Model Driven Development of Agents for Ambient Intelligence

Inmaculada Ayala Viñas

Departamento de Lenguajes y Ciencias de la Computación

Universidad de Málaga

Supervised by

Mercedes Amor Pinilla and Lidia Fuentes Fernández

November 2013

AUTOR: Inmaculada Ayala Viñas

EDITA: Servicio de Publicaciones de la Universidad de Málaga

A los que están y a los que no están.

La Dra. Doña Mercedes Amor Pinilla, Titular de Universidad, y la Dra. Doña Lidia Fuentes Fernández, Catedrática de Universidad, ambas pertenecientes al área de Telemática de la E.T.S. de Ingeniería Informática de la Universidad de Málaga,

Certifican que Dña. Inmaculada Ayala Viñas, Ingeniera Informática, ha realizado en el Departamento de Lenguajes y Ciencias de la Computación de la Universidad de Málaga, bajo nuestra dirección, el trabajo de investigación correspondiente a su Tesis Doctoral titulada:

Model Driven Development of Agents for Ambient Intelligence

Revisado el presente trabajo, estimamos que puede ser presentado al tribunal que ha de juzgarlo, y autorizamos la presentación de esta Tesis Doctoral en la Universidad de Málaga.

Málaga, Septiembre de 2013

Fdo.: Mercedes Amor Pinilla
Titular de Universidad
Área de Telemática

Fdo.: Lidia Fuentes Fernández
Catedrática de Universidad
Área de Telemática

Acknowledgements

This thesis has been supported by a FPI fellowship and project TIN2008-01942 (RAP), both funded by the Ministerio de Ciencia e Innovación.

Special Acknowledgements

Con la escritura de esta tesis doctoral se cierra un ciclo y es el momento de dar las gracias a todos aquellos que han contribuido a ella de una manera u otra.

En primer lugar, me gustaría dar las gracias a mis padres y a mi hermano, a nivel académico no habéis estado en la primera línea de batalla, pero sin lo que he aprendido de vosotros no habría llegado hasta aquí.

En segundo lugar, a Mercedes y a Lidia. Sin estas dos magníficas mujeres la escritura de esta tesis no hubiera sido posible. Muchas gracias por darme ánimos cuando los necesitaba y exigirme más cuando era necesario.

A mis queridísimos compañeros del 3.3.3, tanto a los que veo todos los días como aquellos que están por medio mundo. Sois un grupo de gente maravilloso, en estos años he aprendido mucho de vosotros y hacéis que todos los días de gusto llegar al trabajo.

A mis compañeros del grupo de investigación CAOSD, por su ayuda, enseñanzas y sabios consejos. Un agradecimiento especial para Miguel Aragüez y a José Luis Barreche por su ayuda con las implementaciones.

A mis niñas de Arriate y a mis niñas de Málaga, por su amistad.

A mi familia por ser grande, divertida y llena de gente estupenda.

Thank you to Franco Zambonelli and the members of the Agents and Pervasive Computing Labs (Marco, Laura, Gabriella and Alberto) for accepting me in Reggio Emilia and making my stay there a great pleasure.

Thank you very much to my friends from Reggio Emilia (Italian and Spanish) for their hospitality.

A José Luis Reyes y a Rafael Gutiérrez, por ayudarme con todos los papeles que he tenido que rellenar durante estos cuatro años y sacarme de algún lío.

Por último a Joaquín, el más inteligente de los secuaces y mi incondicional compañero de aventuras. En esta tesis no cabrían los motivos para darte las gracias, pero sobre todo, muchas gracias por no rendirte nunca. Me inspiras todos los días.

Contents

List of Figures

List of Tables

Nomenclature

Acronyms

L^2TS Doubly Labeled Transition System

AAL Ambient Assisted Living

AF Autonomic Function

AFME Agent Factory Micro Edition

AM Autonomic Manager

AmI Ambient Intelligence

AMS Agent Management Service

AOP Aspect Oriented Programming

AOSD Aspect-Oriented Software Development

AOSE Agent-Oriented Software Engineering

CBSE Component Based Software Engineering

CIM Computation Independent Model

DF Directory Facilitator

ECA Event Condition Action

EMF Eclipse Modeling Framework

FSM Finite State Machine

GMF Graphical Modeling Framework

GMS Group Management Service

GUI General User Interface

IM Intelligent Museum

IPMT Inter Platform Message Transport

ISTAG IST Advisory Group

ITS Intelligent Transport System

L2CAP Link Control and Adaptation Control

LOC Line Of Code

M2M Model-To-Model

M2T Model-To-Text

MAPE Monitor Analyze Plan Execute

MAS Multi-Agent Systems

MDA Model Driven Architecture

MDD Model-Driven Development

MOF Meta Object Facility

MTS Message Transport Service

NE Number of Elements

OMG Object Management Group

OO Object Orientation

PAN Personal Area Network

PIM Platform Independent Metamodel

PSM Platform Specific Metamodel

RFCOMM Radio Frequency Communications

SDP Service Discovery Protocol

SRS Strategy Realization System

STD State Transition Diagram

UML Unified Modeling Language

WLAN Wireless Local Area Network

WPAN Wireless Personal Area Network

WSN Wireless Sensor Network

Other Symbols

mA milliamperes

ms milliseconds

Chapter 1

Introduction

Ambient Intelligence (AmI) proposes a new vision of technology that encourages new, more natural relationships between people and electronics becoming invisible through its presence in devices embedded in our natural surroundings. AmI systems, are naturally distributed and normally composed of a myriad of devices which are interconnected through a great diversity of communication technologies, posing new challenges that need to be addressed with appropriate software technologies. In this thesis we explore the applicability of software agents, model driven and aspect-oriented development technologies to improve the development of AmI systems.

Software agents and Multi-Agent Systems (MAS) are considered good options for the development of applications in the AmI domain [Sadri, 2011]. The intelligent, reactive, proactive and social behavior of software agents perfectly meets the requirements of AmI systems. However, in order for agent-based computing to become a widely accepted technology for developing AmI systems it would be advantageous to ease both the design and implementation of AmI systems with agents, by providing adequate development tools. These tools should automate some of the developer's tasks, augmenting the production of applications for AmI systems.

Model-Driven Development (MDD) [Stahl and Völter, 2006] is an approach for software development that promotes both the use of models to formally represent domain-specific concepts, and the automation of software development tasks by

means of model transformations. These models follow a syntax defined by artifacts called metamodels. Although there are already several metamodels for agents, AmI systems have particular characteristics that should be modeled from the early phases of the software's development. The most relevant ones are the necessity of AmI systems to be aware of some properties of their environment (i.e. context awareness) and to be able to react autonomously to changes in those properties (i.e. self-management) [Kephart and Chess, 2003].

We have defined a model-driven process adapted to the necessities of the development of MASs with self-management capacities and able to be executed in AmI devices, such as mobile phones or sensors. In this thesis we focus on a fully distributed and decentralized architecture of MASs, by embedding agents in heterogenous AmI devices. The main motivation for encouraging the embedding of agents in lightweight devices is that it is possible to adapt agents functionality to the specific hardware and the scarcity of computational resources. Recently, some of the most well-known agent approaches have released new versions specifically for lightweight devices (e.g. Jade-Leap [Bellifemine et al., 2001; Bergenti and Poggi, 2002], μFIPA-OS [Laukkanen et al., 2002]) and new ones have been proposed (e.g. Andromeda [Agüero et al., 2009], MAPS [Aiello et al., 2009]). However, they present serious limitations and deficiencies in coping with the diversity of devices and communication protocols. These limitations are addressed as part of this thesis.

MDD helps us to separate the platform independent properties from the platform specific, so it is possible to model AmI special properties irrespective of the target execution agent platform. But, in AmI systems the generation of agents is needed for different kinds of devices (e.g. mobile phones, different kind of sensors, ...), with a diversity of operating systems (e.g. Android, SunSPOT, TinyOS, ...) and interconnected through different communication protocols (e.g. WiFi, Zigbee, ...). So, in order to improve the MDD process we have used Aspect-Oriented Software Development (AOSD) [Filman et al., 2004] concepts mainly to separate the communication-related concerns specific to each AmI device.

1.1 Ambient Intelligence: motivation and challenges

The 20th century was replete with fiction stories that reflected the visions of different writers and film-makers about how our lives would be in the present millennium. Well-known writers such as Philip K. Dick or Isaac Asimov captured different visions of the future, the technological development of which embarrasses scientists and engineers nowadays. Where are the robots that respect the three laws of robotics? Where is HAL 9000? Where is my unmanned car? Of course, they are closer to us than ever, but still far off. Robots do not have a consciencenes that makes these laws necessary, but it is already a reality that they can for example, assist the elderly in their daily lives [Pollack et al., 2002]. We do not have psychopathic killer computers like HAL 9000, but we do have systems that can self-configure, taking into account our preferences thereby making our homes more comfortable [Hagras et al., 2004]. Unmanned cars are not on the streets, but the Google driverless car [Guizzo, 2011] and DARPA Robotic Challenge[1] show that these vehicles could become a reality in our daily lives. The objective of AmI is to make this no longer science fiction, but rather a reality.

Aligned with this vision, the AmI term was coined by the IST Advisory Group (ISTAG) in 2001 in [Ducatel et al., 2001] and later revised in [Ducatel et al., 2003]. These technical reports provide a set of scenarios where AmI has an important role in making our lives more comfortable and safer. In these reports the role of agent technology in accomplishing this vision is highlighted. Additionally, they stress the existing need to adapt agent technology to this new environment composed of heterogeneous devices connected through heterogeneous networks. So, for agent-based computing to become a widely accepted technology for developing AmI systems, it is necessary to deal with some specific **challenges**.

C1 Modeling agents for AmI systems: The complexity of programming with agent technologies for lightweight devices should be alleviated by providing the capacity to express AmI domain concepts at a higher level of abstraction and by providing adequate development tools that automate some of

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

the developers tasks. The purpose of these tools is to simplify the agent programming, augmenting the production of applications for AmI systems, regardless of the platform specific characteristics (e.g. device type, communication protocol or wireless technology). As we have discussed in the introduction, MDD seems to be the most suitable technology to address this challenge. Some approaches have already proved the benefits of MDD in agents [Agüero et al., 2009; Hahn et al., 2009; Pavón et al., 2006]. With MDD it is possible to design an agent-based AmI system specifying high level concepts in a platform-independent agent model (focusing on the domain model), and later automatically transform it for different implementation models, bridging the gap between design and implementation. So, one challenge is to propose novel MDD processes for the automatic generation of agents that can be executed in heterogeneous AmI devices. However, the application of this solution to develop agents for AmI systems presents the following challenges regarding the MDD process:

C1.1 *Facilitate the high level modeling of AmI features:* Surveys of AmI technologies [Cook et al., 2009; Sadri, 2011] shows that most current agent-based AmI systems provide ad hoc solutions, without considering the high level modeling of AmI properties. In an MDD process these domain specific properties should be specified as part of a metamodel. This metamodel should model both the generic properties of agents and the AmI specific characteristics. Although there are many agent metamodels (e.g. PIM4Agents [Hahn et al., 2009], FAML [Beydoun et al., 2009],...), rich enough to represent the domain specific concepts of different application domains, they have to be extended in order to incorporate the specific properties of AmI environments, like context awareness or self-management.

C1.2 *Facilitate the extensibility of the MDD process:* The generation process of agents must consider the continuous emergence of AmI devices with new operating systems (e.g. most recently, Android) and the agent platforms for these devices. So, a crucial requirement for an automatic generation process of agents for AmI, is to facilitate the extensibility

of the process to incorporate new AmI technologies. The challenge is to define a process for the automatic generation of agents capable of being executed in diverse lightweight agent platforms from a single domain model, rather than defining several transformation processes for different target agent platforms (e.g. Jade-Leap or μFIPA-OS).

C2 Efficient embedding of agents in heterogeneous AmI devices: Approaches based on the embedding of agents in AmI devices usually propose ad-hoc solutions, normally specific to a particular AmI domain (e.g. AAL). The advantages provided by agents embedded in devices: (i) agents provide services customized to a devices resources [Stock et al., 2007]; (ii) agents encapsulate private data that must be hidden from other agents in the MAS [Muñoz et al., 2003] (e.g. privacy of critical data); (iii) to vary components (i.e. agents) of the architecture without modifying the architecture of the system [Cook et al., 2006]; (iv) it can be considered a more flexible approach to model open systems, because it enables modeling genuine decentralized MASs. So, in this thesis we focus on a decentralized approach, where agents are embedded in heterogenous devices, which communicate through a variety of communication protocols; whose execution should be efficient, considering the scarce resources of some AmI devices (e.g. sensors). In order to cope with this challenge, several specific challenges were identified:

C2.1 *Manage device and agent platform heterogeneity:* The majority of AmI systems are composed of a heterogeneous set of devices. But, current agent technologies for small devices normally only run in some of them, and cannot interact with agents deployed in different agent platforms [Ayala et al., 2013b]. This is an important limitation, since it prevents the development of some AmI systems using agents, in the case that an AmI device is not supported by the chosen agent platform. Also, it should be possible for agents of the same MAS to be executed and able to interact through heterogeneous agent platforms and devices.

C2.2 *Cope with wireless network diversity:* Normally AmI devices can interact through different wireless technologies. For example, nowadays,

most mobile phones include WiFi, Bluetooth or IrDA interfaces. Current agent platforms for AmI are limited by the use of only one network interface, and are not designed to be easily extended to adopt new wireless technologies [Bellifemine et al., 2001; Muldoon et al., 2006].

C2.3 *Achieve efficiency of the generated code:* In an AmI system based on agents, agent execution in lightweight devices, such as smartphones, tablets or sensors, with a limited set of resources must be possible. Then the code generation process for a given target agent platform must manage these limitations in order to produce resource efficient code, particularly in regard to energy.

C3 Self-managed agents: Normally, the majority of AmI devices show symptoms of degradation, such as energy loss or failure of some network nodes, which requires explicit management action, for example saving energy to guarantee the system's survival. So, self-management is of great importance to AmI systems. But, the main challenge is to achieve a seamless integration of the natural autonomy property of agents and the self-management capacity required by AmI systems. But, where and how the self-management property should be incorporated inside agents are specific challenges that are addressed in this thesis:

C3.1 *Decentralized self-management* The highly decentralized and embedded nature of AmI devices makes it hard to enforce some forms of centralized direct control over each of the networked devices. This makes traditional centralized approaches that use one agent, or a fixed set of agents to perform any kind of self-management task, inadequate and economically unfeasible. So, the challenge here is to propose innovative decentralized solutions for self-management. The proposed solution should consider that in AmI, simple devices (e.g. a sensing unit) coexist with sophisticated ones (e.g. tablets), so the challenge is to generate different kinds of self-management solutions adapted to the necessities and hardware resources of each device.

C3.2 *Modeling and implementing self-managed agents* The challenge is to

provide AmI system designers with high level abstractions to explicitly specify self-management politics as part of the source metamodel of the MDD process. The advantages of achieving this challenges are: (i) explicit modeling of self-management property to improve the reasoning capacity about self-management and its relationship with the rest of the concerns considered in agent models; (ii) the designer does not have to worry about the implementation details of self-management as part of the agent, since its functionality is automatically generated by the MDD process as stated Challenge C1; (iii) it is possible to check the self-management policies, in order to ensure they are correct before deploying them. However, current agent metamodels lack explicit modeling mechanisms and concepts to adequately cope with the modeling of this feature [Bernon et al., 2005; Beydoun et al., 2009].

1.2 Overview

In keeping with **C1**, we have defined an MDD process that automatically derives agents ready to be embedded in a variety of AmI devices, which can interoperate through a set of FIPA compliant agent platforms and different communication protocols (**C2**); and optionally with the capacity of self-management, specially taking into consideration the limited resources normally available in AmI devices (**C3**).

Figure 1.1 gives an overview of our MDD process. At the modeling level (the top of Figure 1.1, label **Modeling**), in order to address **C1.1** we have defined the Platform Independent Metamodel (PIM), Pineapple, which encapsulates both those concerns specific to MAS (e.g. interaction protocols) and concepts related with the self-management property (Challenge **C3**). The self-management property is modeled separately from the MAS concerns, by means of policy specification languages (addressing Challenge **C3.2**). The basis of Pineapple is the PIM4Agents metamodel, which tries to unify the most common agent-oriented concepts [Hahn et al., 2009]. We have significantly extended the PIM4Agents metamodel with specific modeling elements that facilitate the design of self-management capabilities. These modeling elements are inspired by the APPEL policy language

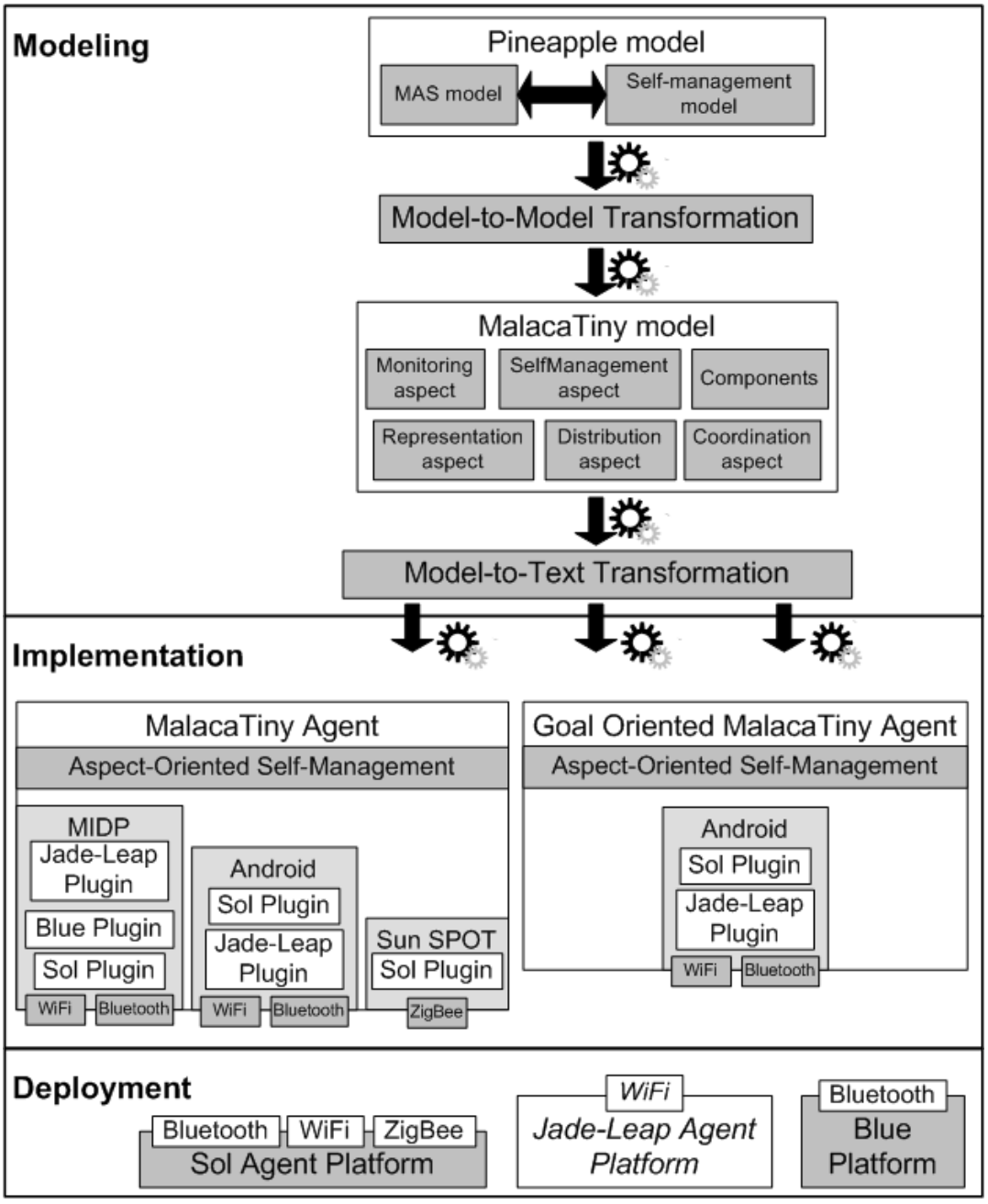

Figure 1.1: Overview of the MDD process of agents for AmI

[Turner et al., 2009] and are contained in a new viewpoint devoted to the self-management. So, the first step of our MDD process, is the modeling of an AmI system using Pineapple. Note that, as we explained in **C3.1**, a single MAS can be composed by agents with different self-management capacities. So the specification of the self-management viewpoint depends on the specific self-management requirements of the AmI device where the agent is to be deployed.

Challenge **C1.2** in our process is addressed by the definition and implementation of a Platform Specific Metamodel (PSM) of a platform-neutral agent model (MalacaTiny), i.e. an agent model not dependent on a specific agent toolkit/agent platform. The use of MalacaTiny simplifies the MDD process compared with other model driven solutions for agents. Typical MDD processes for agents require two transformation processes, i.e. a model-to-model (M2M) and a model-to-text (M2T) , for each target agent model of the process [Gascueña et al., 2012] (e.g. Jade-Leap metamodel). In our solution, instead of generating an agent that can only run in one agent platform (e.g. in Jade-Leap) our process generates agents capable of being executed in any FIPA-compliant agent platform for lightweight devices. So, we have simplified the MDD process as we only need two transformation processes, independently of the number of target agent platforms. The platform neutrality of the MalacaTiny model is achieved by using aspect-orientation concepts. That is, platform specific concerns (e.g. distribution, representation, etc.) are modeled separately be means of *aspects*, so it is possible to generate agents that are adapted to the target agent platform by weaving the correct *aspects* (addressing **C2**).

Once the designer has modeled the AmI system in Pineapple he/she executes the M2M transformation rules, and automatically generates the set of agents composing the AmI system. We have implemented the M2M transformation rules from Pineapple to MalacaTiny in ATL. We have also evaluated the benefit of applying a model driven approach by assessing the degree of automation [Harrington and Cahill, 2011] of our MDD process. The results show that it is possible to automatically generate approximately 40% of the code in complex AmI systems.

Regarding the self-management property, some AmI devices, such as motes (i.e. sensing units) are very simple, and it makes no sense to implement complex self-management policies inside such a device. So, our MDD process can generate

agents with different self-management capacities; all of them being interoperable (fulfill **C3.1**). For those MalacaTiny agents with simple self-management necessities, a set of M2T transformations rules are executed, generating the implementation of simple and reactive MalacaTiny agents. For agents requiring more sophisticated self-management, the M2T transformation rules generate a goal-oriented implementation, more suitable for implementing the self-management property by means of high-level objectives (fulfill **C3.2**). We consider that an agent is goal-oriented when the actions of the agent are the consequence of pursuing goals.

