.

[132] P. Maes. The agent network architecture (ANA). *ACM SIGART Bulletin*, 2(4):115–120, 1991.

[133] D. Martin, M. Burstein, D. Mcdermott, S. Mcilraith, M. Paolucci, K. Sycara, D. L. Mcguinness, E. Sirin, and N. Srinivasan. Bringing semantics to web services with OWL-S. *World Wide Web*, 10(3):243–277, 2007.

[134] A. Mas. *Agentes software y sistemas multiagente: Conceptos, arquitecturas y aplicaciones*. Pearson. Prentice Hall. Madrid, 2005.

[135] M. Matos and A. Sousa. Dependable distributed OSGi environment. In *Proceedings of the 3rd workshop on Middleware for service oriented computing*, pages 1–6. ACM, 2008.

[136] J. McGinnis, K. Stathis, and F. Toni. A Formal Framework of Virtual Organisations as Agent Societies. *EPTCS*, 16:1–14.

[137] G. Meditskos and N. Bassiliades. Structural and role-oriented Web service discovery with taxonomies in OWL-S. *Knowledge and Data Engineering, IEEE Transactions on*, 22(2):278–290, 2010.

[138] P. Mees. *A very public solution: Transport in the dispersed city*. 2000.

[139] L. Merk, M. Nicklous, T. Stober, and U. Hansmann. *Pervasive Computing Handbook*. Springer Verlag, January, 2001.

[140] N. Minsky and V. Ungureanu. Law-governed interaction: a coordination and control mechanism for heterogeneous distributed systems. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 9(3):273–305, 2000.

[141] G. Moore. Lithography and the Future of Moore's Law. *Advances in resist technology and processing*, 2438:2, 1995.

[142] A. Mukherjee. Pervasive Computing: A Paradigm for the 21st Century. *IEEE Computer*, 36(3):25–31, 2003.

[143] J. Munoz, V. Pelechano, and J. Fons. Model driven development of pervasive systems. *International Workshop on Model-Based Methodologies for Pervasive and Embedded Software (MOMPES2004)*, pages 3–14, 2004.

[144] j. Munoz Ferrara. *Model Driven Development of Pervasive Systems. Building a Software Factory*. PhD thesis, Departamento de Sistemas Informá ticos y Computación. Universidad Politécnica de Valencia, 2008.

[145] H. Nakashima, H. Aghajan, and J. Augusto. *Handbook of ambient intelligence and smart environments*. Springer, 2009.

[146] X. T. Nguyen and R. Kowalczyk. Ws2jade: Integrating web service with jade agents. In *Service-Oriented Computing: Agents, Semantics, and Engineering*, pages 147–159. Springer, 2007.

[147] H. S. Nwana. Software agents: An overview. *Knowledge Engineering Review*, 11(3):205–244, 1996.

[148] J. Odell, M. Nodine, and R. Levy. A metamodel for agents, roles, and groups. *Agent-Oriented Software Engineering V*, pages 78–92, 2005.

[149] T. Ohtani and M. THINT. An Intelligent System for Managing and Utilizing Information Resources Over the Internet. *International Journal on Artificial Intelligence Tools*, 11(1):117–138, 2002.

[150] M. Omair Shafiq, A. Ali, H. Farooq Ahmad, and H. Suguri. Agentweb gateway-a middleware for dynamic integration of multi agent system and web services framework. In *Enabling Technologies: Infrastructure for Collaborative Enterprise, 2005. 14th IEEE International Workshops on*, pages 267–268. IEEE, 2005.

[151] (OMG). Object management group. MDA guide version 1.0.1. http://www.omg.org/docs/omg/03-06-01.pdf, June 2003.

[152] (OMG). Object management group. MOF QVT final adopted specification. http://www.omg.org/docs/ptc/05-11-01.pdf, November 2005.

[153] (OMG). Object management group. meta object facility (MOF) 2.0 core specification. http://www.omg.org/docs/ptc/04-10-15.pdf, October 2004.

[154] A. Omicini. SODA: Societies and infrastructures in the analysis and design of agent-based systems. In *Agent-Oriented Software Engineering 1957*, pages 185–193. Springer, 2001.