In this implementation model (label **Implementation** in Figure 1.1), the agent can adjust its operation in the face of changing circumstances and in the face of hardware or software failures. Both implementations use aspect-orientation in order to improve the modularization of the agent internal design. So, MalacaTiny agents are composed of components and aspects, which contribute to the enhancement of the adaptation, reuse and maintenance of the agent architecture. Furthermore, this approach addresses **C2.3** (efficiency of the generated code) since: (i) it generates agent implementations adapted to devices' resources: reactive implementation is simple, does not consume too much energy, and goal-oriented implementation consumes more resources, but could perform a more sophisticated management of device's resources; (ii) the code generated by the M2T transformations is optimized for the target platform. We have evaluated the efficiency and performance of MalacaTiny by comparing it with Jade-Leap and the results have demonstrated that the internal design of MalacaTiny is very efficient. Using our framework even over another agent platform like Jade-Leap has very little or no penalty in resource consumption. Currently, the different versions of MalacaTiny can be embedded in multiple devices and deployed in different FIPA-compliant agent platforms using multiple network technologies. Specifically, MalacaTiny agents can be executed in Android devices, devices with MIDP profile (mobile phones that support Java ME [Oracle, 2013]), and Sun SPOT sensor motes [Labs, 2013].

Aspect-orientation also helps us to cope with the Challenge **C2.1** (*Manage agent platform and device heterogeneity*) and the Challenge **C2.2** (*Cope with Wireless network diversity*). The distribution aspect encapsulates how to use and access the message transport service (MTS) for message delivering, so its implementation

depends heavily on the services offered by the agent platform and the transport protocol used. This aspect hides platform dependencies, and makes the rest of the classes of the agent architecture (components and aspects) independent from the agent platform (Challenge **C2.1**) and the communication service (Challenge **C2.2**) used at runtime. Moreover, it is possible that an agent communicates with other agents through different agent platforms, by simply instantiating the correct distribution aspect for each agent platform (e.g. Jade-Leap) implemented as a plug-in.

By improving the modularization of the agent architecture with aspect-orientation, the addition of new agent platforms therefore only entails implementing a new plug-in. In order to illustrate the effort of adding a new agent platform in our approach and also to address **C2.2**, we have implemented *Blue*, an agent platform for communicating with Bluetooth. Also, to address **C2.2** we have implemented *Sol*, an agent platform that supports the deployment of AmI systems composed by a set of self-configuring agents running embedded in heterogenous devices. The main features of *Sol* are the support for communication of agents in heterogeneous devices (Challenge **C2.1**), coping with heterogeneous transport protocols (WiFi, Bluetooth and ZigBee) and group communication often required by AmI systems (Challenge **C2.2**).

Finally, the agents generated by the MDD process can be deployed in lightweight implementations of FIPA-compliant platforms (bottom of Figure 1.1, label **Deployment**). Thanks to the capacity of self-management it is possible that an agent embedded in an AmI device (e.g. smartphone) can self-configure the agent platform and/or communication protocol (only with *Sol* agent platform) used in each moment, depending on the resources available (e.g. memory, energy, etc.). In order to show the benefits of the self-configuring capacity of MalacaTiny agents, we have conducted a set of experiments that show the benefits of changing the communication protocol in order to save energy or battery. We have implemented different self-management goals like, extending the life of the system, recovering from some faults, etc. We have used the process presented here to develop an Intelligent Museum deployed for the ETSI Informática, and different versions of Intelligent Transportation Systems (ITSs) and Ambient Assisted Living Applications (AAL).

1.3 Contributions

In this section we will enumerate, what we consider to be the most relevant contributions of this work.

1. We have developed a metamodel for agents called Pineapple that includes specific concepts for supporting self-management [Ayala et al., 2011b, 2013c]. In particular, we concern ourselves with the specification of policies to describe when and how to adjust the behavior of the agent and the MAS, by means of agent roles. The foundation of Pineapple is the PIM4Agents metamodel. Additionally, we have integrated concepts of the APPEL policy language for policy specification. This enables the validation of the policy using tools available for APPEL, e.g. the UMC model checker.

2. We have defined the MalacaTiny metamodel [Ayala et al., 2013b]. This is a platform-neutral agent metamodel that uses aspect orientation to separately represent application-specific functionality from the communication-related concerns.

3. We have defined an MDD process that generates MalacaTiny agents [Ayala et al., 2010a,b, 2013b]. We have implemented an M2M transformation process between Pineapple and MalacaTiny, and a set of M2T transformation processes that generate code for the different version of MalacaTiny. The use of MalacaTiny (at modeling and at deployment) makes the MDD process more extensible. MalacaTiny agents are platform-neutral, so with the same MDD process we can generate agents that can be deployed in different agent platforms.

4. We have defined a family of aspect oriented agent implementations for lightweight devices named MalacaTiny [Ayala et al., 2011a, 2013a,b]. MalacaTiny agents can be executed in Android devices, Sun SPOT sensor motes and mobile phones that have support of J2ME. MalacaTiny agents are platform neutral and can be deployed in different agent platforms such as Jade-Leap, *Sol* and *Blue*.

5. We have defined a goal-oriented version of MalacaTiny that can be executed in Android devices [Ayala et al., 2012d]. This agent architecture exploits the capacity of Java for Android with regard to reflection. The aspect composition process of Goal-Oriented MalacaTiny is different to the reactive MalacaTiny and has an extensible joint point model.

6. In order to overcome the limitations of current agent platforms for lightweight devices, we have developed the *Sol* agent platform [Ayala et al., 2012a,b]. The main features of *Sol* are the support for multiple network technologies (ZigBee, WiFi, Bluetooth) and group-based communication. Additionally, in order to test the possibilities of the platform-neutrality, we have developed an agent platform based on Bluetooth, called Blue, and a distribution aspect for Jade-Leap.

7. We have integrated self-management inside agent architectures [Ayala et al., 2011a, 2012b,c,d, 2013a]. The self-management is taken into account in the different phases of the development process. Therefore, it can be specified and validated at modeling and even the self-management related code can be generated using our MDD process.

8. Within the scope of this thesis, we have developed different applications and prototypes that we have used to test and validate the application of the agent technology for AmI environments. Using the Jade-Leap agent platform, we have studied the application of the agent technology for the implementation of intelligent transport systems [Amor et al., 2009, 2010]. Using the MalacaTiny technology and focusing on communication between heterogeneous devices, we have developed a fall detection system that uses an Android device and a Sun SPOT sensor mote [Ayala et al., 2011a, 2013a]. Additionally, we have developed an application for the "Museo de la Informática" of "E.T.S.I. Informática" of Málaga [Ayala et al., 2012a].

1.4 Structure of the Thesis

Following this introduction, the first chapter of this thesis explains to the reader, the fundamental principles used in the proposed solutions, explaining some background concepts, the state of the art and related work. Then, our proposal is presented in the following four chapters. Finally, in the last part, the validation and discussion of our approach is presented together with some conclusions and future work.

Background

This chapter provides the background necessary to understand the rest of this thesis. The special characteristics of Ambient Intelligence are detailed. Additionally, we describe its most important application areas and show some examples of real AmI systems. Then, we analyze how agent technology is applied to AmI and survey agent platforms for Ambient Intelligence. Self-management is another important part of this thesis and is described in this chapter. We pay particular attention to the Autonomic Computing approach of IBM. The last part of this chapter is dedicated to Model Driven Development. We describe the foundations of this technology and review the contributions of the Model Driven Development to the agent paradigm. Finally, Model Driven technologies used for the implementation of some of the contributions of this thesis are presented.

A metamodel for self-managed agents

In this chapter, we present the case study that we are going to use to describe the different contributions of this thesis, an Intelligent Museum. Additionally, we describe the Pineapple metamodel and how to model an AmI application using this metamodel. This chapter shows how to model the application and the self-management functionality. Additionally, we describe how to validate the self-managed behavior of MalacaTiny agents using the UMC model checker.

From Pineapple to MalacaTiny

This chapter begins by presenting the MalacaTiny metamodel. Then, a detailed description of the model-to-model transformation process from the Pineapple metamodel to MalacaTiny is given. We distinguish three parts to our M2M process: transformation of the agent architecture; generation of aspects; and generation of self-management functionality.

Code generation of MalacaTiny agents

The implementation of the different versions of MalacaTiny agents and the model-to-text transformation process for them are described in this chapter. MalacaTiny agents can be executed on mobile phones with MIDP profile, Android-enabled devices and Sun SPOT sensor motes. Architectures of these agents present minor differences, so we focus on the architecture of agents with MIDP profile. We then describe the internal design of Goal Oriented MalacaTiny and how it differs from the other versions of MalacaTiny. In addition, this chapter tackles how to integrate self-management inside agents using aspects. Finally, the model-to-text transformation process for MalacaTiny and Goal Oriented is presented.

The communication concern

In this chapter we are going to present the two distribution aspects that we have developed specifically for MalacaTiny and Goal Oriented MalacaTiny agents, *Blue* and the *Sol* agent platform. *Blue* is a distribution aspect, specific to mobile phones with MIDP profile that we developed to study the energy consumption of MalacaTiny agents. The *Sol* agent platform is our solution to communicate heterogenous devices with heterogenous communication means. This platform has been specially important for the development of the Intelligent Museum.

Validation

The validation of certain aspects of this thesis thesis is presented in this chapter. We present and discuss some results with regard to the automatic generation process of agents for AmI systems and the energy consumption and performance of

MalacaTiny. Additionally, we present the results of the self-management functionality and distribution aspects implemented for MalacaTiny; Jade-Leap, *Blue* and the *Sol* agent platform.

Conclusions

This chapter summarizes the proposal explained throughout this thesis highlighting the contributions of our work. Furthermore, we detail the main publications derived from these contributions. Then, a section discussing lessons learned is also provided. Finally, we outline our prospective future work.

Appendix A: Resumen

This appendix presents a Summary of this thesis in Spanish.

Appendix B: Conclusiones

This appendix presents the conclusion, publications and future work of this thesis in Spanish.

Chapter 2

Background

This chapter provides the necessary background in order to understand the challenges that our approach must meet and the solutions proposed as part of this dissertation. First of all, we provide an overview of AmI and the use of agent technologies in this application domain (Section 2.1). Secondly, focusing on implementation issues, we provide an overview of the agent technologies that can be used for the implementation of AmI systems, placing particular emphasis on agent toolkits and platforms (Section 2.2). Thirdly, Section 2.3 introduces the self-management concept, which inspired our solution for our self-managed agents. Next, regarding to the solution proposed in this thesis, we will present the fundamentals of MDD and surveyed MDD approaches for agents (Section 2.4). Finally, in Section 2.5, we will review the basic principles of AOSD.

2.1 Ambient Intelligence

The term AmI was coined by the ISTAG advisory group, when launched the AmI challenge [Ducatel et al., 2001] in 2001. AmI is the vision of a future in which environments support the people inhabiting them. AmI systems are naturally distributed and normally composed of a myriad of devices interconnected through a great diversity of communication technologies. In AmI environments the traditional computer input and output media disappears, and processors and sensors are integrated in everyday objects.

Table 2.1: Technological issues in AmI systems.

Domain	*Distributed*	*Heterogeneous Devices*	*Multiple Communication Technologies*	*Agent technology*
Smart home	100%	93%	85%	43%
AAL	80%	70%	50%	40%
Healthcare	89%	78%	56%	56%
ITS	43%	71%	14%	14%
Education	87.5%	75%	12.5%	25%
Business	80%	40%	20%	30%
Leisure	87.5%	62.5%	25%	62.5%

The main goal of the solution proposed in this thesis is to automatically generate agents that can be used as building blocks for AmI systems, shortening development times. In order to demonstrate both the importance and benefits of this work, we have conducted a study of the AmI literature with the following objectives: (i) to show the notable importance of AmI systems in different application domains; (ii) to know which are the main features of AmI systems that differentiate them from other systems; (iii) to analyze the use of agents in AmI systems until now, in order to identify their potential benefits and find out current challenges that agent technology may face.

Table 2.1 summarizes the results from analyzing sixty-seven AmI systems [Cook et al., 2009; Sadri, 2011]. These systems have been grouped by their application domain (typical of AmI), i.e. Smart homes, AAL, Healthcare, ITS, Education, Business, and Leisure (first column). The second column "Distributed" specifies the percentage of systems that present a distributed architecture, the third column specifies the percentage of systems composed of sets of heterogeneous devices, the fourth column shows the percentage of systems whose devices must communicate using different wireless technologies. The last column labeled "Agent Technology" represents the percentage of studied systems that apply the agent technology. In following subsections we are going to explain the results of this table.

2.1.1 AmI applications

The first column of Table 2.1 depicts the application of the AmI that we have considered in our study. Nowadays, there is a large variety of AmI systems, and additionally, new ones are continuously appearing. We have selected this applications because they are considered the most well known example of AmI. In this section, we provide an overview of these type of applications.

Smart homes or home automation are an important application domain for AmI. These are houses equipped with special devices to assist its inhabitants, e.g. detecting risk situations or making the building more comfortable. An Smart home is composed of three main components: a set of sensors, a set of activators for controlling sensors and other equipment, such as fridges, windows, and so on. Additionally, it is required computing facilities to which the sensors and activators are linked. One remarkable example of Smart home is the Essex intelligent dormitory, iDorm [Hagras et al., 2004].

AAL is an application of the AmI intended to assist handicapped and elderly people suffering from all kind of disabilities, e.g. neurological alterations, spontaneous fractures and falls or sudden changes in blood pressure [Nehmer et al., 2006]. AAL has a great economical importance due the increase in longevity of the population and the cost of maintenance of national health systems [Corchado et al., 2008]. There are different types of AAL systems, these can be provide services indoor or/and outdoor. Additionally, AAL systems have different purposes, they can provide emergency treatment services, enhance the autonomy of handicapped persons and makes his/her life more comfortable. One recent example of AAL system is the iFall application [Sposaro and Tyson, 2009], which is an emergency service that can work indoor and outdoor. This application is a fall detection system embedded in a mobile phone, that makes emergency call in the case a fall of its user is detected.

Healthcare applications intend to adapt AmI ideas to medical environments. Applications of AmI in hospitals can vary from enhancing safety for patients and professionals to monitor the evolution of patients after surgical intervention. Many of the AmI developments for Smart homes can be adapted for its use in hospitals. AmI have already improve the quality of health care in different ways, e.g. physi-

cians can assist or perform surgeries via remote robots [Riva, 2004] or to detect early cardiac events [Gouaux et al., 2002].

An *ITS* is an AmI environment that improves transportation systems of all kinds. With this aim, different elements of the transportation system (users, vehicles, and elements of the transport infrastructure such as bus stops and bus stations, to name a few) incorporate different kinds of lightweight devices (sensors, smartphones, tablets, etc.) in order to provide valuable and useful services for the vehicles occupants for a journey. In most cases, this information is obtained from diverse information sources that are located in different kinds of devices and locations, mainly external to the vehicle. It is interesting to mention the I-VAIT project [Rakotonirainy and Tay, 2004], whose aim is to assist drivers gathering information of elements of the car (pressure on breaks) and processing driver's face expression.

AmI can help improve the learning experience of student means of its applications in *education*. Educational institutions can use AmI technologies with different purposes, e.g. to track student progress and attendance to class, to asses the attention of students and enable devices to focus their attention [Shi et al., 2003], to support speaker speech [Franklin, 1998] or to improve collaborative learning [Lank et al., 2004], just to mention a few.

The application of AmI in *business* is intended to facilitate interaction in workplace environments. Workers need to focus on the project at hand without being tripped up by technology. There are interesting project that applies AmI to business, that includes retrieving and projecting information according to workers gestures and activities [Cook and Das, 2004], supporting decision making processes in business [Marreiros et al., 2007] or manufacturing process planning [Shen et al., 2005].

AmI has a great potential to support activities related with *leisure* such as shopping, tourism or culture. In leisure, AmI applications offer personalized tailor made services and information to users in open environments. Additionally, these environments are usually responsive and react to the presence of customers according to their identities and profiles. Example of these application are Easishop [Keegan et al., 2008] and the results of the Peach project [Rocchi et al., 2004].

2.1.2 Ambient Intelligence characteristics

As it is stated in the Section 2.1.1, there are a great variety of AmI systems. An AmI system can be a house equipped with special devices to assist its inhabitants, or a set of projectors and lights in a class room that support the learning process, or a set of mobile phones that negotiates the best offer in a commercial area according to user preferences. Despite this diversity, most of AmI systems shares the following characteristics [Sadri, 2011]:

- *Invisible*: Technology is embedded in daily living objects like mobile phones, clothes, watches, etc.
- *Mobility*: Services provided by the AmI application does not require that user is still. In some cases hardware is mobile and in others is the software.
- *Context-awareness*: AmI applications sense the environment for useful information and provides services according to the current context.
- *Anticipatory behavior*: AmI systems act on their own behalf without explicit request from user.
- *Self-management*: AmI systems must be able of reacting to all kinds of abnormal exceptional situations in a flexible way without disruption of their service. More detail of self-management are given in Section 2.3.

In order to integrate these characteristics in real environments different solutions has been adopted. From the results shown in Table 2.1 we can see that the greater majority of AmI systems are distributed, with a mean of 81%. It is striking that some AmI systems are not distributed, as is the case of ITSs (only 43% are distributed). In the latter case, the reason is that most transport systems integrate monitoring devices on vehicles simply to assist drivers in different circumstances. Secondly, a significant percentage of AmI systems are composed by heterogenous devices (e.g. home appliances, hand held devices, sensors, onboard computers), 69.9% is the mean. So including an open set of heterogeneous devices can be considered a must for any technology that is presented for use in AmI systems.

Thirdly, most AmI systems, apart from the traditional communication technologies, make use of new wireless communication technologies such as Wi-Fi, Zigbee, Bluetooth, Wireless USB or Z-Wave, since these AmI systems are mostly deployed in personal area networks (PAN) . So, the diversity of communication technologies seems to be an important requirement for the technologies used to develop AmI systems. Lastly, 38.64% of the AmI systems surveyed use agent technologies. This can be interpreted in two ways. This rate is close to 50%, showing a significant presence of agent technology in these environments. On the other hand, a percentage below 50% also shows the necessity of improving current agent technologies in order to achieve a wider adoption in the domain of AmI systems. Note that these results were taken from surveys that are not specific to agents, so we assume the results are unbiased towards agent technology. Of course, there are other papers, and surveys more specific to AmI and agents [Chen and Cheng, 2010; Tapia et al., 2010], that show that agents are the most appropriate technology to be used in AmI, but this assertion can be considered slightly biased because of the interest in applying agent technology to AmI.

2.1.3 Use of agents in AmI systems

In this section, we are going to analyze those AmI systems developed with agents. Our goal is to identify the challenges that AmI systems pose, for agent technology. In the literature we found that agents can be used as abstractions to model and implement both functionality and devices of an AmI system [Penserini et al., 2005], to encapsulate artificial intelligence techniques [Augusto and Nugent, 2004], and to coordinate the different elements that compose the application [Haigh et al., 2006]. Although the reasons and purposes that justify the use of agents are different, they contribute to highlight the suitability of the agent paradigm in AmI environments. Most AmI systems developed with agents and MASs are relative to leisure activities, which includes applications to aid shopping and tourist activities. These AmI applications demand context aware behavior, by providing services that must be adapted to meet user preferences and current context conditions. Also, these AmI systems are executed in open and highly dynamic environments, where nodes of the application are continuously appearing and disappearing. So, the conclusion

is that agents are particularly well suited for managing context-awareness and the dynamism of open systems.

Among the agent-based AmI systems surveyed, we have not found a common approach regarding the agent solution adopted. Some AmI agents were developed using general-purpose agent toolkits for AmI (such as Jade-Leap), while numerous systems developed their own special-purpose agents specific to an application domain, or to a specific device. In some of these approaches agents execute inside the device and are coordinated through tuple space mechanisms [Penserini et al., 2005]; or agents are mobile and move from one device to another [Keegan et al., 2008], most of them adopt an entirely distributed and decentralized architecture. Additionally, by embedding agents in AmI devices, these special purpose MASs are able to adapt agents to the specific device's characteristics (regarding hardware and software resources) in which it is running [Muñoz et al., 2003]. These solutions are the preferred approach in some AmI environments because of the advantages provided by agents embedded in devices [Caire et al., 2002]: (i) agents provide services customized to device resources [Stock et al., 2007]; (ii) agents encapsulate complex functionality that must be hidden from other agents in the MAS [Muñoz et al., 2003] (e.g. keep privacy of critical data); (iii) to vary components (i.e. agents) of the architecture without modifying the architecture of the system [Cook et al., 2006]; (iv) it can be considered a more flexible approach to model open systems, because it enables modeling genuine decentralized MASs [Ayala et al., 2012b].

However, the approaches based on the embedding of agents in AmI devices usually propose ad-hoc solutions, normally specific to a particular AmI domain (e.g. AAL). But, the adoption of such ad-hoc solutions imposes severe limitations on interoperability and extensibility:

- agents embedded in AmI devices can not be easily reused in other AmI systems;
- the emergence of new AmI devices, including the software or hardware updating of the devices may require developing agents from scratch;
- it is not easy to modify the embedded agent in order to incorporate new communication technologies;

- it is normally not possible to construct an AmI system composed with agents running on heterogenous devices communicating with heterogeneous technologies at the same time;

- because of the minor use of embedded agents in AmI devices, the existence of tools, documentation and integration in well-known agent-oriented methodologies is reduced (unlike general-purpose agents toolkits for AmI).

We therefore conclude that these are limitations and deficiencies that hinder a wider adoption of the agent technology. In Section 2.2, we present agent technologies and toolkits that can be used for the development of AmI applications.

2.2 Agent technologies for AmI systems

In this section we analyze the currently available general-purpose agent toolkits for developing AmI systems. There are a number of approaches that facilitate both the development and the execution of general purpose agents in lightweight devices, contributing to the adoption of agent technology for AmI systems. These agent approaches involve an agent development toolkit, and also the necessary infrastructure for the management and communication of the agents which run embedded in lightweight devices and communicate through an agent platform.

Jack [Howden et al., 2001] is a mature, cross-platform environment for building, running and integrating commercial-grade MASs. Jack is a compact and efficient BDI implementation that runs on any system where Java is available (PDA, high-end multi-CPU enterprize servers, PC...). In addition, Jack provides an IDE that eases the development process of a MAS based on Jack using a graphical notation that enables code-generation. Jack is extremely lightweight and is designed to handle thousands of agents running on relatively low-end hardware, but Jack does not run in Android nor in devices with MIDP profile or in sensors.

Jade-Leap is the lightweight version of the Jade [Bellifemine et al., 2008] that allows the execution and communication of Jade agents in PCs, PDAs, Android devices and MIDP devices. Like Jade, Jade-Leap offers a management GUI and other tools that facilitate the development and maintenance of the MAS.

μFIPA-OS [Laukkanen et al., 2002] is an extension of the FIPA-OS approach, which has been designed from the bottom up. However, it can only be executed on PDAs with Linux or PocketPC operating systems. Similarly to Jade, this agent platform provides a set of tools to monitor agent platform execution.

The 3APL-M [Koch et al., 2006] framework defines a programming language for implementing BDI agents. This agent technology provides different versions for execution in different devices as does Jade-Leap.

Agent Factory Micro-Edition (AFME) [Muldoon et al., 2006, 2008] is a FIPA compliant framework for the fabrication of BDI agents that can run in MIDP devices and Sun SPOT sensors. This framework has tools to manage the agent platform and a compiler that eases the development of AFME for the different devices that can support it. However, the communication between agents running in heterogenous devices (i.e. between sensors and MIDP devices) is not possible. AFME agents for MIDP only communicate using a remote mail service (using HTTP confections), while AFME agents for Sun SPOT only use the Sun SPOT radiogram protocol. Therefore, despite AFME supporting different transport protocols, these are independently supported for its different agent versions.

There other approaches only intended for embedding agents in Android devices. Andromeda [Agüero et al., 2009] extends Android services to embed agents that are directly based on the Android infrastructure. The μ-Agent approach [Frantz et al., 2012] tries to augment the functionality of the Java for Android API using Agent Oriented Software Engineering (AOSE) principles. The focus of μ-agent is agent organizations and it is possible to have coalitions of agents running inside our application. This agent technology has an agent platform named μ^2, that offers communication services. JaCa-Android [Santi et al., 2011] is an approach used to implement applications in Ubiquitous Environment and the Internet of Things using BDI agents. It uses the joint work of Jason, an agent programming language rooted on a strong notion of agency, and Cartago, an environment programming framework. Another interesting approach is Jadex micro agents [Pokhar, 2013], which is a reactive agent architecture that focus on performance and low memory footprint. Jadex micro agents are intended to interact with former Jadex BDI agents [Braubach et al., 2005] that provides a BDI infrastructure to Jade.

Additionally, there are agent approaches solely for wireless sensor networks

(WSNs) [Vinyals et al., 2011], although most of these approaches focus on mobility features. Following this trend, we find Agilla [Fok et al., 2005], ActorNet [Kwon et al., 2006], MAPS [Aiello et al., 2011] and MASPOT [Lopes et al., 2011]. They are agent platforms or middlewares intended for the development of applications in wireless sensor networks based on mobile agents. Agilla and ActorNet have been implemented on the MICA2 and TinyOS platform, while MAPS and MASPOT on Sun SPOT sensor motes.

As you can see from this related work (see Table 2.2), there is no agent approach that supports the three kinds of devices that we find in AmI environments (i.e. PC, personal devices and sensors) or in the case of AFME, does not offer an interoperable solution. Additionally, although FIPA compliance is a feature that has been taken into account in most approaches, interoperability between all of them is not possible.

Another important concern is the extensibility of the agent approaches. Nowadays, the appearance and disappearance of new devices and wireless communication technologies is becoming usual. In recent years we have seen how some of them have became obsolete (such as IrDa[1]) while new ones have appeared (e.g. NFC[2] and Wi-Fi Direct[3]). However, it is rather complex to extend current agent platform message transport services or incorporate new functions into agents to support new wireless communication capabilities. Such extension is not straightforward and depends on the programmers' expertise in certain agent platforms. More flexible agent architectural design should allow software agents to cope with the evolution and emergence of new technologies and devices (e.g. new transport services, message encoding, etc.), boosting the production of adequate versions of agents for different lightweight devices.

We have seen in this section that current agent approaches for the development of AmI systems present serious limitations and deficiencies in coping with the technological requirements concerning the implementation of these systems. As a solution to these issues we propose MalacaTiny agents, that try to overcome the limitations present in other agent approaches for AmI environments.

[1] http://en.wikipedia.org/wiki/Infrared_Data_Association
[2] http://en.wikipedia.org/wiki/Near_Field_Communication
[3] http://en.wikipedia.org/wiki/Wi-Fi_Direct

Table 2.2: Agents for AmI systems

Agent Platform	*Device*	*API*	*Agent platform management GUI*	*IDE*	*Transport protocols*	*Wireless technology*	*FIPA*
Jack	PC, PDA	Java	Yes	Yes	TCP	802.11	No
Jade-Leap	PC, MIDP, PDA, Android	Java, Java ME, Java for Android	Yes	No	TCP	802.11	Yes
μFIPA-OS	PDA	Java	Yes	No	TCP	802.11	Yes
3APL-M	PC, MIDP, PDA	Java, Java ME	No	No	TCP	802.11	Yes
AFME	MIDP, Sun SPOT	Java, Java ME	Yes	No	HTTP (over TCP), Radiogram (datagram-based)	802.11, 802.15.4	Yes
Andromeda	Android	Java for Android	No	Yes	TCP	802.11	Yes
μ-Agent	Android	Java for Android	Yes	Yes	TCP	802.11	No
JaCa-Android	Android	Java for Android	No	No	TCP	802.11	No
Jadex	PC, Android	Java, Java for Android	Yes	Yes	TCP	802.11	Yes
Agilla	Mica2	NesC	No	Yes	Radiogram (datagram-based)	802.15.4	No
ActorNet	Mica2	NesC	No	Yes	Radiogram (datagram-based)	802.15.4	No
MAPS	Sun SPOT	Java, Java ME	No	No	Radiogram (datagram-based)	802.15.4	No
MASPOT	Sun SPOT	Java, Java ME	Yes	No	Radiogram (datagram-based)	802.15.4	No

2.3 Self-management

Normally, the majority of AmI devices show symptoms of degradation, such as energy loss or failure of some network nodes, which requires explicit management action, for example saving energy to guarantee the systems survival. Consequently AmI systems demand the reconfiguration of their internal functioning in response to changes in their environment. This means that AmI systems must behave as autonomic systems with a self-management capacity.

Self-management is a concept that brings together many fields of computing with the purpose of creating computing systems that self-manage [Huebscher and McCann, 2008]. For this purpose, complex computing systems should be able to independently and according to a set of rules or policies, take care of the regular maintenance, configuration and optimization tasks, thus reducing the workload on the system administrators. In order to incorporate self-management, the system should support the so-called autonomic functions [Dobson et al., 2010]: self-awareness, self-situation; self-monitoring and self-adjusting. Self-awareness is the capacity for introspection, while self-situation is related to the awareness of current external operating conditions; self-monitoring refers to the ability to detect changing circumstances in the agent environment. Finally, self-adjusting is the ability to accordingly adapt to these environment changes. These functions support the so-called self-* properties: self-configuring, self-healing, self-optimizing and self-protecting.

Self-configuring requires the system to configure itself according to high-level goals, i.e. by specifying what is desired, not necessarily how to accomplish it. This can mean being able to install and set itself up based on the needs of the platform and the user. Self-healing entails that the system detects and diagnose problems, such as hardware failure or software problem, and it should attempt to fix them. Fault tolerance is and important aspect of self-healing. Self-optimizing involves the system is able to proactively optimize its use of resources, improve its performance or quality of service. Finally, self-protecting means that the system protects itself from malicious attacks.

The Information Technology industry has recognized the importance of self-management and launched different initiatives [Kephart, 2005]. Hewlett-Packard's

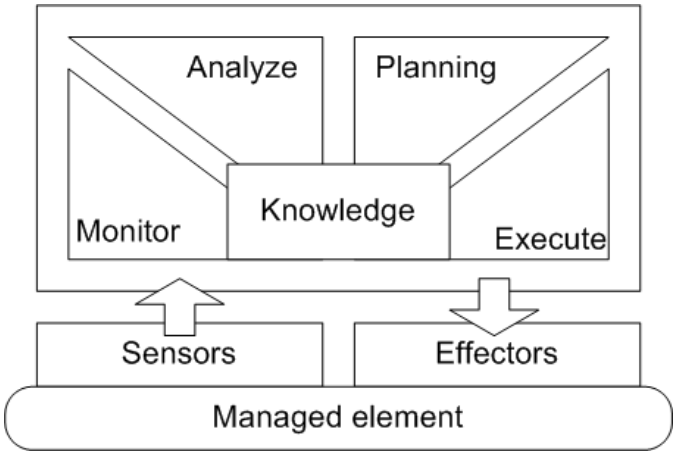

Figure 2.1: The MAPE-K control loop.

has launched Adaptive Enterprise initiative and Microsoft the Dynamic Systems initiative. However, one of the most remarkable initiatives is the IBM Autonomic Computing that was launched in 2001 [Horn, 2001]. A key element of any autonomic system is the Autonomic Manager (see Figure 2.1). The AM is conceived as a software component that ideally can be configured by human administrators using high-level goals for self-managing any software or hardware resource (i.e. the managed element or resource). The AM includes sensors and effectors: Sensors collect information about the managed element, while effectors carry out changes at the managed element. The AM uses the monitored data from sensors and internal knowledge of the systems to plan and execute, based on these high-level goals, the low-level actions that are necessary to achieve these goals.

To perform self-management, the functionality of this component is divided in four main functions: Monitor, Analyze, Plan, and Execute (MAPE) . These four functions, which share knowledge, are used to describe the architectural aspects of autonomic systems, and obey an intelligent control loop known as MAPE-K (K is for Knowledge). This loop is similar to (and probably inspired by) the generic agent model proposed by Russell and Norvig [Russell et al., 1995], in which an intelligent agent perceives its environment through sensors, and uses these percepts to determine actions to execute on the environment.

The behavior of the analyze component is ruled by policies that can be specified

in terms of Event Condition Action (ECA) , goals or utility functions [Kephart and Walsh, 2004]. An ECA policy dictates the action that should be taken whenever the system is in a given current state. A goal policy specifies either a single desired state, or one or more criteria that characterize an entire set of desired states. A utility function policy is an objective function that expresses the value of each possible state.

2.4 Model Driven Development and Agents

MDD is an approach for Software Development where models are now first class citizens of the software development process, and even the code is managed as a model. This technology advocates generating software systems from high-level models using model transformation languages. MDD is a different concept to Model-Driven Architecture (MDA) . The MDA[1] is an initiative promoted by the Object Management Group (OMG) , which offers a conceptual framework for defining a set of standards in support of MDD [Selic, 2003]. Some of the standards related with the MDA initiative are the Unified Modeling Language (UML) , the Meta Object Facility (MOF) or the Common Warehouse Metamodel, just to mention a few.

MDD ideas can bring important benefits to the development of MAS as shown in [Amor et al., 2005; Giorgini et al., 2005; Pavón et al., 2006]. With MDD it is possible to specify a MAS in a platform-independent model, focusing on the domain model, and later transform it automatically to different design or implementation models, bridging the traditional gap between design and implementation.

In following subsections, we are going to provide an overview of MDD and present works that relates MDD and agents. Firstly, Subsection 2.4.1 introduces the foundation of MDD. Subsections 2.4.2 and 2.4.3 surveys related work in metamodels and MDD for agent technologies. Finally, Subsection 2.4.4 overviews model definition and transformation languages.

[1] http://www.omg.org/mda/index.htm

2.4.1 Foundations of Model Driven Development

MDD is an approach for Software Development that promotes the use of models and metamodels to formally represent domain concepts. A model is the representation of the function, structure and behavior of a system within a given context, and from a specific point of view. These artifacts are expressed in a language that exists at some abstraction level, whose syntax is described means of metamodels. A model conforms to its metamodel in the way that a computer program conforms to the grammar of the programming language in which it is written.

A metamodel specifies the concepts and their relationships for the purpose of building and interpreting models [Hahn et al., 2009]. The MDA initative consider three level of abstraction and depending on the level a metamodel receives different names and has different purpose in the development process [Pastor et al., 2008]. In the most abstract level, a metamodel is a Computation Independent Metamodel (CIM) and focused on the environment and system requirements. CIMs do no take into account computation issues related with the system being modeled. PIMs have a higher level of abstraction and consider the computation that system have to perform. However, it still does not consider the technological platform that will support the implementation. Finally, PSM support the description of systems attending to the specific characteristics of the platform will support it.

One important contribution of MDD is that a software system is obtained through the transformation of different metamodels defined at different abstraction layers, automating the process of constructing a target model for a given source model. Therefore, developing model transformations describes an important aspect in MDD. A model transformation is a transformation of one or more source models to one or more target models, based on the metamodels of each of these models. In order to transform models, it is necessary define mappings between metamodels that define the syntaxes of the transformed metamodels. These mappings describes how one, or more elements in the source model should be transformed to the target model. When the complete transformation from the source model to the target model is defined, the target model could be automatically generated.

In a MDD process, transformations are usually between model with different

levels of abstraction, i.e. we transform a CIM model in a PIM model and a PIM model in a PSM model. A last step of the transformation process is the generation of the code related related with the PSM metamodel.

Exists different technologies for supporting model definition and transformation. Some of them also consider its integration in frameworks like the Eclipse Modeling Framework. An overview of these technologies is given in Subsection 2.4.4.

2.4.2 Metamodels for agents

The boom in MDD has encouraged the development of metamodels for agents [Bernon et al., 2005]. They has mainly been developed to support specific agent technologies [Braubach et al., 2005] or engineering approaches [Giorgini et al., 2005]. Additionally, there are a metamodels that try to get a common representation of what agents and MASs are [Beydoun et al., 2009]. As part of this thesis dissertation, we present the Pineapple metamodel, which is intended for agent-based AmI applications with self-management capacities. Then, in this section we are going to introduce agent metamodels and modeling approaches related with AmI and self-management. Although the majority of agent technology contributions to the AmI focus on the implementation level, there are interesting approaches that focus on the modeling level as in this thesis dissertation [Sadri, 2011].

One of the most remarkable examples of the agent-based modeling for AmI is the approach presented in [Penserini et al., 2005], which uses Tropos to design an interactive tourist guide in an Intelligent Museum. In this proposal, role descriptions have been used to deal with the dynamic functionality of an AmI application. In [Molesini et al., 2010], the SODA methodology is used to test how to design an agent-based application for controlling an intelligent home. This contribution focuses on the use of agents to design the coordination of a distributed and intelligent system with complex interaction patterns. Related with the AmI, the work [Morganti et al., 2009] uses the MAS theory to model a home automation system. On the other hand, Agent-π [Agüero et al., 2009] is a metamodel to design ubiquitous applications embedded in Android devices. In these approaches, the usage of agents to model the application is justified because of the distributed and dynamic

nature of AmI systems. Agent methodologies offer advanced mechanisms to design distributed interaction and adaptable behavior. However, these approaches do not deal with any self-management capability of the agents in the AmI domain.

An interesting approach that puts together MAS and self-management is MaCMAS [Peña et al., 2007]. This approach uses agent-based modeling techniques and software product lines to model software systems that evolve over time. This evolution of a system is expressed by means of roles that an agent performs in a given time to achieve a set of goals. Roles are designed as a set of features that can be composed using different operators. MaCMAS has been applied to model the behavior of autonomous robots, showing the potential for modeling other AmI systems. However, this work focuses only on the evolution of the system architecture, and does not offer mechanisms to validate the self-managed behavior. Other proposal that relates agents and self-management is [Trencansky et al., 2006]. This work uses the Agent Modeling Language [Červenka et al., 2005] to design the entities that control and manage self-managed systems, which are also known as autonomic managers. However, these agents are not able to manage themselves as our agents do. Moreover, agents and autonomic computing concepts have been used to model self-managed business processes [Greenwood and Rimassa, 2007].

Finally, there are approaches for the systematic modeling of self-managed systems in general. Although they do not consider particularities of the agent domain, they take into account typical agent's concepts like reactive behavior or distributed interaction [Vassev and Hinchey, 2009; Weyns et al., 2012]. ASSL [Vassev and Hinchey, 2009] provides a framework for the specification, validation, and code generation for self-management of complex systems. The self-managed behavior is provided by wrappers promoting the separation of concerns. This separation of concerns is also approached by FORMS [Weyns et al., 2012], a reference model for the formal specification of distributed autonomic systems. In particular, it uses a reflective modeling perspective that, like our metamodel, allows reasoning about the architecture of the self-managed system. Although both approaches use a formal notation, they do not consider mechanisms to detect policy conflicts.

2.4.3 Model Driven approaches for agents

As stated before, MDD is an advanced software technology that can naturally address the generation of agents for diverse agent platforms, by means of transformations between CIM, PIM and PSM. There are some MDD agent-based approaches that consider the CIM, sometimes called a domain model [Brossard et al., 2011]. The CIM level is mainly used in agent approaches to analyze and represent the requirements of the system. With the CIM level it is possible to explore and specify the problem domain of AmI systems using agents. However, MDD processes that provide a general purpose solution for agents in AmI (as we do), contribute to the solution space and not to the problem or domain space. So, normally these approaches do not consider the CIM level, closer to an AmI specific application or domain. So, a straight forward definition of a general purpose MDD process for agents would use a generic agent metamodel such as PIM, one PSM for each agent platform, and define M2M transformations for each PSM. In order to generate code, the subsequent M2T transformations must also be defined.

Most of the most important agent-oriented methodologies apply MDD to the development of MAS. One of the most relevant MDD approaches for agents is the DSML4MAS approach. A general overview of this MDD process is shown Figure 2.2. The MDD process of the DSML4MAS approach uses a generic PIM for agents named PIM4Agents [Hahn et al., 2009] to generate Jack and Jade agents. The generation process is composed of 4 transformations: two M2M transformations to generate Jack and Jade models and two M2T transformations to generate Jack and Jade code. MDE principles have also been applied in [Agüero et al., 2009], specifically to generate agents for Android devices. It takes the agent$-\pi$ as PIM, a metamodel for mobile agents, and defines transformations for two mobile-specific PSMs, Andromeda and Jade-Leap. Similarly to DSML4MAS it follows the MDD straight forward approach depicted in Figure 2.2. With a similar proposal, other agent-methodologies, like Tropos [Giorgini et al., 2005], INGENIAS [Pavón et al., 2006], ASEME [Spanoudakis and Moraitis, 2010], and Prometheus [Gascueña et al., 2012], support the development process of MAS using MDD processes.

Although these MDD approaches improve and automatize the development

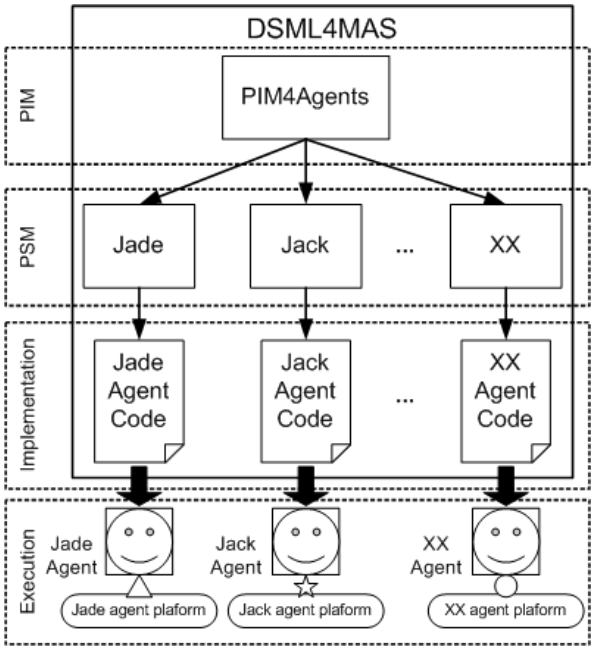

Figure 2.2: Overall picture of the MDD process of the DSML4MAS approach.

process, when generating agents for multiple agent platforms (mainly for Jade), we have found that only one of the methodologies provides explicit support for the generation of code for lightweight devices typical of AmI systems [Agüero et al., 2009]. Although in general terms, MDD improves the adoption of new technologies just by developing a new code generator, the fact is that, for all the MDD approaches considered, a different set of transformations must be defined and implemented from the corresponding PIM (specific to each MDD process) for each agent platform (i.e. PSM). This means that the cost of extending the proposal with a new agent platform (PSM) is very high. Although this is acceptable for some application domains, for AmI systems, which are composed by a set of heterogeneous devices with different agent technologies interacting with each other, this is a serious limitation. Including a new agent platform for a specific device, e.g. as in the DSML4MAS approach, requires the definition of two new sets of transformation rules: a M2M transformation from PIM4Agents to the metamodel of the new agent platform; and a M2T transformation from the new agent platform metamodel into code. This is a very complex task, sometimes impossible to

perform properly due to: (i) the metamodel of the target agent framework must be available, which is not always the case; (ii) sometimes the target metamodel is not specified completely, and some mappings for the target metamodel are carried out in an ad-hoc manner; (iii) this task also requires some expertise in a transformation language (e.g. ATL); and (iv) also the transformations from the target platform metamodel to code have to be implemented, requiring an in-depth knowledge of the target agent's implementation framework.

2.4.4 Model Driven Development technologies

As stated before, the MDA initiative has defined different standards in support of the MDD. In this thesis dissertation, we have mainly used technologies provided by the Eclipse Modeling Framework (EMF) [Steinberg et al., 2009] and other related projects, like the openArchitectureware [Haase et al., 2007]. The EMF is a part of the MDA in the Eclipse family tools [Moore et al., 2004]. This framework was started as an implementation of the MOF standard and has evolved into a set of tools that facilitate model definition and transformation. Specifically, we have used the following tools: Ecore [Foundation, 2013a] for metamodel specification, ATL [Jouault et al., 2008] for M2M transformation and Xpand [Klatt, 2008] for M2T transformation. Although there are outstanding technologies and frameworks for MDD, like Acceleo [Company, 2013], we decided to use this technologies because they are mature, widely supported by their users community and can be integrated together easily.

2.4.4.1 The Ecore languague

Ecore is the model used to describe metamodels or class modeling concepts in the EMF. This model has its roots in MOF and UML, and was designed to map cleanly to Java implementations. Actually, Ecore is a small and simplified subset of the UML and it is considered the implementation of the MOF standard in the EMF. Ecore acts as its own metamodel, so it is defined in terms of itself. In Figure 2.3, we can see the Ecore metamodel of the Kernel Ecore metamodel (i.e. only the main classes of the the full ecore metamodel).

Essentially, in order to define a Ecore metamodel we have four types of objects.

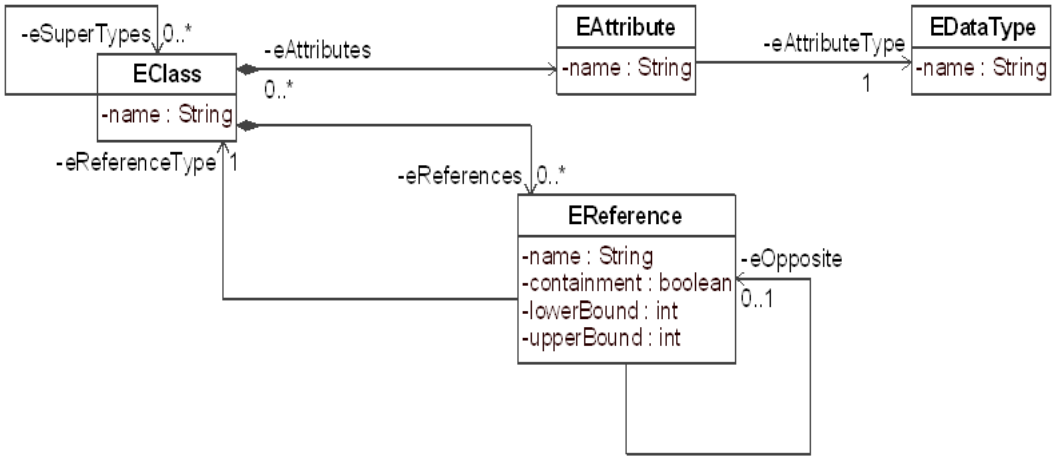

Figure 2.3: Metamodel of the Ecore metamodel.

1. *EClass* models classes themselves. Classes are identified by name and can contain a number of attributes and references. To support inheritance, a class can refer to a number of other classes as its supertypes.

2. *EAttribute* models attributes, the components of an object's data. They are identified by name, and they have a type.

3. *EDataType* models the types of attributes, representing primitive and object data types that are defined in Java, but not in EMF. Data types are also identified by name.