[155] A. Omicini, A. Ricci, and M. Viroli. RBAC for organisation and security in an agent coordination infrastructure. *Electronic Notes in Theoretical Computer Science*, 128(5):65–85, 2005.

[156] S. Ossowski. *Agreement technologies*, volume 8. Springer, 2012.

[157] M. Papazoglou and D. Georgakopoulos. Service-oriented computing. *Communications of the ACM*, 46(10):25–28, 2003.

[158] M. Papazoglou, P. Traverso, S. Dustdar, and F. Leymann. Service-oriented computing: State of the art and research challenges. *IEEE Computer Society*, 40(11):38–45, 2007.

[159] J. Pavón and J. Gómez-Sanz. Agent oriented software engineering with INGENIAS. In *Proceedings of the 3rd Central and Eastern European conference on Multi-agent systems*, pages 394–403. Springer-Verlag, 2003.

[160] J. Pavón, J. Gómez-Sanz, and R. Fuentes. Model Driven Development of Multi-Agent Systems. *Lecture Notes in Computer Science*, 4066:284 – 298, 2006.

[161] A. Perini and A. Susi. Automating Model Transformations in Agent-Oriented Modelling. In *Agent-Oriented Software Engineering VI*, pages 167 – 178, 2006.

[162] G. Picard and M. pierre Gleizes. The ADELFE methodology. In *Methodologies and Software Engineering for Agent Systems*, volume 8, 2004.

[163] M. Piunti, A. Ricci, and A. Santi. Soa/ws applications using cognitive agents working in cartago environments. In *Proceedings of 10th joint conference AI* IA TABOO From Objects to Agents (WOA 2009)*, 2009.

[164] A. Poggi, M. Tomaiuolo, and P. Turci. An agent-based service oriented architecture. In *Proc. 8th AI* IA/TABOO Joint Workshop From Objects to Agents: Agents and Industry: Technological Applications of Software Agents, Genova*, pages 157–165, 2007.

[165] V. Poladian, J. Sousa, D. Garlan, B. Schmerl, and M. Shaw. Task-based adaptation for ubiquitous computing. *IEEE Transactions on Systems, Man, and Cybernetics, Part C: Applications and Reviews, Special Issue on Engineering Autonomic Systems*, 36(3):328–340, 2006.

[166] D. Preuveneers, J. Van den Bergh, D. Wagelaar, A. Georges, P. Rigole, T. Clerckx, Y. Berbers, K. Coninx, V. Jonckers, and K. De Bosschere. Towards an extensible context ontology for ambient intelligence. pages 148–159, 2004.

[167] J. B. Quintero and R. Anaya. MDA y el papel de los modelos en el proceso de desarrollo de software. *Revista EIA*, 8:131–146, 2007.

[168] C. Ramos, J. Augusto, and D. Shapiro. Ambient intelligence—The next step for artificial intelligence. *Intelligent Systems, IEEE*, 23(2):15–18, 2008.

[169] A. S. Rao, M. P. Georgeff, et al. BDI Agents: From Theory to Practice. In *ICMAS*, volume 95, pages 312–319, 1995.

[170] J. S. Rellermeyer, G. Alonso, and T. Roscoe. R-OSGi: distributed applications through software modularization. In *Proceedings of the ACM/IFIP/USENIX 2007 International Conference on Middleware*, pages 1–20. Springer-Verlag New York, Inc., 2007.

[171] A. Ricci, M. Piunti, M. Viroli, and A. Omicini. Environment programming in CArtAgO. In *Multi-Agent Programming:*, pages 259–288. Springer, 2009.

[172] P. Ricordel and Y. Demazeau. From analysis to deployment: A multi-agent platform survey. In *Engineering societies in the agents world*, pages 93–105. Springer, 2000.