4. *EReference* is used in modeling associations between classes; it models one end of such an association. Like attributes, references are identified by name and have a type. However, this type must be the EClass at the other end of the association. If the association is navigable in the opposite direction, there will be another corresponding reference. A reference specifies lower and upper bounds on its multiplicity. Finally, a reference can be used to represent a stronger type of association, called containment; the reference specifies whether to enforce containment semantics.

The concepts expressed in this metamodel should be quite familiar to modelers and object-oriented programmers. When we describe a class, we describe its

attributes, which are modeled using EAttribue, and it references, modeled with EReference.

2.4.4.2 The ATL language

ATL is a domain-specific language for specifying M2M transformations. It is considered a hybrid language because it provides a mix of declarative and imperative constructs. This language has an execution environment based on Eclipse that can works with metamodels described in Ecore. The ATL development environment provides tools to support the major tasks involved in using the ATL language: editing, compiling, executing, and debugging. ATL is inspired by the OMG QVT requirements [Group, 2013b] and builds upon the OCL formalism [Group, 2013a].

ATL transformations are organized in modules. A module contains a mandatory header section, import section, a set helpers and a set of transformation rules. The header sections gives the name of the transformation module and declares source and target metamodels. Helpers and transformation rules are the constructs used to specify the transformation functionality.

ATL helpers can be viewed as the ATL equivalent to methods. They make it possible to define factorized ATL code that can be called from different points of an ATL transformation. Helpers are only defined for the source metamodels since target models are not navigable. Each helper is characterized by its context, its name, its set of parameters and its return type. The context of a helper defines the kind of elements the helper applies to, i.e. the type of the elements from which it will be possible to invoke it. Example 2.4.1 depicts an ATL helper that return true when type object of the metamodel Families whose class is Member (remember ATL works with Ecore) is female. The context of this rule is *Families!Member*, its name is *isFemale* and the return type is *Boolean*.

Example 2.4.1. ATL helper.

```
1 helper context Families!Member def:isFemale() : Boolean =
2     if not self.familyMother.oclIsUndefined() then
3         true
4     else
5         if not self.familyDaughter.oclIsUndefined() then
6             true
```

```
7         else
8             false
9         endif;
10    endif;
```

Transformation rule is the basic construct in ATL used to express the transformation logic. ATL rules may be specified either in a declarative style (matched rules) or in an imperative style (called rules or action blocks). In order to understand contributions of this thesis, in this Section we are only consider ATL matched rules. An ATL matched rule is a mean to specify the way target model elements must be generated from source model elements. An ATL matched rule is applied to each element of an specific class that optionally holds some condition. For this purpose, a matched rule enables to specify: (i) which source model element must be matched; (ii) the number and the type of the generated target model elements, and (iii) the way these target model elements must be initialized from the matched source elements. Example 2.4.2 depicts an example of matched rule related with Example 2.4.1. This rule is applied to each element whose class is *Member* and the result of the application of the *isFemale()* helper is true. Then, it is generated an element whose class is *Male* from the *Persons* metamodel. The *fullName* attribute is initialized with the concatenation of the *firstName* and the *familyName* of the input element.

Example 2.4.2. ATL matched rule.

```
1 rule Member2Male{
2         from
3             s:Families!Member (not.isFemale())
4         to
5             t:Persons!Male(
6                 fullName <- s.firstName+' '+s.familyName;
7             )
8     }
```

There is another type of matched rules named lazy rules, that has the same notation as matched rules but are indicated with the keyword lazy. In order to be invoked, a lazy rule must be invoked by other rules, i.e. it is not automatically

matched like matched rules. They are applied on a single match as many times as it is referred to by other rules, every time producing a new set of target elements.

2.4.4.3 The xPand language

The Xpand domain specific language was developed as part of the openArchitectureWare project devoted to M2T transformations. Now, it is an important part of the Eclipse Modeling Project, within other M2T technologies like Jet [Foundation, 2013b] or Acceleo [Company, 2013]. Xpand is a template engine, that has unique properties for generating code from models like type safety and polymorphic dispatch [Friese, 2010]. Like other technologies related with the EMF, Xpand is fully integrated with Ecore metamodels. Additionally, it provides an integrated development environment and tools to write and validate code generations.

A code generator in Xpand consists of the input metamodel and a set of templates, which transform the model into code. With Xpand, we can generate any kind of textual transformation, so it is also used to generate manuals and other documentation artifacts. The Example 2.4.3 depicts an example of an Xpand template, which generates an HTML form. Xpand allows to insert information from models into code sections, easing the model transformation process.

Example 2.4.3. Xpand template to generate HTML.

```
1 <<DEFINE form FOR Form>>
2 <<FILE name + ".html">>
3 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "
      http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
4 <html xmlns="http://www.w3.org/1999/xhtml">
5 <head>
6   <title><<this.title>></title>
7   <meta http-equiv="Content-Type" content="text/html; charset=UTF
       -8" />
8   <link rel="stylesheet" type="text/css" href="../static/style.css
       " />
9 </head>
```

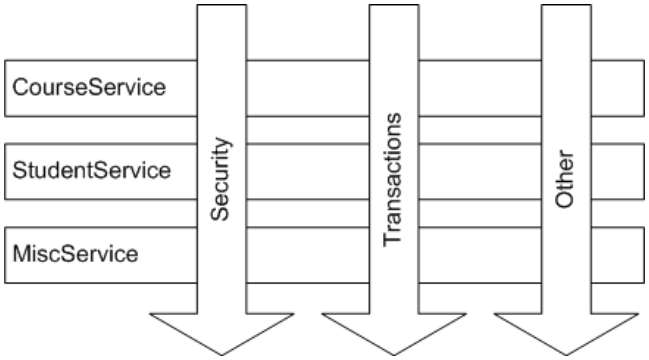

Figure 2.4: Properties that cut across several modules [Walls and Breidenbach, 2005].

2.5 Aspect Oriented Software Development

As stated in the introduction MalacaTiny agents are based on components and aspects. In this section, we will review the basic principles of AOSD. The main motivation of the aspect-orientation [Filman et al., 2004; Kiczales et al., 1997] is to overcome some limitations of traditional software development techniques, such as Object Orientation (OO) or Component Based Software Engineering (CBSE) , due the existence of crosscutting concerns. Crosscutting concerns are properties of an application that can be dispersed in multiple classes. Typical examples are properties such as logging or authentication, whose code is normally scattered in several modules of the system. Even if they are well-encapsulated in a logging or an authentication module, the rest of modules requiring these properties need to include implicit calls to them. A graphical example is shown in Figure 2.4, in which three different services (*CourseService*, *StudentService*, and *MiscService*) encapsulating the main functionality of an application are crossed by extra-functional properties such as the *Security* module and the *Transactions* module, among others.

Aspect-orientation improves the separation of concerns providing the mechanisms for encapsulating each crosscutting concern appropriately in an independent module, called *aspect*, and then specifying how this aspect must be composed with the software modules it crosscuts. The predominant definition for aspects is the one that comes from the AspectJ programming language [Kiczales et al., 2001].

AO principles are: (i) Software base modules (e.g. objects or components) do

not contain any reference or code related to crosscutting concerns (e.g. coordination between agents, persistence, distribution, etc.); (ii) Crosscutting concerns are encapsulated in special modules called aspects (an aspect module contains the implementation of a crosscutting concern, which is called advice in aspect-oriented programming (AOP) terminology); (iii) Each software module permits the injection of crosscutting concerns at well-known and special points of their execution (e.g. after the reception of an input message, or before the execution of a method), which are called join points; (iv) The patterns that specify the set of join points being intercepted for a given application are known as pointcuts (e.g. after the reception of a call-for-proposal message); (v) The composition process in which aspect behavior is injected into the join points specified at the pointcuts is known as the weaving process. The aspect system can implement the weaving at compile, load or run time. The last one is preferred when adaptability requirements (such as for agents in open environments) exist.

Chapter 3

A metamodel for self-managed agents

In this chapter, we to describe the Pineapple metamodel and how to design an AmI application using it. The modeling of our application is in two parts: the modeling of specific functions and interactions of the MAS and the modeling of the self-management behavior. In order to illustrate our approach, we use scenarios in an Intelligent Museum (IM) for the case study.

This chapter is structured as follows: Section 3.1 describes services and agents that compose the IM. Sections 3.2 and 3.3 give details of the Pineapple metamodel and how to use it to model the context-aware application. Subsection 3.3.2 describes the modeling of the self-management functionality of the AmI application. Finally, Section 3.4 summarizes the contributions of this Chapter.

3.1 Case study

In order to illustrate how to use the Pineapple metamodel, we present an IM as a case study. The trend of intelligent museums is to include a considerable number of sensors which can provide very useful environmental and contextual information. Besides this, the museum staff (guides, security, and other members of the workforce, such as maintenance personnel) wear personal hand-held devices (normally smart mobile phones), which are spread throughout the halls and rooms

of the museum, constituting an important source of contextual data that can be used in the development of specific context-aware services. Sensors provide data that can be used by support services that help museum guides and security staff in their work. They can also be used to provide visitors with location-based services. This IM is an AmI application and provides different services for its target users:

1. for the *museum guide*, it provides support for organizing the route inside the museum considering the presence of other groups in the halls and rooms of the museum or the state of a given room, and helps share and disseminate information between the guide and his/her group of visitors;

2. in the case of the *security staff*, the AmI nodes (sensors and personal handheld devices) provide information about both the presence of people in the museum and environmental conditions (temperature, light, humidity,...) and send global notifications to the different groups of people that are in the museum;

3. and in the case of *visitors*, we take advantage of the fact that the majority of people usually bring a mobile phone with them, so our IM system can provide different pieces of location-based information, for example, details about a temporary exhibition.

This AmI system is designed as a MAS the agents of which are embedded in devices (such as tablets, smart phones and mobile phones) that people (visitors, guides and security staff members) bring with them, and inside sensors located in the building. The MAS has four types of agents (see Figure 3.1): (1) the *GuideAgent* agent for the museum guides; (2) the *SecurityAgent* agent that helps the security staff members; (3) the *SensorAgent* for the sensors in the building; and (4) the *VisitorAgent* agent for visitors, where each one is executing in a specific device.

We would like to highlight that this case study was implemented for a real museum that is physically located at the Informatics School-University of Málaga ("E.T.S.I. en Informática"). This museum shows the evolution of informatics technologies over time (e.g. computers, memory, modems, disks, etc.) and its components are spread over different areas of the Informatics School: different

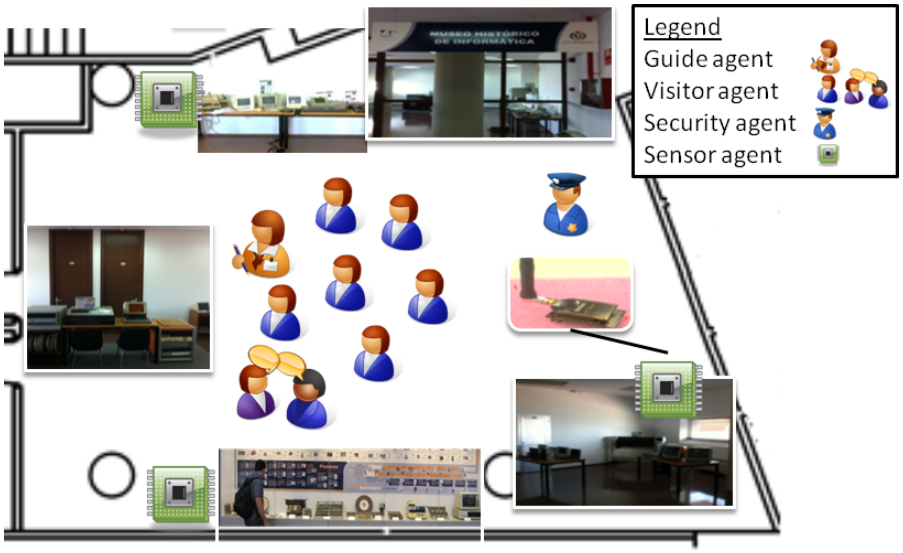

Figure 3.1: Plane of the room 2 of the "Museo de la Informática".

rooms and halls, located on different floors and buildings. Room 1 is located in the school's reception, Room 2 is located in an adjacent service building and its use is exclusively for exhibition purposes (see Figure 3.1). There is another room in the library of the school (Room 3) that contains bibliographic resources, and there are also posters and panels (describing different aspects related to the history of computer science engineering and its most relevant researchers) that are placed in different corridors and classrooms of the school. Sensors are distributed throughout Rooms 1 and 2, while panels and posters are labeled with QR/BIDI codes and NFC/RFID tags. Our scenarios take place mainly in Rooms 1 and 2, where several devices and resources (minicomputers, work stations, servers, multiprocessors, hard disks, memory cards, etc) are exhibited. Figure 3.1 shows the plan of Room 2, and the layout and distribution of elements in the room (agents and exhibited resources).

As stated, the museum includes a considerable number of sensors that form a WSN. In general terms, a WSN consists of a set of sensor nodes monitoring physical or environmental conditions, such as temperature or sound. Sensed data is sent through the network to a device node or a sink node. Source sensor nodes generally operate on a resource of limited power-supply capacity such as a battery, and are

compact and inexpensive. Sink nodes have more resources than source nodes, but they operate on a resource with a limited power-supply capacity. Normally, the majority of sensors show symptoms of degradation, such as energy loss or failure of some network nodes, which requires explicit management action, for example saving energy to guarantee the system's survival. Consequently WSN demand the reconfiguration of their internal functioning in response to changes in their environment. This means that they must behave as autonomic systems with a self-managing capacity.

In our application, each sensor node or device accomplishes different tasks related to the application-specific functionality, e.g. to monitor light levels and send the value to a sink node. Sink nodes receive monitored data, process it, and send it to another device (e.g. a PC). These tasks are performed by agents running inside each sensor node or device. In addition to these tasks, these devices accomplish other tasks related with self-management in order to extend the system's lifetime or recover from the failure of some of the nodes. For example, when any of these situations is detected, sensor agents can allocate additional tasks to agents in sensors with more battery life.

Therefore, apart from their application-specific tasks, agents have to be endowed with additional behavior which constantly checks the context and automatically adapts the agent behavior to changing conditions. The adaptation is ruled by policies that can be specified in terms of ECA, goals or utility functions [Kephart and Walsh, 2004]. An ECA policy dictates the action that should be taken whenever the system is in a given current state. A goal policy specifies either a single desired state, or one or more criteria that characterize an entire set of desired states. A utility function policy is an objective function that expresses the value of each possible state. Policies in the form of goals or utility functions require planning and are inadequate for lightweight devices like sensor motes, due to their limited resources. So, in order to adopt a homogenous approach to model the self-management behavior of the devices in the IM, we choose ECA policies. A description of the rules that comprise the polices for the sensors of our case study is given in Table 3.1 in informal semantics. The first column (*Name*) is the name of the policy, the second column (*Condition*) describes the situation (expressed by a logical expression or the occurrence of an event) that causes the execution of the

Table 3.1: Self-management policies for the WSN.

Name	*Condition*	*Action*
Decrease Sampling Frequency	$batteryLife < 10\% \wedge Active(Task) \wedge F < 0.003 \wedge F > 3.33 * 10^{-5}$	1: Task task=getComponent(ID); 2: double s=getSampling(); 3: task.setSampling(s - X);

Task Allocation	$batteryLife < 10\% \wedge Active(Task) \wedge F >= 0.003 \wedge F <= 3.33 * 10^{-5}$	1: send(REQUEST):
		2: if receive(PROPOSE) and not waiting_confirm then
		send(ACCEPT);
		waiting_confirm:=true;
		3: if receive(CONFIRM) then
		removeRole(Task);

Sink Drop	*Communication exception*	1: send(SINK_DROP) to AMS;
		2: if receive(SINK:YOU) then
		assignRole(Sink);
		elseif receive(SINK:ID) then setSink(ID);
		elseif receive(RETRY) then wait 10 seconds and retry

actions depicted in the third column (*Action*). The first two rules correspond to self-optimizing policies, while, the third one is a self-healing policy.

Additionally, there are policies specifically for agents running in mobile phones. *VisitorAgent* enables a cache for museum contents when the response time of the network is low. *SecurityAgent* can disable the encryption of the information shared between security staff members in order to prolong the device's lifetime in which it is running. These policies are modeled using the same procedure as sensor policies but the differences with the policies for sensors become clear upon implementation. So, this section focuses on policies for sensors which will be looked at again in Chapter 5.

3.2 The Pineapple viewpoints

The Pineapple metamodel, like PIM4Agents, is a generic metamodel that can be used to specify agents of the most representative architectural styles (BDI and reactive), unifying the most common agent oriented concepts used in well-known agent frameworks. A design in Pineapple is structured in eight viewpoints, each focusing on a specific concern of a MAS:

- the *Multiagent* viewpoint contains the main building blocks of a MAS (roles, agents and so on) and their relationships;
- the *Agent* viewpoint describes agents, the capabilities they have to solve tasks and the roles they play within the MAS;
- the *Behavioral* viewpoint describes agent plans;
- the *Organization* viewpoint describes how single autonomous entities cooperate within the MAS and how complex organizational structures can be defined;
- the *Role* viewpoint covers the abstract representations of functional positions of autonomous entities within an organization or other social relationships;
- the *Interaction* viewpoint describes how the interaction in the form of interaction protocols takes place between autonomous entities or organizations;

- the *Environment* viewpoint contains any kind of resource that is dynamically created, shared, or used by the agents or organizations;

- the *Deployment* viewpoint contains a description of the MAS application at runtime, including the types of agents, organizations and identifiers.

- the *Selfmanagement* viewpoint describes the self-managed behavior of agents using roles.

In the following subsections, we explain the concepts that are contained in these viewpoints and how they are used to model agent based self-managed AmI systems. This explanation focuses on how to model the MAS using Pineapple and how to model self-management capabilities. Figure 3.2 depicts the different viewpoints of Pineapple and where the concepts are defined (*is Defined* column), i.e. created in the model, and described (*is Described* column), i.e. where the concepts are filled with information. As you can see, concepts that compose the model of the MAS are defined in some viewpoints and later, they are completed in one or more viewpoints of the metamodel. These viewpoints can be the same viewpoint where the concept was defined or it can be a different one (follow grey arrows in Figure 3.2 to see these relationships). This makes the process not even neither in cascade, and it is possible to work with some viewpoints simultaneously. The following sections describe the concepts included in the different viewpoints and their relationships.

3.3 Modeling of the agent-based application

This section shows how to model the IM in Pineapple, our source metamodel. Before describing the design of the IM, we explain the main concepts of Pineapple and how we use them to specify context-awareness. As stated, Pineapple is based on PIM4Agents, so the different steps that we have to take to model the MAS are similar in both metamodels. Pineapple has been implemented in Ecore, so the resultant design is an XMI document. To make it more readable, we illustrate the use of the metamodel using it as a UML profile.

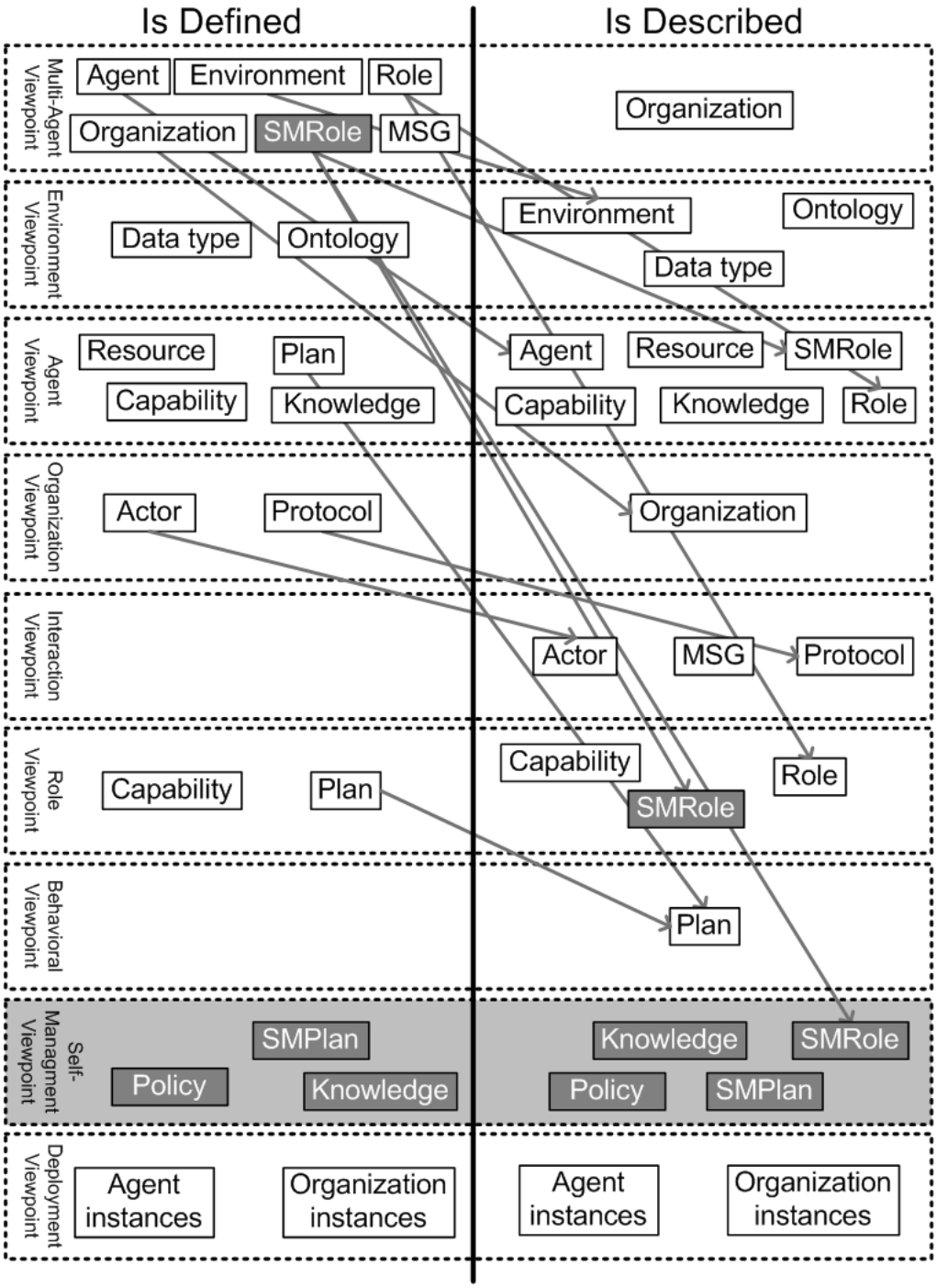

Figure 3.2: Modeling process of the MAS using the Pineapple metamodel.

3.3.1 Multi-agent system design in Pineapple

The first viewpoint in the design process of our application is the MAS viewpoint (see Figure 3.2), which specifies the main building blocks of the MAS (roles, agents, services, organizations and so on) and their relationships. Figure 3.3 shows the MAS viewpoint of the IM system. The representation of the agents, organizations and roles is straightforward in the Pineapple design model. In order for agents to interact, first they must be members of an *Organization* in this viewpoint. Agents *GuideAgent*, *SecurityAgent*, *VisitorAgent* and *SensorAgent*, all interact to provide services and control the museum, and are members of the *IntelligentMuseum* organization. Additionally, *SecurityAgents* and *VisitorAgents* are members of the *PhoneSelfManagement* organization and *SensorAgents* are members of *Monitoring*, *SensorCommunication* and *SensorSelfManagement* organizations. An *Organization* defines the social structure of the MAS, so it includes *Roles* that agents can perform. An agent is considered member of an organization if it performs roles of the organization. These membership relationships are represented by the requires relation of these agents through the depicted roles. This viewpoint also includes a representation of the agent execution environment, which includes the set of objects, data types and functions that can be accessed by agents (represented by the *IMEnvironment* element in Figure 3.3); and the set of messages that are exchanged between the agents within the organization (in Figure 3.3 we see some of them *RouteRequest*, *RouteInformation* and *RouteGenerated*).

In the *Environment* viewpoint the designer can describe the internal information of the system (data and functions). This viewpoint contains the description of the internal components of the agents (such as the set of available sensors and actuators) that provide context data (e.g. a location component); the data types to model the context (e.g. rooms in the museum); and the internal events. The internal components that compose the agent are given in the agent viewpoint.

The *Agent* viewpoint could be the next step in the modeling process (see Figure 3.2) and it deals with the design of the agents internal elements (by means of data, roles, behaviors and capabilities). An agent in Pineapple is an entity that can play particular roles and show various behaviors; and the agent has certain capabilities that group a set of behaviors. The Agent viewpoint shows how agents

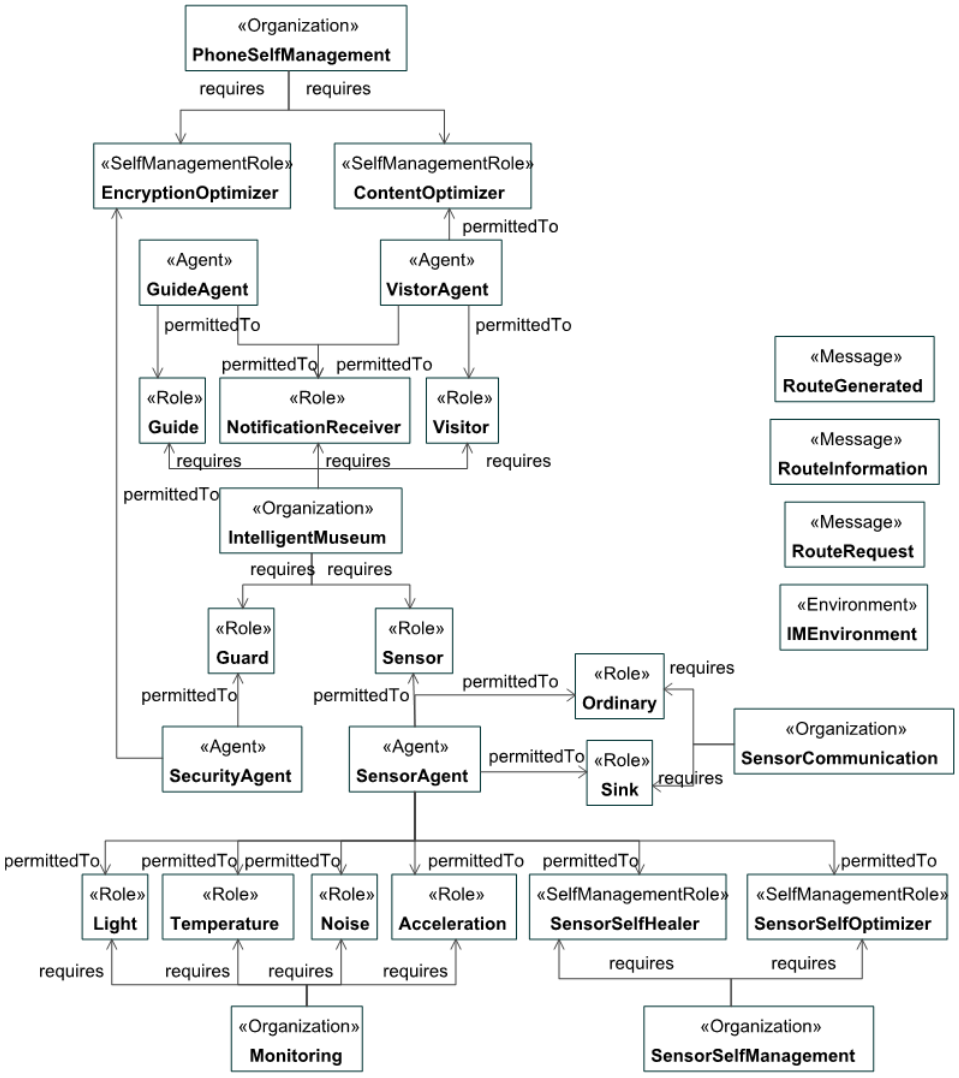

Figure 3.3: UML class diagram corresponding to the MAS viewpoint.

perform the roles included in the MAS viewpoint via the plans that each role has associated with it. Additionally, it is possible to directly assign capabilities to roles. This is specially important when we model self-management capabilities, as we show later. For the IM case study, the *GuideAgent* agent, which plays the role of *Guide* and *NotificationReceiver*, includes 8 different plans to: plan route inside the museum and support route planning of other agents, to control the visit (this includes time required for the visit and information about groups that are reaching the room where the guide is), to send information to its associated group of tourists and to receive notifications from security staff. The *SecurityAgent* agent, which plays the role of *Guard*, includes plans to: send global notifications to visitors and guides, request the room's environmental information and to control the number of people in the IM. The *VisitorAgent* agent plays the roles of *Visitor* and *NotificationReceiver* and has plans for receiving notifications from security agents and information of the guides in the museum, to find a guide to assist the visitor in the museum and receive recommendations for exhibits according to its position in the museum and its interests. Finally, *SensorAgent* is a special case because its roles are related with self-management. So, we address issues related with this type of agent in the next subsection.

The *Organization* and *Interaction* viewpoints deal with the design of interactions from specific perspectives. The IM case study (if we do not consider interaction related with self-management) is composed of 6 Interaction viewpoints and Figure 3.4 shows the interaction viewpoint of the *RoutePlanning* interaction protocol, which covers the interaction between two agents which play the role of guide when suggesting of a certain route inside the museum. This viewpoint shows the set of actors involved (*Planner* and *Supporter*), the set of messages exchanged (*RequestInformation*, *RouteInformation* and *myRoute*), and how the exchange of messages takes place. The *Planner* actor sends a *Request* message (*Request* is the performative of the message) to get the information of the route that the *Supporter* actor is following, then this sends this information in an *Inform* message and finally *Planner* sends the information of the route that it is going to follow. At run-time, the *Planner* and *Supporter* are performed by *GuideAgents*.

The *Behavior* viewpoint describes agent plans. A plan is composed of a set of atomic tasks or actions, such as sending a message, which are related using com-

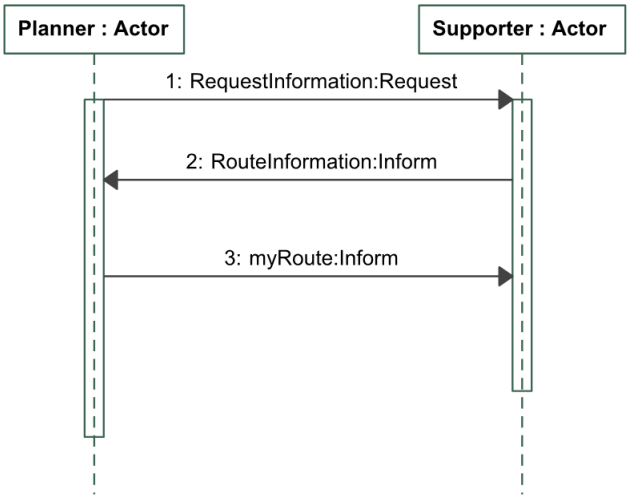

Figure 3.4: UML class diagram corresponding to the Interaction viewpoint.

plex control structures. Additionally, the plan viewpoint shows how information flows between the different actions that compose the plan. The plan *ExhibitRecommender* (Figure 3.5), which is used to recommend exhibits of interest to the user or ones that are new to the musesum, represents a context dependent behavior. Context awareness is normally modeled by defining the elements that fit the context data, and how a change in the context influences the internal behavior of the agent (which is shown in the *Behavior* viewpoint). When the context changes, the knowledge base of the agent is updated with the context value. When the knowledge base is updated, an event is thrown (*KnowledgeBaseEvent* in Figure 3.5) which contains information about the change (the knowledge that has been updated). For this case study the agent context is represented by the position where the visitor is. The event handling is a feature that is not explicitly supported by the PIM4Agent metamodel (the foundation of Pineapple). So, we have solved this limitation by using an atomic task named *ReceiveEvent* that shows that an event (namely a resource) has been thrown and caught. Once the event has been caught (i.e. when an event is received), a subsequent action (*checkRoom*) focuses on handling this context change, by deciding which room and checking whether there is something interesting for the user because of its profile or the novelty of the exhibit. The decisions *checkUserProfile* and *checkUpdates* refer to

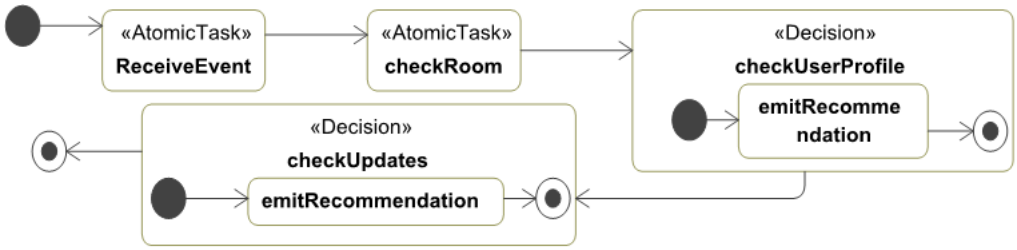

Figure 3.5: UML class diagram corresponding to the *Behavior* viewpoint.

the decision to emit a recommendation to the user in this room of the museum.

Finally, Pineapple also provides the description of deployment information at runtime (such as agents' identifiers and number of instances of a specific agent or an organization) in a viewpoint named *Deployment* viewpoint. In the case of this application, this viewpoint is useless because the number of agents in the system is not known in the modeling. Additionally, our MAS is very dynamic so it is quite common that agents are constantly appearing in and disappearing from it.

3.3.2 Design and validation of the self-management

As stated, Pineapple evolves from the PIM4Agents metamodel, extending it to add a new viewpoint for the modeling of self-management and additionally, existing viewpoints are extended with new self-management related concepts. The new viewpoint, called Self-management, includes concepts of other viewpoints, and is designed to explicitly model self-management policies and the roles involved in self-management functions. However, the modeling of self-management also implies the incorporation and representation of new resources, knowledge and behaviors, which are described in the corresponding viewpoints. Figure 3.6 provides a high-level description of the relationships and dependencies of existing viewpoints (large rectangles labeled viewpoint) and concepts (small rectangles) of the base metamodel and the Self-management viewpoint (grey rectangle) and related concepts (*Policy* and *SelfManagementRole* entities). In what follows, we explain the concepts that are contained in this viewpoint and how they are used to model self-managed AmI systems.

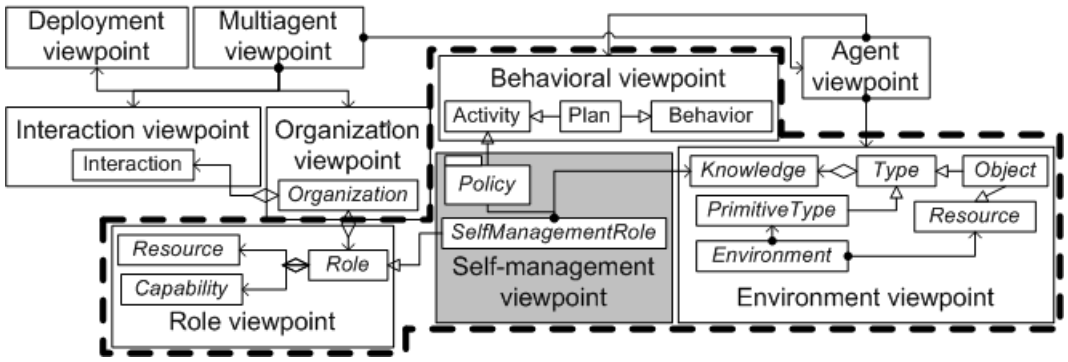

Figure 3.6: Relationship between the Self-management and base metamodel viewpoints.

3.3.2.1 Organizations for self-management

As can be seen in Figure 3.2, the first step in the design of the self-management is to include organizations for self-management. The concept of organization is present in several metamodels [Bernon et al., 2005] and can be used to separate the different concerns of the application. Therefore, we have used organizations in order to integrate self-management inside the MAS. Then, in our MAS we have 4 organizations: *IntelligentMuseum* that covers the communication needs of the MAS; *Monitoring* that encapsulate roles to monitor the environment; *SensorCommunication* which tackles communication in the WSN using sinks and ordinary sensor nodes; and *SensorSelfManagement* and *PhoneSelfManagement* that encapsulate the self-management functionality.

The roles contained in *SensorSelfManagement* organization implement the policies depicted in Table 3.1. So, we have considered two roles: *SensorSelfHealer* and *SensorSelfOptimizer*. The modeling of these policies has influenced the design of the other organizations for *SensorAgents* (*Monitoring* and *SensorCommunication*). Table 3.1 envisions a MAS the agents of which have a dynamic functionality. A role encapsulates a set of behaviors (i.e. capability), resources and participation in interaction protocols. An agent can accomplish some roles at runtime or indeed stop fulfilling a role. Therefore, we have encapsulated the dynamic functionality of sensor agents using roles that belong to different organizations. We have a single type of *SensorAgent* (see Figure 3.3) that belongs to three organizations via the

roles it performs.

At this stage of the modeling, organizations and roles are a good option for integrating self-management because organizations promote the separation of the application concerns and roles deal with dynamic functionality. Although, by using these concepts we could model policies for distributed systems like AmI applications, we would still be lacking an important concept which should be integrated within roles, which is the knowledge required by the role. For example, if an agent is the sink of the WSN, i.e. it performs the *Sink* role in the *SensorCommunication* organization, it must store knowledge about the radio addresses of the agents it receives data from. However, in most agent metamodels (included PIM4Agents), knowledge can only be associated with agents, and not with roles [Bernon et al., 2005]. Therefore, agents all have the same knowledge, regardless of the roles they perform. Since an agent can adopt different roles at runtime, it would be better for it to include the knowledge associated with the role, and not only the behavior in the modeling stage. In our opinion this is not a serious limitation, but it is not conceptually correct. In order to overcome this, a new role is defined, *SelfManagementRole*, which extends *Role* (in this way we extend the metamodel and overcome this limitation only for self-management behavior, while *Role* semantics remain the same). For example, the description of the *SensorSelfHealer* role (see Figure 3.3) would be included in the *SelfManagement* viewpoint. The policy associated with this role is given in the third row of Table 3.1.

The second viewpoint related with self-management is the *Environment* viewpoint. In this viewpoint the designer describes the internal information of the system which is relevant for self-management, e.g. the internal components of the agents. So, here we define data types associated with self-management and later, we use them to define the knowledge belonging to the *SelfManagementRoles*. For example, the knowledge required by the *SensorSelfHealer* role concerns the agent that it must locate in case of a sink failure.

In self-managed systems, it is important to distinguish between policies and activities that support the application of policies. In Pineapple, these activities are modeled using capabilities associated with the *Role* concept (see Figure 3.6), and so it is possible to model a *SelfManagementRole* without policies. For example, we can have an organization with two roles, *Slave* and *Master*. The function

of the *Slave* is to periodically send "is alive" signals to the *Master*. While the function of the *Master* is to restore a *Slave* if in a given time span it does not receive an "is alive" signal. So, the actions of the role *Slave* are intended to support self-management, and its execution does not depend on an event or a condition. A typical action contained in capabilities is the monitoring behavior for self-management.

3.3.2.2 Policies using APPEL notation

The next step in our modeling approach is to define the policies for self-management. In order to model policies, one option is to use the existing viewpoints that are used to model the dynamic behavior of the system, which are the *Interaction* and the *Behavior* viewpoints. However, the principal limitation on using the *Behavior* viewpoint for the modeling of agents' plans is that it does not have support for conflict detection. Conflicts between policies arise when two or more policies can be applied at the same time and those contain actions that can cause conflict, e.g. one policy requires an increase in the frequency of a task, while the other states that the frequency of the same task must be decreased. To resolve this, we propose using a similar approach to the one presented in [ter Beek et al., 2009]. This approach uses an ECA policy language named APPEL, which has been used for policy specification in environments related with AmI (e.g. management of sensor networks or ambient assisted living). The main advantage of APPEL for us is that this language has support for conflict detection using the UMC model checker [ter Beek et al., 2011].

We have integrated the use of APPEL into our metamodel using its syntax to define policies (see Figure 3.7) that any *SelfManagementRole* has to follow. Pineapple uses the syntax depicted in Table 3.2, which has being adapted for the agent domain. This domain-specific language, which is based on APPEL, specifies the triggers, conditions and actions of the self-management policy. Triggers can be internal events of the agent (*EventTrigger*), or the sending and reception of a message (*MSGTrigger* and *Type*). The *Condition* concept can be simple or composed as in the APPEL notation, but it is related with agent knowledge (*Knowledge*) or an equality expression over the trigger of the rule (*TriggerPredi-*

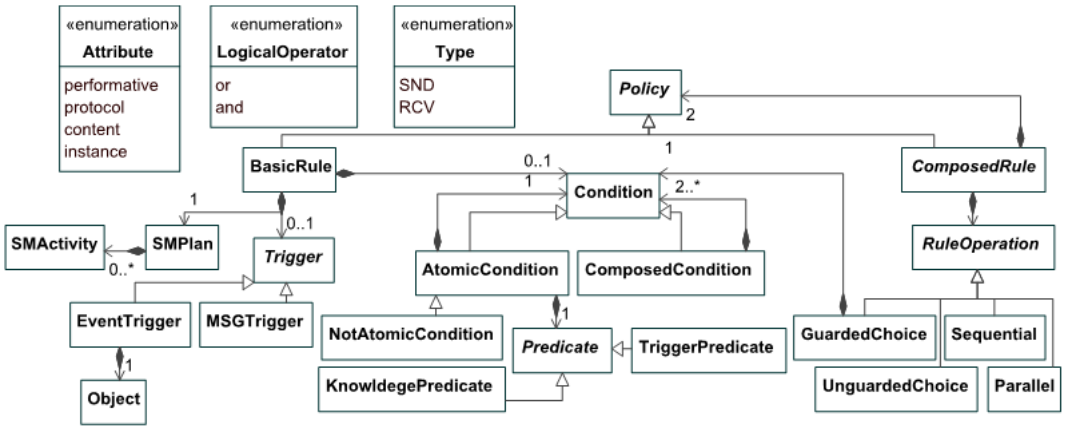

Figure 3.7: Metamodel for Policy concept using APPEL notation.

cate). The *TriggerPredicate* concept represents an equality expression between a value and an attribute (*Attribute* enumeration) that can be the type of the trigger (*instance*) or a field of the message associated with the trigger (*performative*, *protocol* or *content*). Actions associated with policies, i.e. *SMPlan*, are described in the next section. The policy Sink Drop (third row in Table 3.1) can be expressed in APPEL-based notation as depicted in Example 3.3.1.

In order to validate this policy using the UMC model checker, it has been translated to a set of UML finite state machines (FSMs) using the mapping proposed in [ter Beek et al., 2009]. In our approach, policies are simplified before being introduced into the model checker. Specifically, actions have no arguments and logic conditions will be logic variables. So, the policy *Task Allocation* (second row in Table 3.1) is transformed into 4 policies that are composed using parallel FSMs, one for each task that the agent can allocate and logical conditions like $batteryLife < 10\%$ are introduced like the variable *lowBatteryLevel* in UMC. In Example 3.3.2, the code section of the UMC specification for the allocation of the light monitoring can be seen. A UMC specification consists of a set of objects' specifications the behavior of which is described as an FSM. This FSM is described using a set of transition rules the notation of which is $state_A \rightarrow state_B\{trigger[condition]/actions\}$. An FSM, which is in $state_A$, transits to $state_B$ if it receives a *trigger* and a *condition* is met. Triggers, conditions and actions are optional fields in a transition, while origin and destination states are mandatory. When the transition takes

Table 3.2: APPEL syntax.

policy	::=	polRuleGroup |polRuleGroup policy
polRuleGroup	::=	polRule |polRuleGroup op polRuleGroup
op	::=	**g**(condition) |**u** |**par** |**seq**
polRule	::=	[**when** trigger] [**if** condition] **do** action
trigger	::=	trigger |trigger **or** trigger
condition	::=	condition |**not** condition |condition **and** condition |condition **or** condition
action	::=	action |action actionop action
acntionop	::=	**and** |**or** |**andthen** |**orelse**

place, *actions* are executed. Due to the fact states in the FSM can be composed, the state name must be attached to its super states, e.g. in the state name *Top_TaskAllocation.Top_LightAllocation.LightAllocation_wait* from the first line of Example 3.3.2 means the super state of the state *LightAllocation_wait* is *Top_LightAllocation* and *Top_TaskAllocation* is the super state of the latter.

The final step of the specification is to indicate what properties of the FSM are observable, this is done in a special section of the UMC model named *abstractions*. In our case we want to detect whether or not conflicting actions are executed. To do so, we transform these actions into variables and include them in the abstraction section. As an example we check whether an agent can simultaneously remove all its roles for self-management. Then, we include these actions in the abstraction section as depicted in Example 3.3.3.

Example 3.3.1. Sink Drop policy in APPEL notation.

```
1 seq(when CommunicationException do (send(SINK_DROP));
2         par(when MSGTrigger.RCV if Content=SINK_YOU
3             do (assignRole(Sink));
4             when MSGTrigger.RCV if Content=ID
5                 do (setSink(ID));
6             when MSGTrigger.RCV if Content=RETRY
7                 do (wait(10);send(SINK_DROP)););)
```

The UMC framework takes the FSM and transforms it into a doubly labeled

transition system (L^2TS) , which is a formal model of the system's evolution [ter Beek et al., 2011]. This L^2TS can be model checked using logic formulae expressed in UCTL, a UML-oriented-time temporal logic. When the translated model is introduced in the UMC tool, we can check whether or not a conflicting action is executed simultaneously. For example, the expression $AG\neg((RemoveLightRole(true))$ $\&(RemoveNoiseRole(true))\&RemoveAccelRole(true)\&RemoveTempRole(true))$ would be used to express that is not possible to remove all roles simultaneously.