[173] M. Rodrigo, B. Roselli, J. Rodríguez-Aguilar, V. Julián, and C. Carrascosa. Mixing Electronic Institutions with Virtual Organizations: A Solution Based on Bundles. In J. Pérez, J. Corchado, M. Moreno, V. Julián, P. Mathieu, J. Canada-Bago, A. Ortega, and A. Caballero, editors, *Highlights in Practical Applications of Agents and Multiagent Systems*, volume 89 of *Advances in Intelligent and Soft Computing*, pages 143–150. Springer Berlin Heidelberg, 2011.

[174] D. Ronzani. The battle of concepts: Ubiquitous Computing, pervasive computing and ambient intelligence in Mass Media. *Ubiquitous Computing and Communication Journal*, 4(2):9–19, 2009.

[175] L. Rudolph. Project Oxygen: pervasive, human-centric computing–an initial experience. In *Advanced Information Systems Engineering*, pages 1–12. Springer, 2001.

[176] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall Series in Artificial Intelligence. Englewood Cliffs, New Jersey, 1995.

[177] N. Sadeh, E. Chan, and L. Van. MyCampus: an agent-based environment for context-aware mobile services. 2002.

[178] M. Satyanarayanan. Pervasive computing: Vision and challenges. *IEEE Personal Communications*, 8(4):10–17, 2001.

[179] E. Schoitsch and A. Skavhaug. Special: Embedded Intelligence. In *ERCIM NEWS. European Research Consortium for Informatic and Mathematics*, number 67, October 2006.

[180] B. Selic. The pragmatics of model-driven development. *IEEE software*, 20(5):19–25, 2003.

[181] E. Serrano and J. Botia. Validating ambient intelligence based ubiquitous computing systems by means of artificial societies. *Information Sciences*, 222:3–24, 2013.

[182] N. Shadbolt. Ambient intelligence. *IEEE Intelligent Systems*, 18:2–3, 2003.

[183] V. Silva, A. Garcia, A. Brandão, C. Chavez, C. Lucena, and P. Alencar. Taming agents and objects in software engineering. *Software engineering for large-scale multi-agent systems. LNCS*, 2603:103–136, 2003.

[184] M. Singh and M. Huhns. *Service-oriented computing: semantics, processes, agents*. John Wiley & Sons Inc, 2005.

[185] N. Skarmeas and K. Clark. Component based agent construction. *International Journal on Artificial Intelligence Tools*, 11(1):139–164, 2002.

[186] J. Soldatos, I. Pandis, K. Stamatis, L. Polymenakos, and J. Crowley. Agent based middleware infrastructure for autonomous context-aware ubiquitous computing services. *Computer Communications*, 30(3):577–591, 2007.

[187] A. Sturm, D. Dori, and O. Shehory. Single-model method for specifying multi-agent systems. In *Proceedings of the second international joint conference on Autonomous agents and multiagent systems*, pages 121–128. ACM, 2003.

[188] J. M. Such, A. García-Fornes, A. Espinosa, and J. Bellver. Magentix2: A privacy-enhancing agent platform. *Engineering Applications of Artificial Intelligence*, 26(1):96–109, 2013.

[189] J. Sudeikat, L. Braubach, A. Pokahr, and W. Lamersdorf. Evaluation of Agent–Oriented Software Methodologies–Examination of the Gap Between Modeling and Platform. *Agent-Oriented Software Engineering V*, pages 126–141, 2005.

[190] K. Sycara, M. Paolucci, J. Soudry, and N. Srinivasan. Dynamic discovery and coordination of agent-based semantic web services. *Internet Computing, IEEE*, 8(3):66–73, 2004.

[191] K. Sycara, S. Widoff, M. Klusch, and J. Lu. Larks: Dynamic matchmaking among heterogeneous software agents in cyberspace. *Autonomous agents and multi-agent systems*, 5(2):173–203, 2002.

[192] triskell project (IRISA). The metamodeling language kermeta. http://www.kermeta.org, 2006.

[193] A. van Breemen, X. Yan, and B. Meerbeek. iCat: an animated user-interface robot with personality. pages 143–144, 2005.

[194] U. Varshney. Pervasive healthcare. *Computer*, 36(12):138–140, 2003.