Example 3.3.2. Code section of the Task Allocation policy in the UMC model checker

```
1 Top_TaskAllocation.Top_LightAllocation.LightAllocation_wait ->
    Top_LightAllocation.AllocateLight
2     {-[LowBatteryLife and not LightFrequencyModifiable and
          LightActive]}
3 Top_LightAllocation.AllocateLight.SendLightAllocRequest ->
    AllocateLight.WaitForLightAllocProposals
4     {-/SendLightAllocRequest:=true}
5 Top_LightAllocation.AllocateLight.SendLightAllocRequest ->
    Top_LightAllocation.LightAllocation_fail
6 AllocateLight.WaitForLightAllocProposals.WaitLightAllocProposal ->
     WaitForLightAllocProposals.SendLightAllocAccept
7     {ReceiveLightAllocPropose [not WaitingConfirm] /
          SendLightAllocAccept:=true; WaitingConfirm:=true}
8 WaitForLightAllocProposals.SendLightAllocAccept ->
    LightAllocation_fail
9 WaitForLightAllocProposals.SendLightAllocAccept ->
    WaitForLightAllocProposals.WaitForLightAllocConfirms
10 WaitForLightAllocProposals.WaitForLightAllocConfirms.
    WaitLightAllocConfirm -> WaitForLightAllocConfirms.
    RemoveLightRole
11     {ReceiveLightAllocConfirm/RemoveLightRole:=true}
12 WaitForLightAllocProposals.WaitForLightAllocConfirms.
    RemoveLightRole -> LightAllocation_fail
```

Distributed policies are a special type of policy, which are quite common in AmI systems. These policies require the communication and coordination of different self-management roles. In order to model these policies, we require the viewpoints

to model the distributed behavior of the system (*Organization* and *Interaction*), in addition to the *SelfManagement* viewpoint. An example of a distributed policy is the *Task Allocation* policy (second row in Table 3.1), which requires the agents to interact in order to allocate a task. Firstly, we associate an interaction protocol (*TaskAllocationProtocol*) with the *SensorSelfManagement* organization in the *Organization* viewpoint. Secondly, in the same viewpoint we model the actors required by the protocol (*Requester* and *Responder*) and the roles that these actors can play (*SensorSelfOptimizer* in Figure 3.3). Finally, in the *Interaction* viewpoint, the message exchange between actors is defined and described.

Example 3.3.3. Abstractions section of the Task Allocation policy in the UMC model checker

```
1 Abstractions  {
2     State Rules.RemoveLightRole=$1 -> RemoveLightRole($1)
3     State Rules.RemoveNoiseRole=$1 -> RemoveNoiseRole($1)
4     State Rules.RemoveAccelRole=$1 -> RemoveAccelRole($1)
5     State Rules.RemoveTempRole=$1 -> RemoveTempRole($1)
6 }
```

3.3.2.3 Actions for self-management

The last step in our process is to specify the actions required by the policies. The *Behavior* viewpoint describes plans associated with agents and capabilities of agents and roles. Due to the inadequacy of the *Behavior* viewpoint for modeling some concepts and activities specific to self-management (the Autonomic Functions (AFs) described in Section 2.3), we introduce the concept of *SMPlan* (see Figure 3.7). The *SMPlan* is similar to the *Plan* concept. Like plans, an *SMPlan* comprises a set of actions such as sending a message, which are related using complex control structures such as loops. In the moment of modeling, self-awareness and self-adjusting related actions, which are closely related to the agent architecture and are independent of the application domain, require a common vocabulary to avoid ambiguous specifications. So, it would be better to have specific purpose actions included in the *Behavior* viewpoint and to avoid ad-hoc solutions. For this reason, we have developed the *SMActivity* concept that is shown in Figure 3.8. With the

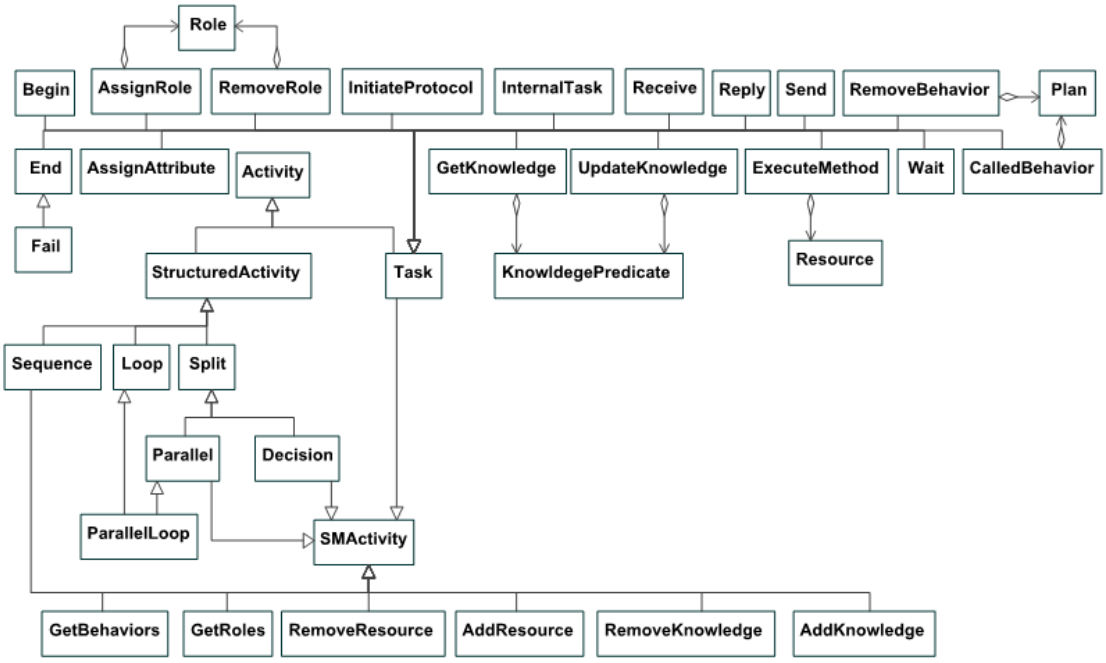

Figure 3.8: Metamodel for *SMPlan* concept.

definition of the *SMPlan* concept, we ensure that self-management actions (i.e. *SMActivities*) cannot be used for plans related to other concerns of the application.

An *SMPlan* can include the same actions of any plan and actions for self-management to overcome the limitations of agent metamodels with regard to the AFs. Concretely, it includes the following actions (see Figure 3.8): *GetBehaviors*, *GetRoles*, *RemoveResource*, *AddResource*, *RemoveKnowledge* and *AddKnowledge*. The first two actions are defined to support self-awareness, while the last four actions are intended to support self-adjusting. The base metamodel has actions that support self-management too, such as *AssignRole*, *RemoveRole* or *CalledBehavior*. Additionally, due to the restrictions imposed by the APPEL syntax, the control structures are limited to loops, parallel actions and decisions. In Figure 3.9 we can see the *SMPlan* that corresponds to the *Task Allocation* policy (see Table 3.1).

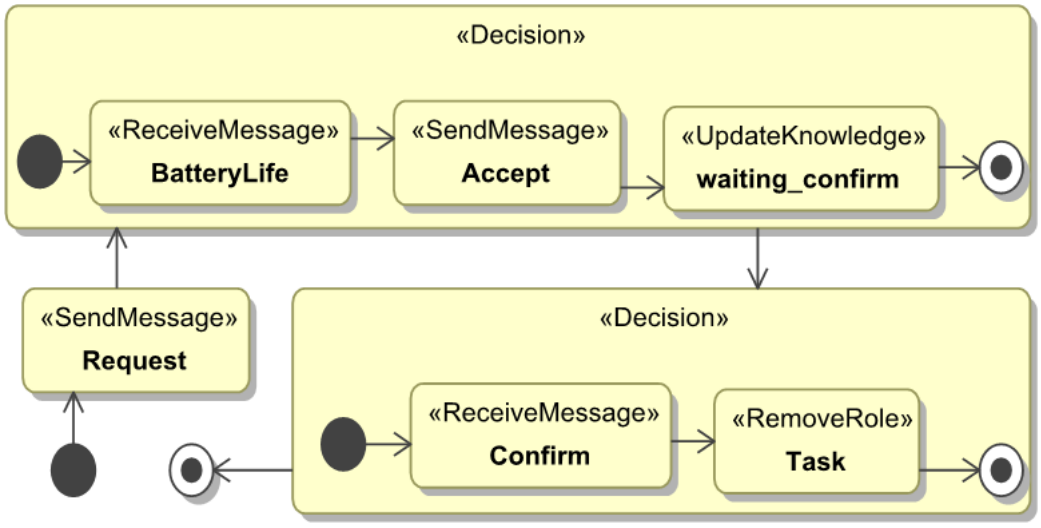

Figure 3.9: UML state machine corresponding to the *SMPlan* of the *Task Allocation* policy.

3.4 Summary

In this chapter we have presented Pineapple, a metamodel for designing AmI applications based on agents with self-management capabilities. Additionally, we have presented the modeling process of this metamodel using an IM for the case study. Agents are a natural metaphor for the modeling of distributed applications with autonomous and intelligent behavior, like AmI systems. These environments are characterized by a high degree of unpredictability and dynamism in the execution context, which makes the application of techniques like self-management necessary. Agent metamodels offer an excellent basis for modeling self-managed AmI systems, however they have the following limitations: (i) poor specification of dynamic behavior; (ii) lack of support to validate the self-managed behavior; and (iii) ambiguous notation to express AFs. In order to overcome these limitations, we have defined Pineapple, a metamodel which extends the PIM4Agents metamodel in different ways in order to overcome the aforementioned limitations. The principal contributions focus on self-management modeling, we have defined a new modeling viewpoint called *Selfmanagement*, which allows the roles for self-management (including the knowledge related with self-management) to be modeled together

with the policies that drive the self-managed behavior of the AmI system. Policies are described using a domain specific language that follows the APPEL syntax. The use of APPEL allows conflict between policies to be detected using the UMC model checker. Finally, the new viewpoint also includes specific actions for modeling AFs, facilitating the modeling of self-awareness and self-adjusting functions.

Chapter 4

From Pineapple to MalacaTiny

In this chapter we explain the M2M transformation process from the Pineapple metamodel to the MalacaTiny metamodel. This chapter starts by introducing the MalacaTiny metamodel (Section 4.1). We then explain the transformation process (Section 4.2) using the Intelligent Museum case study introduced in the previous chapter (Section 3.1).

4.1 The MalacaTiny metamodel

The MalacaTiny metamodel translates the advantages of the Malaca agent architecture to the metamodel level. The MalacaTiny metamodel is the PSM of the MalacaTiny implementations that will be presented in the Chapter 5. Most existing agent architectures focus on the type of agent (BDI, reactive, ...), but do not provide direct support for handling and reusing properties and functionality separately. This approach results in agent design and implementations being quite complex, brittle and difficult to understand, maintain, and reuse in practice. The main feature of the internal architecture of a Malaca [Amor and Fuentes, 2009] agent is that it represents, application-specific functions separately from extra-functional agent properties. This separation improves the internal modularization of the agent architecture, which is based on the composition of components and aspects, and contributes to enhancing the adaptation, reuse and maintenance of the software agent. These architectural features are maintained by the MalacaTiny

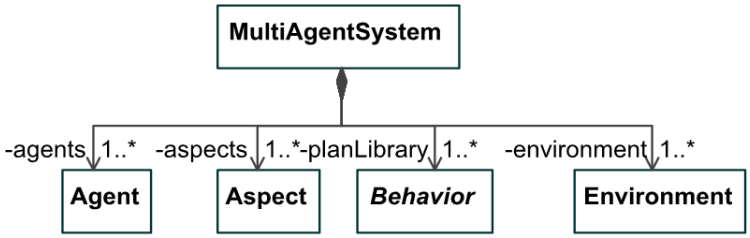

Figure 4.1: UML class diagram of the *MultiAgentSystem* concept in the MalacaTiny metamodel.

metamodel and even, upon implementation with the MalacaTiny agents. Within the MalacaTiny metamodel we can distinguish three main parts: (i) the modeling of the agent architecture; (ii) the modeling of the aspects ; and (iii) the modeling of the self-management. The root concept of the MalacaTiny metamodel is the *MultiAgentSystem* (see Figure 4.1). A *MultiAgentSystem* is composed of a set of *Agents*, *Aspects*, *Behaviors* and *Environments*. *Environments* are global to the MAS and contain data types that are handled by agents. The rest of the concepts are explained in the following sections.

4.1.1 Agent modeling

As stated, an agent in the MalacaTiny metamodel is composed of a set of components and aspects. This is a simplified vision of the MalacaTiny agent, in Figure 4.2 we can see the fragment of the MalacaTiny metamodel devoted to the *Agent* concept. The application specific functionality and knowledge are encapsulated in a set of *Component* and *Knowledge* elements, respectively. The access to environmental information (the environment can be computational or the external world) is encapsulated in *Facet* elements. This information about the external world for the applications is stored in *Context* elements. *Context* and *Facet* are mainly related with the self-management of the agent.

Additionally, the *Agent* concept includes the data needed to locate and compose aspects using the Malaca model [Amor and Fuentes, 2009]. In Malaca, aspect composition is ruled by composition rules (the *CompositionRule* element) and occurs in specific interception points (the *InterceptionPoint* element), the basic interception

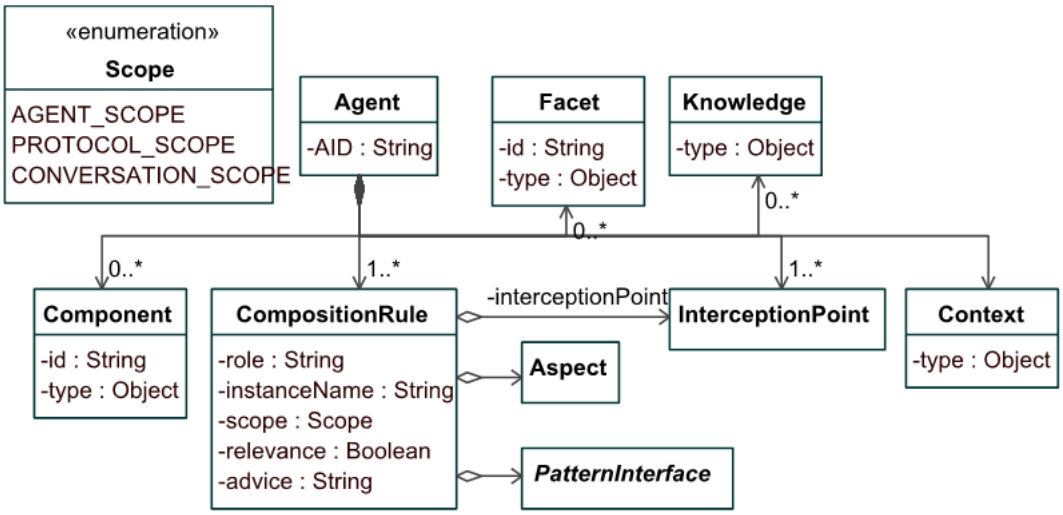

Figure 4.2: UML class diagram of the *Agent* concept in the MalacaTiny metamodel.

points of Malaca are the sending and receiving of a message, and when an event is thrown. In the MalacaTiny metamodel, interception points can be extended and it is possible to model the composition of aspects in other execution points. Malaca also considers the scope of an aspect (the *Scope* concept) that has three possible values. If the scope of an aspect is *AGENT_SCOPE*, then the aspect is unique in the agent architecture. In the case the scope is *PROTOCOL_SCOPE*, then the aspect encapsulates a coordination protocol that is unique in the agent architecture. Finally, if the scope is *CONVERSATION_SCOPE*, the aspect encapsulates a coordination protocol that is instantiated for each new conversation that the agent starts.

A *CompositionRule* includes the following: the *role* attribute specifies the function of the aspect in the agent architecture, e.g. an aspect that encapsulates the codification of the messages using a specific syntaxis has the role of *Representation*; the *instanceName* attribute has the identifier of the aspect in the agent architecture, this value is set according to the *scope* of the aspect; the *relevance* attribute sets whether the application of the aspect is relevant for the agent architecture, if it is set as true when the application of an aspect fails the composition process is stopped; and *advice* specifies the method that includes the behavior of the aspect

to be composed in the aspect composition. *CompositionRule* has a reference to the *InterceptionPoint* where the aspect is applied and to the *Aspect* applied in the aspect composition process. Finally, *CompositionRule* includes a reference to *PatternInterface*, that models the condition that must be met by the thrown event or the sent or received message in order to apply an aspect to the composition.

In Figure 4.3, a fragment of the modeling of the IM in the MalacaTiny metamodel is depicted . Specifically, it shows part of the modeling of the *SecurityAgent*. This agent has three components which are the user interface named *Component_GUI9*, an internal timer to coordinate some of the activities called *Component_Timer10* and a component to encrypt messages *Component_Encryptor11* (lines 2-4). Additionally, it has a set of composition rules to drive the aspect composition process (lines 5-22). As contextual information, this agent has the current position of the guard named *Location* (line 23) and the facet that gets this information (*Facet_LocationProvider11* in line 23). This agent has knowledge relative to the state of the museum (*Room1*, *Room2* and *Occupation*) (lines 24-28) and the basic interception points (*SND_MSG*, *RCV_MSG* and *THRW_EVNT*) (lines 29-31).

4.1.2 Aspect modeling

Aspects encapsulate crosscutting concerns of the agent architecture. In MalacaTiny, we have considered as crosscutting concerns of the agent, the *Representation* of the messages in a specific format, the *Monitoring* functionality of the agent, the *Distribution* of the messages using a specific technology, the *Coordination* using interaction protocols and the *SelfAdjusting* of the agent as part of the self-Management functionality inside the agent.

The part of the MalacaTiny metamodel devoted to aspects (see Figure 4.4) considers a super class named *Aspect*, which is extended by the other aspects. The *Aspect* element is composed of a set of *advice* names and the *class* that implements this aspect at implementation stage. The *Monitoring* aspect includes the *frequency* of the monitoring, the source of information (*Facet*) and the context of the agent where this information will be updated (*Context*). *Representation* and *Distribution* aspect do not add extra information to the super class. The *Coordination* aspect is

```
1: <agents name="SecurityAgent">
2:    <components name="Component_GUI9" type="/0/@environment.0/@dataTypes.0"/>
3:    <components name="Component_Timer10" type="/0/@environment.0/@dataTypes.8"/>
4:    <components name="Component_Encryptor11" type="/0/@environment.0/@dataTypes.9"/>
                                                ...
5:    <compositionRules advice="handleOutputMessage" aspect="/0/@aspects.9"
6:         instanceName="AuroraAdaptor" interceptionPoint="/0/@agents.0/@interceptionPoints.0"
7:         role="Representation" relevance="true"/>
8:    <compositionRules advice="handleOutputMessage" aspect="/0/@aspects.8"
9:         instanceName="SolAdaptor" interceptionPoint="/0/@agents.0/@interceptionPoints.0"
10:        role="Distribution" relevance="true"/>
                                                ...
11:   <compositionRules advice="handleInputMessage" aspect="/0/@aspects.9"
12:        instanceName="AuroraAdaptor" interceptionPoint="/0/@agents.0/@interceptionPoints.1"
13:        role="Representation" relevance="true"/>
14:   <compositionRules advice="handleInputMessage" aspect="/0/@aspects.2"
15:        instanceName="Sender" interceptionPoint="/0/@agents.0/@interceptionPoints.1"
16:        role="Coordination" relevance="true"/>
                                                ...
17:   <compositionRules advice="handleEvent" aspect="/0/@aspects.12"
18:        instanceName="EncryptionOptimizer" interceptionPoint="/0/@agents.0/@interceptionPoints.2"
19:        role="SelfManagement" relevance="true"/>
20:   <compositionRules advice="handleEvent" aspect="/0/@aspects.4"
21:        instanceName="Controller" interceptionPoint="/0/@agents.0/@interceptionPoints.2"
22:        role="Coordination" relevance="true"/>
                                                ...
23:   <facets type="/0/@environment.0/@dataTypes.2" id="Facet_LocationProvider11"/>
24:   <context name="Location" type="/0/@environment.0/@dataTypes.3"/>
25:   <context xsi:type="Arce.Agent:Knowledge" name="Room1" type="/0/@environment.0/@dataTypes.6"/>
26:   <context xsi:type="Arce.Agent:Knowledge" name="Room2" type="/0/@environment.0/@dataTypes.6"/>
27:   <context xsi:type="Arce.Agent:Knowledge" name="Occupation"
28:          type="/0/@environment.0/@dataTypes.9"/>
                                                ...
29:   <interceptionPoints name="SND_MSG"/>
30:   <interceptionPoints name="RCV_MSG"/>
31:   <interceptionPoints name="THRW_EVNT"/>
32: </agents>
```

Figure 4.3: Fragment of the modeling in XMI of the *SecurityAgent* in the MalacaTiny metamodel.

composed of a *FiniteStateMachine*, which is described below. Then, the following Subsection details the *SelfAdjusting* aspect.

FiniteStateMachine associated with *Coordination* describes the behavior of the agent in an interaction protocol. At runtime, this finite state machine is in an initial state and then, receives messages and internal events of its agent, that causes the machine to transit between its states. The transition occurs as follows: the machine receives a message or an event (i.e. an input), if this input follows a specific pattern, the machine transits to another state. Each time the machine transits, a plan is executed. In the MalacaTiny metamodel, a *FiniteStateMachine* has an *initial* state and is composed of a set of *State* and *Transition* elements, that describe how the machine transits between its states. A *Transition* is composed of *source* and *target* states, which represent the two states associated with the transition, the pattern that must be conformed by the input (the *PatternInterface* concept) and the *Plan* that is executed when the transition occurs. *PatternInterface* is an abstract class that is extended by *MessagePattern*, *InstancePattern* and *SituationalPattern*. *MessagePattern* allows a pattern to be established for the fields of an ACL message. To the contrary, *InstancePattern* is used to create patterns that are only followed by instances of a specific class. *SituationalPattern* models situations of interest in the agent context.

A *Plan* is an ordered set of *StandardAction* elements. A *StandardAction* is an abstract concept that is extended by *Task* and *StructuredAction*. *Task* represents simple atomic actions like the sending of a message (*SendMessage*) and user defined actions (*AtomicTask*). *StructuredAction* represents control structures like if-then-else (*IfThenElse*) or loops (*While*).

Figure 4.5 depicts a section of the modeling of the IM in the MalacaTiny metamodel. Specifically, it shows part of the modeling of the *RoutePlanningRequest* coordination aspect. *RoutePlanningRequester* models the set of actions required for a guide to generate a route in the Museum according to the current groups that are visiting the museum. This aspect has three advices (lines 2-4), three states (lines 15-17) and its transitions are driven by an event (line 6) and a message received (lines 11). Additionally, the *SendRouteRequest* plan is shown (lines 20-29), which is the initial plan that is executed to send a Message (lines 25-28) to other *GuideAgents* when an event is received (line 21-24).

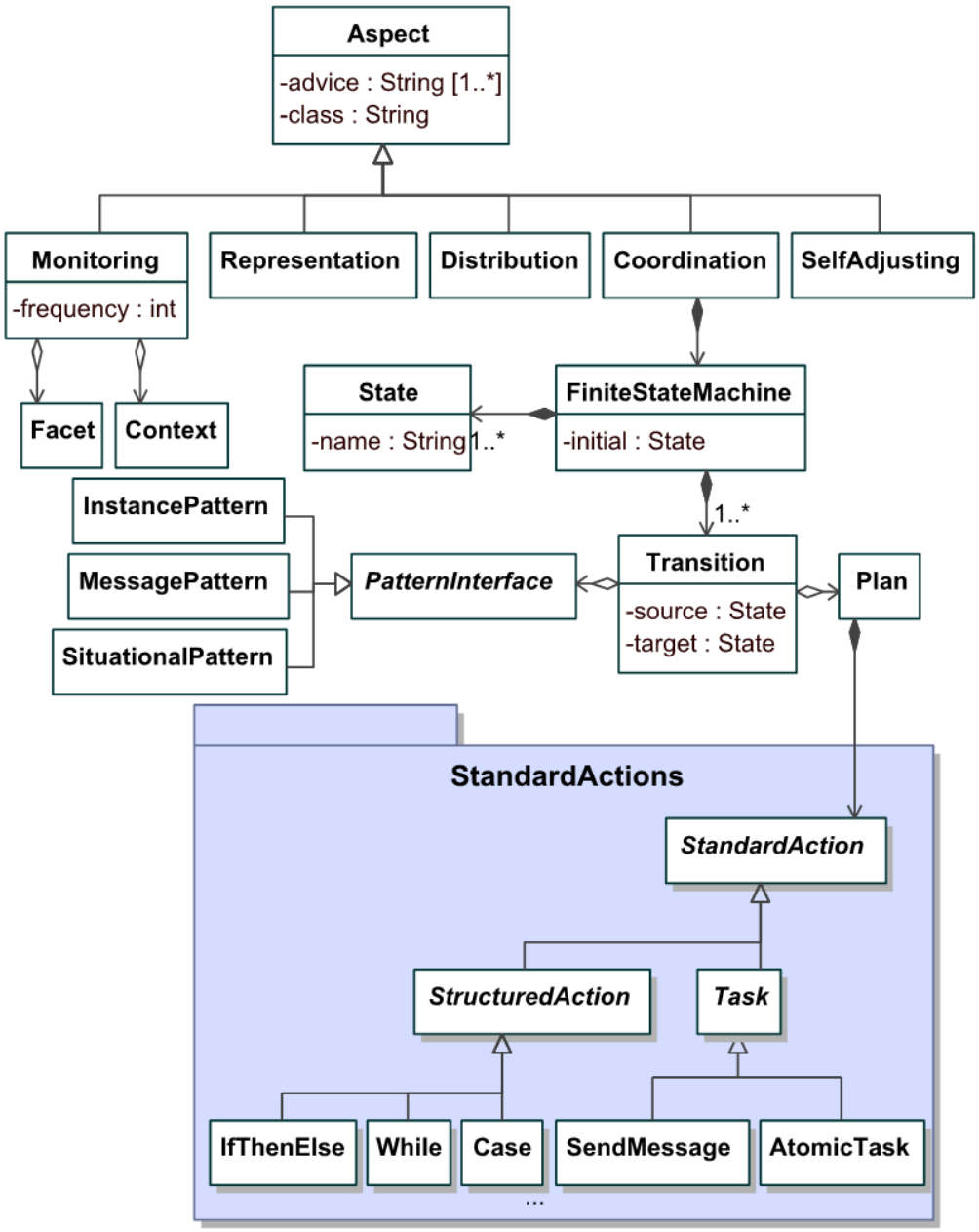

Figure 4.4: UML class diagram of the *Aspect* concept in the MalacaTiny metamodel.

```
1:   <aspects xsi:type="Arce.Aspect:Coordination" class="RoutePlanningRequester">
2:        <advices>handleInputMessage</advices>
3:        <advices>handleOutputMessage</advices>
4:        <advices>handleEvent</advices>
5:        <fsm initial="/0/@aspects.0/@fsm/@states.0">
6:          <transitions source="/0/@aspects.0/@fsm/@states.0" target="/0/@aspects.0/@fsm/@states.1"
7:           plan="/0/@planLibrary.1">
8:           <pattern xsi:type="Arce.Agent:InstancePattern" name="InitialEventRoutePlanning"
9:           instanceType="/0/@environment.0/@dataTypes.7"/>
10:         </transitions>
11:         <transitions source="/0/@aspects.0/@fsm/@states.1" target="/0/@aspects.0/@fsm/@states.2"
12:          plan="/0/@planLibrary.3">
13:           <pattern xsi:type="Arce.Agent:MessagePattern" name="messagePattern_10"/>
14:         </transitions>
15:         <states name="InitialRoutePlanning"/>
16:         <states name="ForkRequestInformation"/>
17:         <states name="ForkMyRoute"/>
18:       </fsm>
19:     </aspects>
                                                    . . .
20: <planLibrary xsi:type="Arce.Plan:Plan" name="SendRouteRequest">
21:      <steps xsi:type="Arce.Plan:GetInput" name="ReceiveEvent"/>
22:       <steps xsi:type="Arce.Plan:AtomicTask" name="ComposeRequest">
23:        <output name="source" type="/0/@environment.0/@dataTypes.6"/>
24:       </steps>
25:       <steps xsi:type="Arce.Plan:SendMessage" name="SendRequestInformation">
26:         <input name="source" type="/0/@environment.0/@dataTypes.6"/>
27:         <msg performative="inform"/>
28:       </steps>
29: </planLibrary>
```

Figure 4.5: Fragment of the modeling in XMI of the *RequestRoomCondition* coordination aspect and the *SendRoomConditionRequest* plan in the MalacaTiny metamodel.

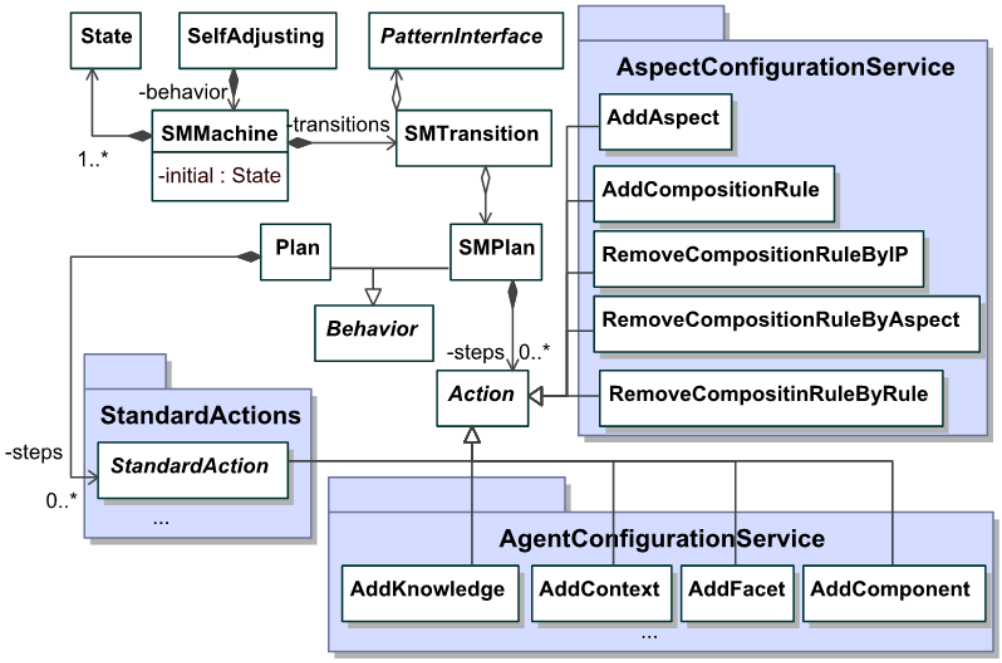

Figure 4.6: UML class diagram of the *SelfAdjusting* concept in the MalacaTiny metamodel.

4.1.3 Self-Management modeling

To perform self-management, the agent must be able to monitor the environment (self-situation), and itself (self-awareness); to detect changing circumstances (self-monitoring) and after that the agent must be able to adapt its behavior (self-adjusting), probably in coordination with other agents of the AmI environment. In the MalacaTiny metamodel self-situation and self-awareness are achieved means of the *Monitoring* aspect (see Figure 4.4) and the rest of the functions are encapsulated in the *SelfAdjusting* aspect and its associated plans called the *SMPlan* (see Figure 4.6).

The *SelfAdjusting* aspect is very similar to the *Coordination* aspect although their main difference is in the kind of plans that are executed in the transitions. This aspect is composed of a finite state machine (*SMMAchine* in Figure 4.6), which is composed of a set of *State* and *SMTransition* elements. Patterns associated to *SMTransition* represent situations that requires self-management actions (*SituationalPattern* or *InstancePattern*) or are related with the implementation of

self-management policies that requires the interaction between some agents (*MessagePattern*).

SMPlan, which is the element that models self-adjusting actions, is similar to *Plan* elements. An *SMPlan* is an ordered set of *Action* elements, while *Plan* is an ordered set of *StandardAction* elements (see Figure 4.6). *Action* is a super-type of *StandardAction* that is extended by actions to configure aspects (*AspectConfigurationService*) and to configure the agent (*AgentConfigurationService*).

Figure 4.7 depicts a fragment of the modeling of the IM in the MalacaTiny metamodel. Specifically, it shows part of the modeling of the *SensorSelfOptimizing* aspect. This aspect is the combination of the self-management policies that are depicted in the first two rows of Table 3.1. Because our metamodel does not accept parametric actions, we replicate policies for the different tasks that can be allocated or the frequency of which can be decreased. Therefore, the *SensorSelfOptimizing* aspect has 17 states (lines 23-38) and its transitions are ruled by *SituationalPattern* elements. In Figure 4.7, one of them is shown (lines 6-22). The *RequestLightMonitoringPlan* (lines 41-45) is an *SMPlan* that sends a message request to allocate light monitoring.

4.2 From Pineapple to MalacaTiny

In this section, we present the generation process from Pineapple (presented in Chapter 3) to MalacaTiny (presented in the previous section). This generation is one of the main contributions of this thesis and addresses some of the challenges raised in Subsection 1.1. A general overview of our approach is presented on the right-hand side of Figure 4.8. As source PIM and starting point of our MDD process we use Pineapple that meets the following requirements: (i) it is possible to represent concepts from different agent types (e.g. BDI, reactive agents), (ii) it is easy to specify MAS for different domains; (iii) it provides elements to model self-management policies and (iv) the context-aware behavior, required by AmI systems.

The design of a MAS using the Pineapple metamodel corresponds to the first step of our development process (see Figure 4.8), contributing to the achievement of the Challenge C1.1 (*Facilitate the high level modeling of AmI features*). Following

```
1:<aspects xsi:type="Arce.SelfManagement:SelfAdjusting
                        class="aspects.selfadjusting.SensorSelfOptimizing">
2:        <advices>handleInputMessage</advices>
3:        <advices>handleOutputMessage</advices>
4:        <advices>handleEvent</advices>
5:        <behavior initial="/0/@aspects.11/@behavior/@states.0">
6:        <transitions xsi:type="Arce.FiniteStateMachine:Transition"
7:           source="/0/@aspects.11/@behavior/@states.0"
8:           target="/0/@aspects.11/@behavior/@states.1" plan="/0/@planLibrary.6">
9:            <pattern xsi:type="Arce.Agent:SituationalPattern" name="SP_1">
10:             <situation xsi:type="Arce.Agent:ComposedCondition" operator="AND">
11:               <condition xsi:type="Arce.Agent:AtomicCondition">
12:                 <predicate knowledge="/16"/>
13:               </condition>
14:               <condition xsi:type="Arce.Agent:AtomicCondition">
15:                 <predicate knowledge="/18"/>
16:               </condition>
17:               <condition xsi:type="Arce.Agent:AtomicCondition">
18:                 <predicate knowledge="/17"/>
19:               </condition>
20:             </situation>
21:           </pattern>
22:       </transitions>
                                              . . .
23:       <states name="InitialSelfOptimizingPolicyState"/>
24:       <states name="DecreaseLightMonitoringFrequencyPolicyState"/>
25:       <states name="DecreaseNoiseMonitoringFrequencyPolicyState"/>
26:       <states name="DecreaseAccelerationMonitoringFrequencyPolicyState"/>
27:       <states name="DecreaseTemperatureMonitoringFrequencyPolicyState"/>
28:       <states name="RequestLightMonitoringState"/>
29:       <states name="AcceptLightAllocationProposalState"/>
30:       <states name="ConfirmLightAllocationProposalState"/>
31:       <states name="RequestNoiseMonitoringState"/>
32:       <states name="AcceptNoiseAllocationProposalState"/>
32:       <states name="ConfirmNoiseAllocationProposalState"/>
33:       <states name="RequestAccelerationMonitoringState"/>
34:       <states name="AcceptAccelerationAllocationProposalState"/>
35:       <states name="ConfirmAccelerationAllocationProposalState"/>
36:       <states name="RequestTempMonitoringState"/>
37:       <states name="AcceptTempAllocationProposalState"/>
38:       <states name="ConfirmTempAllocationProposalState"/>
39:     </behavior>
40:   </aspects>
                                              . . .
41:    <planLibrary xsi:type="Arce.SelfManagement:SMPlan" name="RequestLightMonitoringPlan">
42:      <steps xsi:type="Arce.Plan:SendMessage" name="SendLightAllocationRequest">
43:        <msg performative="inform"/>
44:      </steps>
45:    </planLibrary>
```

Figure 4.7: Fragment of the modeling in XMI of the *SelfAdjusting* aspect and the *RequestLightMonitoringPlan* in the MalacaTiny metamodel.

an MDD approach, and in order to define automatic transformations from source and target metamodels, we propose using MalacaTiny, as the target metamodel. Both metamodels are implemented in Ecore, and the transformation rules in ATL contribute to automatizing the MDE process. ATL is de facto standard which provides ways of producing a set of target models from a set of source models via a set of ATL transformations.

The second step of the MDD process is to generate MalacaTiny agents from the PSM. In this step the choice of MalacaTiny as the PSM is because of its platform-neutrality. This means, that MalacaTiny agents are able to use the communication infrastructure and services of any FIPA compliant agent platform or even use other communication technologies to interact with other agents. MalacaTiny is a metamodel and an efficient implementation of the Malaca architectural model that can be executed in multiple devices (Android, PDA, MIDP and Sun SPOT sensors) and deployed on top of various agent platforms, running different operating systems. The agent implementation generated by the M2T transformations from the MalacaTiny metamodel is independent of the target platform where the agent will be executed. The key is that MalacaTiny agents maintain the modularization of the MalacaTiny metamodel. The MalacaTiny agent metamodel separates platform-dependent functions (i.e the access to the agent platform services, such as the MTS, the Directory Facilitator (DF) or the Agent Management Service (AMS)) as aspects, following an aspect-oriented approach.

The entire agent functionality that depends on the target agent platform is encapsulated in an independent component (i.e. the distribution aspect), which can be incorporated inside the agent implementation as a plug-in in the deployment phase. So, we just need an M2M transformation from Pineapple to MalacaTiny (second step of our process), and a set of M2T transformations from the MalacaTiny metamodel to the different versions of MalacaTiny (third step of our process). Our process can automatically derive agents that can be deployed to use the communication infrastructure of different agent platforms using the appropriate plug-in. So, our MDD can be considered more extendable than others as adding a new agent platform only entails implementing a new plug-in thereby meeting challenge C1.2 (*Facilitate the extensibility of the MDD process*).

So, the benefit of using MalacaTiny as a PSM is threefold: (i) the incorpora-

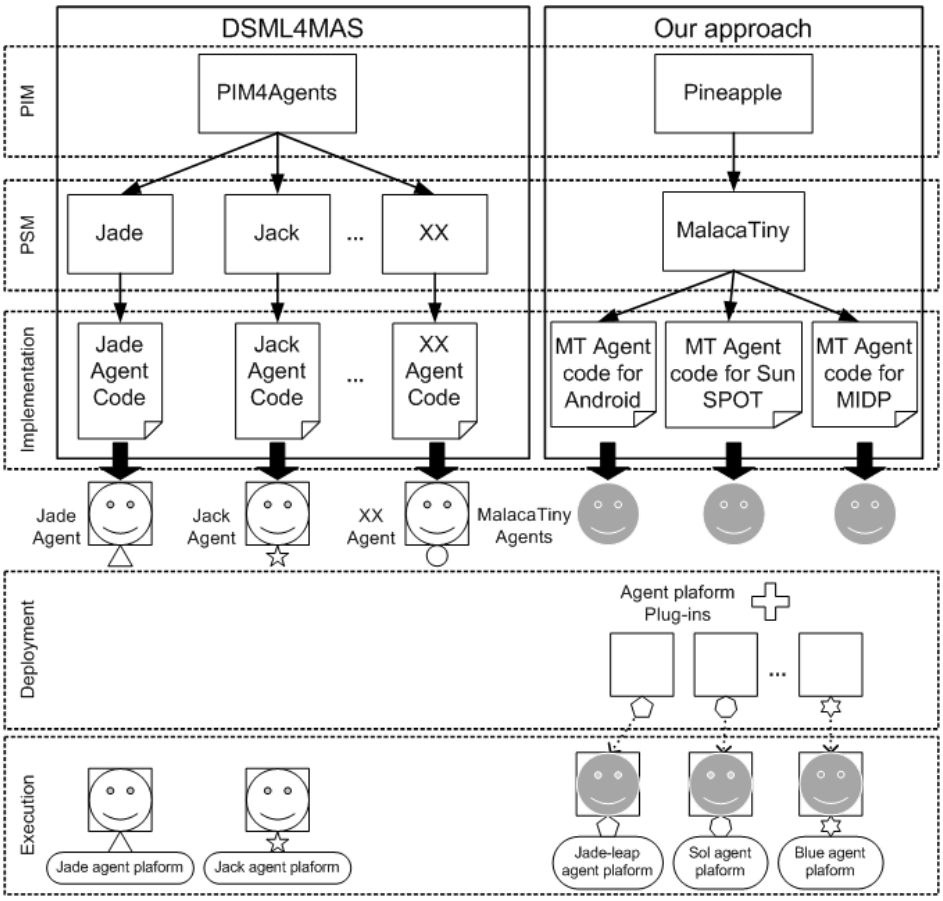

Figure 4.8: The overall picture: MDE straight forward approach (DSML4MAS) on the left hand side and from Pineapple to MalacaTiny on the right hand side.

tion of new agent platforms to this proposal has a lower cost than in an MDD straight forward approach like DSML4MAS (Figure 4.8 left hand side). Instead of requiring the specification of PSM metamodels and coding a new set of transformation rules, we only require the implementation of a new plug-in encapsulating the particularities of use of the new agent platform communication facilities (the Challenge C1.2, *Facilitate the extensibility of the MDD process*); (ii) the implementation of an MAS for different agent platforms does not require transforming and implementing it for each platform, instead, it just involves selecting and using the appropriate agent platform plug-in for each MalacaTiny agent (the Challenge C2.1, *Manage device and agent platform heterogeneity*). This plug-in receives the incoming messages and delivers outgoing messages to an agent platform, hiding platform-specific dependencies. This plug-in is modeled and composed as an aspect (i.e. the distribution aspect), following an aspect-oriented approach as we will show in subsequent chapters (the Challenge C2.2, *Cope with wireless network diversity*); and (iii) thanks to the separation of platform and communication dependent concerns, it is possible to easily extend Malaca/MalacaTiny to support new agent platforms and communication technologies (contributing to C2.2). For example, MalacaTiny agents running in smartphones can communicate using IEEE 802.11 (Wi-Fi), while the communication of MalacaTiny agents running in sensors relies on the IEEE 802.15.X (WPAN/Bluetooth/Zigbee) technologies.

In this section we summarize the main mappings between Pineapple concepts and MalacaTiny concepts, performed by a set of ATL mapping rules. The mapping rules included in this section do not constitute an exhaustive list. We have only included those that help the reader to understand the most relevant model mappings required for the use case scenario. Some mapping rules are applied automatically (simple ATL rules), while the application of other rules depends on the previous application of other mapping rules or must be invoked by other rules (ATL lazy rules) (see Subsection 2.4.4.2). For each concept from the Pineapple metamodel there are one or more transformation rules to map this concept to the MalacaTiny metamodel. This is not the case of the organization concept of PIM4Agents. MalacaTiny focuses on the internal design of the agent and organizations are not considered explicitly. Therefore, this concept is not mapped to MalacaTiny. In this section the transformation rules are described in detail and

Table 4.1 summarizes the main mappings between the concepts of Pineapple and the MalacaTiny metamodel section devoted to agent concepts, Table 4.2 does the same for aspect concepts and Table 4.3 for the self-management concepts. We have labeled each rule in the form of an AGnumber for agent rules, a ASnumber for aspect rules and SMnumber for self-management rules to make identifying a specific rule in the text easier.

We also illustrate the application of each rule in our example. The diagrams presented in Subsections 3.3.1 and 3.3.2 provide enough information to generate the MalacaTiny specifications for *GuideAgent*, *SecurityAgent*, *SensorAgent* and *VisitorAgent*. This section focuses on the code generation process of the *SecurityAgent* (Figure 4.3), the *RoutePlanning* protocol (Figure 4.5) and the *TaskAllocation* policy (Figure 4.7).

4.2.1 Generating agents

The work of the rules presented here is illustrated in Figure 4.3.

Agent Rule 1. This rule is the first to be applied in the M2M transformation process and the trigger for the rest of the rules presented in these sections. It takes the *MultiagentSystem* concept from the Pineapple metamodel and generates the basic structure of the *MultiAgentSystem* concept of MalacaTiny. Additionally, it generates the elements by default in the M2M process such as aspects for *Representation* and *Distribution*.

Agent Rule 2. This rule maps an *Agent* to the basic structure of an *Agent* in Pineapple and its constituent parts: *name*, *CompositionRule* and *InterceptionPoint*. The *CompositionRule* element is partially generated (lines 5-13), since only those aspects considered obligatory for the correct functioning of the agent are derived (i.e. the *Representation* and *Distribution* aspects). Both aspects are platform-dependent so only the default information is generated here. The interception points by default (*SND_MSG*, *RCV_MSG* and *THRW_EVNT*) are also generated by this rule (lines 29-31).

Agent Rule 3. This rule maps a *Resource* (an Internal component of the agent in the Pineapple metamodel) to a *Component* in MalacaTiny which is a set of iden-

Table 4.1: Mapping process between the Pineapple metamodel and agent concepts from MalacaTiny metamodel

Target	*Source*	*Explanation*
AG1: *MultiAgentSystem*	*MultiagentSystem*	Each *MultiagentSystem* is mapped to a *MultiAgentSystem*
AG2:*Agent*	*Agent*	Each *Agent* is mapped to the basic structure of an *Agent* with basic interception points and a set of aspects by default
AG3:*Component*	*Resource*	Each *Resource* is mapped to an agent *Component*
AG4:*Facet*	*Resource*	Each *Resource* that extends *Facet* is mapped to a *Facet* element
AG5:*Knowledge*	*Knowledge*	Each *Knowledge* is mapped to a *Knowledge*
AG6:*Context*	*Knowledge*	Each *Knowledge* that extends *Context* is mapped to a *Context*
AG7: *CompositionRule*	*Actor*	Each *Actor* is mapped to a *CompositionRule* of a *Coordination* aspect with the role of *Coordination* and affected by the *RCV_MSG* interception point.
AG8: *CompositionRule*	*Plan*	Each *Plan* which is not used within an *Interaction* is mapped to a *CompositionRule* of a *Coordination* aspect with the role of *ContextAwareness* and affected by the *THRW_EVNT* interception point.
AG9: *CompositionRule*	*SMRole*	Each *SMRole* is mapped to a *CompositionRule* of a *SelfAdjusting* aspect with the role of *SelfManagement* and affected by different interception points depending on its triggers.

tifiers that permit the identification of the component in the deployment phase in order to assign a specific implementation to it (lines 2-4). Specifically, it generates the component's name, its type, operations and internal attributes.