[195] S. A. Velastin, B. A. Boghossian, B. P. Lo, J. Sun, and M. A. Vicencio-Silva. PRISMATICA: toward ambient intelligence in public transport environments. *Systems, Man and Cybernetics, Part A: Systems and Humans, IEEE Transactions on*, 35(1):164–182, 2005.

[196] A. F. Vilas, R. P. Díaz Redondo, J. J. Pazos Arias, M. R. Cabrer, A. Gil Solla, J. G. Duque, et al. Context-aware personalization services for a residential gateway based on the OSGi platform. *Expert Systems with Applications*, 37(9):6538–6546, 2010.

[197] M. Vinyals, J. Rodriguez-Aguilar, and J. Cerquides. A survey on sensor networks from a multiagent perspective. *The Computer Journal*, 54(3):455–470, 2011.

[198] M. Weiser. The computer for the 21st century. *Scientific American*, 265(3):94–104, 1991.

[199] M. Weiser. Some computer science issues in ubiquitous computing. *Communications of the ACM*, 36(7):75–84, 1993.

[200] M. Wooldridge and N. R. Jennings. Agent theories, architectures, and languages: a survey. In *ECAI-94: Proceedings of the workshop on agent theories, architectures, and languages on Intelligent agents*, pages 1–39, New York, NY, USA, 1995. Springer-Verlag New York, Inc.

[201] M. Wooldridge, N. R. Jennings, and D. Kinny. The GAIA Methodology for Agent-Oriented Analysis and Design. *Journal of Autonomous Agents and Multi-Agent Systems*, 3:285–312, 2000.

[202] M. Wooldridgey and P. Ciancarini. Agent-oriented software engineering: The state of the art. In *Agent-Oriented Software Engineering, LNAI 1957*, pages 55–82. Springer, 2001.

[203] C.-L. Wu, C.-F. Liao, and L.-C. Fu. Service-oriented smart-home architecture based on OSGi and mobile-agent technology. *Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on*, 37(2):193–205, 2007.

[204] F. Zambonelli, N. R. Jennings, and M. Wooldridge. Developing multiagent systems: The GAIA methodology. *ACM Trans. Softw. Eng. Methodol.*, 12(3):317–370, 2003.

[205] C. Zato, G. Villarrubia, A. Sánchez, J. Bajo, and J. M. Corchado. PANGEA: A New Platform for Developing Virtual Organizations of Agents. *International Journal of Artificial Intelligence™*, 11(A13):93–102, 2013.

[206] C. Zato, G. Villarrubia, A. Sánchez, I. Barri, E. Rubión, A. Fernández, C. Rebate, J. A. Cabo, T. Álamos, J. Sanz, et al. PANGEA–Platform for Automatic coNstruction of orGanizations of intElligent Agents. In *Distributed Computing and Artificial Intelligence*, pages 229–239. Springer, 2012.

Anexo I

Migrando *Android* en la Beagleboard

En esta sección se resumen los pasos para obtener el código fuente completo de las imágenes de Android (cargador de arranque, el núcleo y sistema de archivos), para su compilación posteriormente.

Estos pasos suponen que el computador de compilación para Android se basa en Ubuntu, con una versión 10.04 o superior. El computador utiliza algunos paquetes necesarios para la compilación de Android, éstos se pueden descargar e instalar dinámicamente usando los siguientes los comandos de la FiguraI.1 (para Ubuntu de 32 bits):

```
$ sudo add-apt-repository "deb http://archive.canonical.com/lucid partner"
$ sudo add-apt-repository "deb-src http://archive.canonical.com/ubuntu lucid partner"
$ sudo apt-get update
$ sudo apt-get install git-core gnupg sun-java6-jdk flex bison gperf libsdl-dev ...
   libesd0-dev libwxgtk2.6-dev build-essential zip curl libncurses5-dev ...
   zlib1g-dev minicom tftpd uboot-mkimage expect
$ sudo update-java-alternatives -s java-6-sun
```

Figura I.1: Comandos para instalar los paquetes requeridos

Es de notar que Android Gingerbread (2.3.4) necesita Java 6 en Ubuntu, mientras que la versión anterior Froyo (2.2) usa Java 5. Una vez que el computador tiene las dependencias necesarias se realizan los siguientes pasos:

Obtener el código fuente: los desarrolladores pueden descargar el código fuente desde el repositorio gitorious.org/rowboat (se supone que vamos a utilizar la distribución de Android conocida como rowboat[1]). Para ello se usa la

[1]http://code.google.com/p/rowboat/

herramienta llamada `repo` que ayuda a buscar y descargar las fuentes de Android desde sus repositorios. Para instalar, inicializar y configurar `repo`, se crea el directorio `bin/` y luego, se descarga el script `repo` y se convierte en un fichero ejecutable (como indica la Figura I.2).

```
$ mkdir ~/bin
$ PATH=~/bin:$PATH

$ curl https://dl-ssl.google.com/dl/googlesource/git-repo/repo > ~/bin/repo
$ chmod a+x ~/bin/repo
```

Figura I.2: Instalar la herramienta `repo`

Finalmente, los comandos de la Figura I.3 permiten clonar los archivos fuentes del kernel de Android desde el repositorio, suponemos que usamos la versión de Android conocida como Gingerbread (2.3.4).

```
$ mkdir $HOME/rowboat-android
$ cd $HOME/rowboat-android
$ repo init -u git://gitorious.org/rowboat/manifest.git -m rowboat-gingerbread.xml
$ repo sync
```

Figura I.3: Descargar el kernel

Ajuste del compilador: se configura para que el compilador apunte a la herramienta o compilador ARM `arm-eabi-`, como se muestra en la Figura I.4

```
$ export PATH=$HOME/rowboat-android/prebuilt/linux-x86/toolchain/arm-eabi-4.4.3/bin:$PATH
```

Figura I.4: Ajustar el compilador

Compilar el cargador de arranque: permite iniciar la Beagleboard apropiadamente. Para ello se debe cambiar de directorio a `u-boot` y escriba los siguientes comandos (ver Figura I.5). Al finalizar se generarán los ficheros para el arranque `MLO` y el`u-boot`.

Configurar el Kernel: esto permite configurar y activar las características necesarias del Kernel. Este paso generalmente es necesario para habilitar y exportar a los usuarios los puertos de extensión de la Beagleboard, necesarios para conectar los dispositivos: como el GPIO, SPI, I2C, etc. (ver Figura I.6).

Compilar el Kernel: esto genera el fichero `uImage` (la imagen del kernel) en la carpeta `kernel/arch/arm/boot`, para ello se debe cambiar a la carpeta `kernel` y escriba los siguientes comandos de la Figura I.7.

```
$ cd u-boot
$ make CROSS_COMPILE=arm-eabi- distclean
$ make CROSS_COMPILE=arm-eabi- beagleboard_config
$ make CROSS_COMPILE=arm-eabi-
```

Figura I.5: Compilar en cargador de arranque

```
$ cd kernel
$ make ARCH=arm menuconfig
```

Figura I.6: Configurar el kernel

```
$ cd kernel
$ make ARCH=arm CROSS_COMPILE=arm-eabi- distclean
$ make ARCH=arm CROSS_COMPILE=arm-eabi- beagleboard_android_defconfig
$ make ARCH=arm CROSS_COMPILE=arm-eabi- uImage
```

Figura I.7: Compilar el Kernel

Compilar el sistema de archivos: se descargaron los fuentes de rowboat usando el comando `git`, esto clona el árbol del sistema de archivo de *Android* en una carpeta con el nombre `rowboat-android` (ver Figura I.8).

```
$ git clone -b rowboat-gingerbread git://gitorious.org/rowboat/rowboperf.git $ cd rowboperf
```

Figura I.8: Descargar del sistema de ficheros

Para compilar el sistema de ficheros (rootfs, librerías, archivos del sistema) se emplea el comando `make TARGET_PRODUCT=beagleboard`. Después de compilar, el sistema de archivos se encuentra en la carpeta `out/target/product/beagleboard`. Finalmente, los archivos generados, el cargador de arranque, la imagen del Kernel y el sistema de archivo, se deben copiar en la memoria flash o la tarjeta SD de la Beagleboard.