Agent Rule 4. This rule maps a *Resource* to a *Facet* in MalacaTiny (line 23). Due to the fact Pineapple does not have specific element to model *Facet* elements, we add to the *Environment* of the Pineapple model a data type named *Facet*, that is extended by data types that are considered *Facet* by the system modeler. As AG3, this rule generates a set of identifiers that permit the identification of the facet in the deployment phase in order to assign a specific implementation to it.

Agent Rule 5. This rule maps the *Knowledge* of an agent in Pineapple, to the *Knowledge* of an agent in MalacaTiny. Specifically, the identifier and the type of the knowledge (lines 24-28) are mapped.

Agent Rule 6. This rule generates *Context* from *Knowledge* elements (line 23). In MalacaTiny, *Context* is an element that represents knowledge about the external world (environment or device). As in the case of *Facet* elements, there are no specific elements in Pineapple to model *Context*, so the modeler must add an element named *Context* to the *Environment*, which is extended by the elements that are *Context* elements.

Agent Rule 7. This rule generates composition rules for coordination aspects. This kind of aspect can be easily identified in the source metamodel, because it must be defined in the interaction viewpoint. So, if a *DomainRole* is associated with a *Protocol* in Pineapple (by means of a *Collaboration* element), then two *CompositionRule* elements are derived for the *RCV_MSG* (lines 13-15) and *THRW_EVNT* (lines 20-22) interception points.

Agent Rule 8. This rule generates composition rules for the context-aware aspects of the agent. Each context-aware *Plan* is mapped to a *CompositionRule* with *Role* mapped to *ContextAwareness*, *RoleInstance* mapped to *Plan*'s name and *Scope* mapped to *Agent*. A Plan is context aware if it has an *AtomicTask* named *ReceiveEvent* with a *Knowledge* associated with it (Figure 3.5).

Agent Rule 9. This rule generates composition rules for self-adjusting aspects. This kind of aspect can be easily identified in the source metamodel, as it must be

defined within an *SMRole*. So, if an *SMRole* is associated with a *Policy* in Pineapple, then a *CompositionRule* element is derived. By default, a *CompositionRule* is generated, associated with the interception point *THRW_EVNT* (lines 17-19) and additionally, it is possible that other rules will be mapped depending on the triggers associated with the *Policy*. For example, if a trigger of a *Policy* is the sending of a message, then a composition rule associated with the *SelfAdjusting* aspect is generated, affected by the *InterceptionPoint SND_MSG*

4.2.2 Generating aspects

The work of the rules presented here will be illustrated in Figure 4.5, and corresponds to the actions of the *Requester* in the *RoutePlanningProtocol* (see Figure 3.4).

Aspect Rule 1. Each *Plan* which is not associated with a *Protocol* (i.e. *MessageScope* elements associated with their action are not set) and has an action *ReceiveEvent*, is mapped to a *Coordination* aspect the *FiniteStateMachine* of which has a single *State* and a single *Transition*. The *Behavior* of this *Aspect* is to execute actions depicted in *Plan* each time the event associated with the *ReceiveEvent AtomicTask* is thrown.

Aspect Rule 2. Each *Actor* is mapped to the basic structure (the name of the class that implements the aspect (line 1), advices (lines 2-4) and the structure of the *FiniteStateMachine* (line 5)) *Coordination* aspect. This is one of the most complex rules in our M2M process because its work is supported by multiple helpers and lazy rules (See Subsection 2.4.4.2). An *Actor* is composed of a set of *MessageFlows* and *MessageScopes*, that set how messages are exchanged between protocol participants (i.e. *Actors* in the Pineapple metamodel). *MessageFlows* are mapped as states of the *FiniteStateMachine* associated with the *Aspect* using AS3. Additionally, this rule calls the lazy rule AS4 which generates the transitions of the machine.

Aspect Rule 3. This rule maps a *MessageFlow* to a *State* of a *FiniteStateMachine* that is part of a *Coordination Aspect* (lines 15-17).

Table 4.2: Mapping process between the Pineapple metamodel and aspect concepts from MalacaTiny metamodel

Target	*Source*	*Explanation*
AS1:*Coordination*	*Plan*	Each *Plan* with an *AtomicTask* named *ReceiveEvent* is mapped to a *Coordination* aspect with the role of *ContextAwareness*
AS2:*Coordination*	*Actor*	Each *Actor* is mapped to the basic structure of the *Coordination* aspect
AS3:*State*	*MessageFlow*	Each *MessageFlow* associated with an *Interaction* is mapped to a *State*
AS4:*Transition*	*MessageFlow*	Each *MessageFlow* associated with an *Interaction* is linked to the next MessageFlow associated with the Actor and mapped to a *Transition* of *FiniteStateMachine*
AS5:*Plan*	*Plan*	Each *Plan* is mapped to a *Plan*
AS6:*AtomicTask*	*InternalTask*	Each *InternalTask* is mapped to an *AtomicTask* with the same name
AS7:*Send*	*Send*	Each Send Task is mapped to a Send
AS8:*GetInput*	*Receive*	Each *Receive* is mapped to a *GetInput* that accesses the input that causes the transition of the *FiniteStateMachine*
AS9:*GetInput*	*ReceiveEvent*	Each *AtomicTask* the name of which is *ReceiveEvent* is mapped to a *GetInput* that accesses the input that causes the transition of the *FiniteStateMachine*
AS10:*IfThenElse*	*Decision*	Each *Decision* is mapped to the *StructuredAction IfThenElse.*

Aspect Rule 4. This is the lazy rule that is called by AS2. If *MessageFlows* are mapped to states of the finite state machine inside the *Coordination* aspect, then two *MessageFlow* elements and a *Plan* determines the transition of the machine (lines 6-14). This rule takes a *MessageFlow* and determines its successor *MessageFlow* and the associated plan using helpers. The pattern that causes the transition is determined by the ACL message linked to the *MessageFlow*.

Aspect Rule 5. This rule maps a *Plan* in Pineapple to a *Plan* in MalacaTiny. The main difference between these elements is how to determine the order of the actions contained in the plan. In Pineapple it is determined by elements named *ControlFlow*, while in MalacaTiny actions are contained in an ordered list. This rule transforms the Pineapple *Plan* into an ordered set of actions (line 20). Finally, concrete actions are mapped using transformation rules such as AS5-6.

Aspect Rule 6. For each *Task* or *StructuredActivity* of the Pineapple metamodel there is a transformation rule that maps this element to the Pineapple metamodel. In this rule, an *AtomicTask* is mapped to an *InternalTask* with the same name and input and output parameter. The input and output parameter are mapped from elements named *LocalKnowledge* which can be associated with activities in the Pineapple metamodel. *LocalKnowledge* concepts are modeled within *KnowledgeFlow* elements, which set how information is shared between the activities of a plan. Using *KnowledgeFlow*, the rule can determine whether the parameter is an input or an output or both. The mapping of the parameter is common to the other rules devoted to map activities.

Aspect Rule 7. This rule is like AS6, but for the *Send* activity (lines 25-27).

Aspect Rule 8. This rule is like aspect rules 6 and 7 but transforms a *Receive Activity* into an *GetInput StandardAction*. This is one of the main differences between Pineapple and MalacaTiny. In MalacaTiny, messages or events are not explicitly received by plans, they are captured by aspects and cause the execution of plans. A *Plan* in MalacaTiny can access the input that causes its execution using the *GetInput StandardAction*.

Aspect Rule 9. This rule is like AS8, but for the *AtomicTask* the name of which is *ReceiveEvent* (lines 21-24).

Table 4.3: Mapping process between the Pineapple metamodel and self-managment concepts from MalacaTiny metamodel

Target	*Source*	*Explanation*
SM1: *SelfAdjusting*	*SMRole*	Each *SMRole* is mapped to a *SelfAdjusting* aspect
SM2: *SMMachine*	*BasicRule*	Each *BasicRule* is mapped to an *SMMachine* with a single state and a single transition
SM3: *SMMachine*	*ComposedRule*	Each *ComposedRule* is mapped to an *SMMachine* the states and transitions of which are determined by its sub-policies
SM4: *Transition*	*BasicRule*, *BasicRule*	Lazy rule that transforms two *BasicRule* elements from a *ComposedPolicy* in a *Transition* of the *SMMachine*
SM5:*State*	*BasicRule*	Each *BasicRule* that is part of a *ComposedRule* is mapped to a *State* of the *SMMachine* generated from such said *ComposedRule*
SM6: *InstancePattern*	*EventTrigger*	Each *EventTrigger* is mapped to an *InstancePattern*
SM7: *MessagePattern*	*MSGTrigger*	Each *MSGTrigger* is mapped to a *MessagePattern*
SM8: *SituationalPattern*	*Condition*	Each *Condition* is mapped to a *SituationalPattern*

Aspect Rule 10. This rule is similar to those aforementioned, but for a *Decision StructuredActivity*. There is a transformation rule for each type of *StructuredActivity* in the Pineapple metamodel. These rules are similar to AS6, but they are composed of other *Activity* elements.

4.2.3 Generating self-management

The work of the rules presented here is illustrated in Figure 4.7.

Self-Management Rule 1. This rule generates the basic structure of the *SelfAdjusting*, i.e. the name of the class that implements the aspect and its advices

(lines 1-4). Additionally, the structure of the *SMMachine* is generated and its initial state (line 5) are generated, the transitions and states of this element are generated by the following rules.

Self-Management Rule 2. This rule maps the behavior depicted in a *BasicRule* to an *SMMachine*. *BasicRule* policies are simple ECA policies, so they are mapped to a simple finite state machine (*SMMachine*) with a single *State* and *SMTransition*. *BasicRule* has optional trigger and condition, so it is possible to define a rule without condition and trigger and with the two elements simultaneously. If any element is defined, the transition of the machine is driven by a *SituationalPattern* that always returns true. If only a *Condition* is defined, the transition is driven by a *SituationalPattern* that represents the *Condition*. In the case, a *Trigger* (for messages or events) is modeled, then an *InstancePattern* or a *MessagePattern* is mapped. In the case that both elements are modeled, the *Trigger* is used to define the pattern of the transition and the condition to set a precondition to the execution of the *Plan* included in the *Transition*. The *SMPlan* of the *BasicRule* (see Figure 3.7) is mapped to the *SMPlan* of the *SMMachine* (see Figure 4.6).

Self-Management Rule 3. This rule maps the basic structure of a *SelfAdjusting* aspect that models the behavior of a *ComposedPolicy*. The *SensorSelfOptimizing* role has a *ComposedPolicy* with the *Sequential* operator that contains the policies *Decrease Frequency* and *Task Allocation* (rows 2 and 3 of Table 3.1). The states of the *SMMachine* inside the aspect are generated from the *BasicPolicy* elements that compose the *ComposedPolicy* (lines 23-38). *Transitions* of the *SMMachine* are determined by the *RuleOperation* (see Figure 3.7) attached to the *Policy* (lines 6-22). If the operator *Sequential* generates finite state machines the states of which are sequential. On the other hand, the other operators produce machines with states in parallel.

Self-Management Rule 4. This lazy rule, takes two *BasicPolicy* elements that are consecutive states according to *RuleOperation* elements and their *ComposedPolicy* and maps a transition of the *SMMachine* (lines 6-22). *Trigger*, *Condition* and *Plan* are taken of the second *BasicRule* and are mapped using the same procedure depicted in SM2.

Self-Management Rule 5. Each *BasicRule* which is part of a *ComposedRule* is mapped to a *State* of *SMMachine* (lines 23-38). The name of the *State* is generated automatically.

Self-Management Rule 6. This rule maps an *EventTrigger* to an *InstancePattern* that is part of an *SMTransition*.

Self-Management Rule 7. This rule maps an *MSGTrigger* to an *MessagePattern* that is part of an *SMTransition*.

Self-Management Rule 8. This rule maps a *Condition* to an *SituationalPattern* (lines 10-20) that can be part of an *SMTransition* or the pre-condition of an *SMPlan*.

4.3 Summary

In this chapter we have presented the MalacaTiny metamodel and the MDD process to generate MalacaTiny agents from a Pineapple model. The model driven solution proposed covers the design by the transformation of a design model of the AmI system in Pineapple, to a design in the MalacaTiny metamodel. MDD is the most natural approach to automate the derivation of agent implementations from high level agent models, considering different target agent platforms. The process presented in this chapter significantly simplifies this process by using MalacaTiny, a platform-neutral agent metamodel. This enhancement is particularly important for AmI environments, since new devices are continuously appearing and this trend is expected to continue. With MalacaTiny it is possible to configure agents to be executed in different target agent platforms for different mobile and lightweight devices, as required by most AmI environments. Compared with other MDD approaches for agents like the DSML4MAS approach, in our process including a new agent platform requires less effort and user skills.

Chapter 5

Code generation of MalacaTiny agents

In this chapter we explain the M2T transformation process of MalacaTiny agents. There is an M2T transformation process for each version of the MalacaTiny agent (see Figure 1.1). However, they are very similar to each other so we are going to focus on the generation of MalacaTiny agents for MIDP profile and the generation process for Goal Oriented MalacaTiny. Prior to the description of the code generation process, this chapter will focus on the implementation issues of these versions of MalacaTiny.

This chapter is structured as follows: Section 5.1 describes the internal design of MalacaTiny for MIDP devices and Goal Oriented MalacaTiny; Section 5.2 presents the code generation process; and Section 5.3 summarizes the contributions of this chapter.

5.1 The MalacaTiny agents implementation

In this section, we present the basis for meeting Challenge C2 (*Efficient embedding of agents in heterogeneous AmI devices*), the MalacaTiny agents.

MalacaTiny is a family of lightweight agent implementations based on the Malaca agent architecture [Amor and Fuentes, 2009] for typical AmI devices. This means that in our approach, the internal architecture of the agent is customized

considering the restrictions of the resources and capacities of each AmI device (e.g. the communication protocol used). The architectural design of Malaca and MalacaTiny agents is aspect oriented, separately representing application specific functionality concerns from the crosscutting agent properties. This separation improves the internal modularization of the agent architecture, which is based on the composition of components and aspects, and contributes to enhancing of reuse and especially the adaptation, reconfiguration and evolution of the software agent. Some of the *aspects* used by MalacaTiny are already present in Malaca (e.g. distribution and representation of messages and coordination through an interaction protocol) and others were defined to support the self-management property in MalacaTiny as we will show later. Currently the different versions of MalacaTiny can be executed in Android devices, devices with CLDC/MIDP profile (mobile phones that support Java ME) and Sun SPOT sensors. This feature of MalacaTiny meets Challenge C2.1 (*Manage device and agent platform heterogeneity*).

The use of aspect-orientation helps us to successfully deal with Challenge C2.2 (*Cope with wireless network diversity*). MalacaTiny agents are able to support different communication technologies (simultaneously). Interagent communication is supported by both the agent and the agent platform. While the agent is responsible for creating, formatting and sending a message, the agent platform provides the message transport between the agents, among other capabilities. Inside the agent, agent communication is mainly supported by the *Distribution* aspect, which resolves the delivery of agent messages through a communication mechanism and/or agent platform customized to the AmI device. Additionally, aspect-orientation contributes to meeting Challenge C3 (*Self-managed agents*).

In what follows, we show the internal design of MalacaTiny and Goal-Oriented MalacaTiny. Specifically, we present the internal design of MalacaTiny agents running in Sun SPOT. In the context of the MalacaTiny technologies, there are two devices that support the MIDP profile, mobile phones that are Java ME enabled and the Sun SPOT sensor motes. The different versions of MalacaTiny (not Goal Oriented MalacaTiny) have a lot of points in common, and only differ in implementation details. So, we consider the Sun SPOT version a representative example of the MalacaTiny agents and relevant for the case study of the Intelligent Museum.

5.1.1 The core agent classes

In this section, we show the details of the core agent class of the MalacaTiny and Goal-Oriented MalacaTiny.

5.1.1.1 The *Mediator* class of MalacaTiny

The MalacaTiny implementation presented in this section (Figure 5.1) is a reactive agent that must be efficient in resource consumption and also consider the particular limitations of Java ME compared to the standard Java. One of these limitations is the absence of the Reflection API, which, in the former Malaca, gave the necessary technical support to compose their internal elements (components and aspects) via a late binding mechanism at runtime. With this mechanism the developer only has to specify (not codify) the necessary information to instantiate and invoke the method of a specific component. But, this is not possible in Java ME, so we implemented a simpler mechanism in MalacaTiny: the components are added to the agent with an identifier (e.g. *addComponent("GUI", UserGuiClass)*) so that other elements of the architecture (normally the aspects) can refer to this identifier and require the services provided by a certain component, by first obtaining the reference of the required component using the *getComponent* method (e.g. *getComponent("GUI")*). This feature improves the modularization and evolution of application specific functionality, since aspects do not keep an explicit reference to the components they use, and only need to maintain a link to the *Mediator* entity, in charge of performing the composition between components and aspects, according to the composition rules.

To implement a MalacaTiny agent, we extend the *MalacaTiny* class that allows a core class of MalacaTiny, the *Mediator* (see Figure 5.1) to be configured. This class contains the elements that configure the architecture of the agent and performs the aspect composition process via the *AspecCompositionService* and the interception of the execution points where the aspects are composed (the sending and reception of messages via the *AgentCommunicationService* and the throwing of an event). If a component of the agent architecture is intended to throw events, that will be intercepted by *Mediator*, this component will extend the *EventCommunication* class. Additionally, *Mediator* has a reference to the agent *Knowledge-*

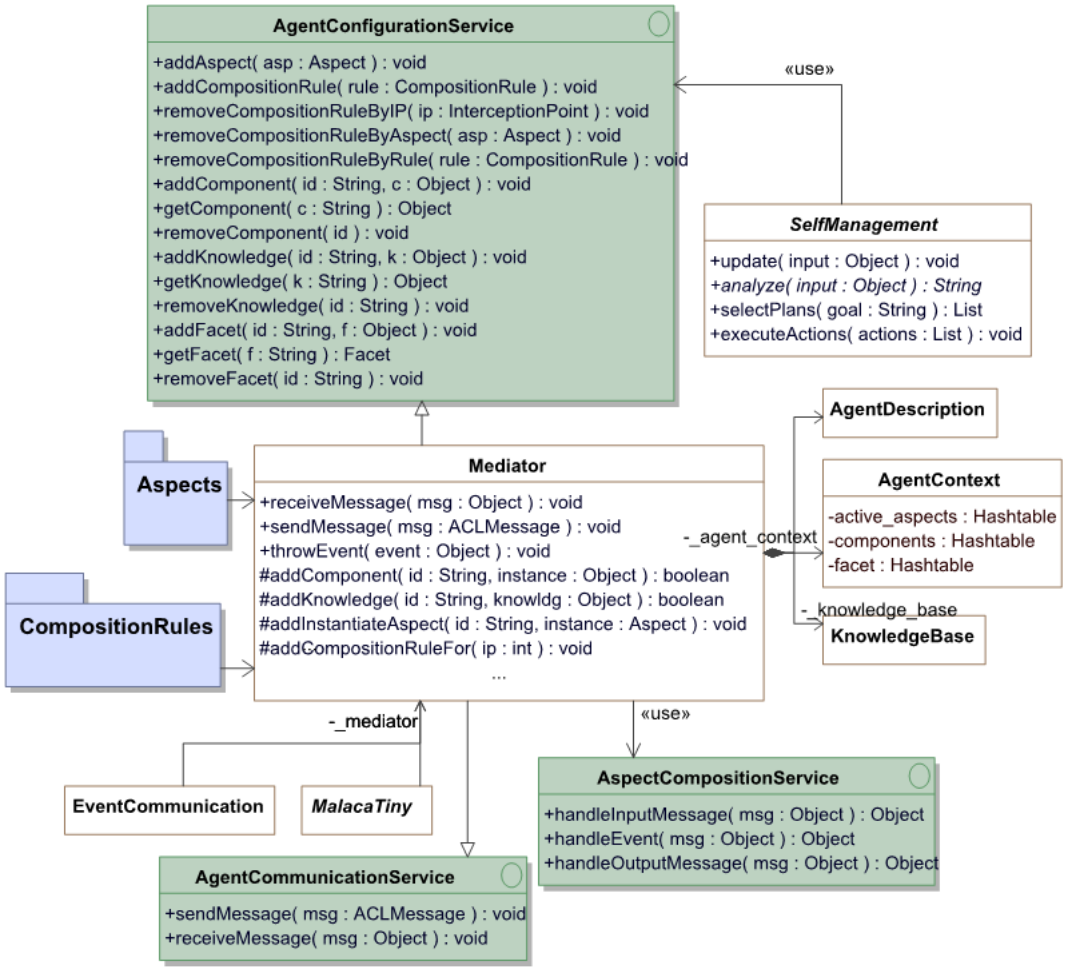

Figure 5.1: UML class diagram of the internal design of MalacaTiny for MIDP devices.

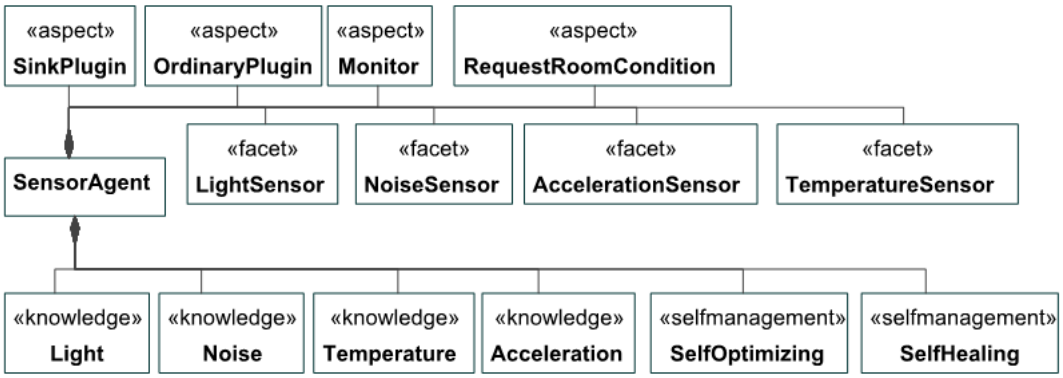

Figure 5.2: UML class diagram of the *SensorAgent*.

Base, *AgentContext* and *AgentDescription*. *AgentContext* contains the information of working aspects and components, while *AgentDescription* provides information about available components and aspects in the MalacaTiny agent. Finally, the *AgentConfigurationService* interface offers methods to accomplish adaptations in the agent architecture, that are particularly relevant for the self-management property.

In Figure 5.2, we present the design of the *SensorAgent*. As depicted in the figure, MalacaTiny also considers facets described in the MalacaTiny metamodel (see Subsection 4.1.1) for accessing the sensory functions (*LightSensor*, *NoiseSensor*,...). Additionally, the agent has the knowledge to contain such information sensed by facets (*Light*, *Noise*,...). The design of this agent also comprises a set of aspects that will be explained in following subsections.

5.1.1.2 The *Agent* class of Goal-Oriented MalacaTiny

As any other MalacaTiny agent, the Goal-Oriented MalacaTiny agent has been conceived as a software system running in a lightweight device, capable of interacting with its environment and with other agents. In contrast with other MalacaTiny agents, its behavior is goal-oriented and deliberative. Agent actions are organized in plans (as MalacaTiny agents), each plan associated with a goal to be achieved, to which it contributes. The achievement of goals depends on the current context and their internal state.

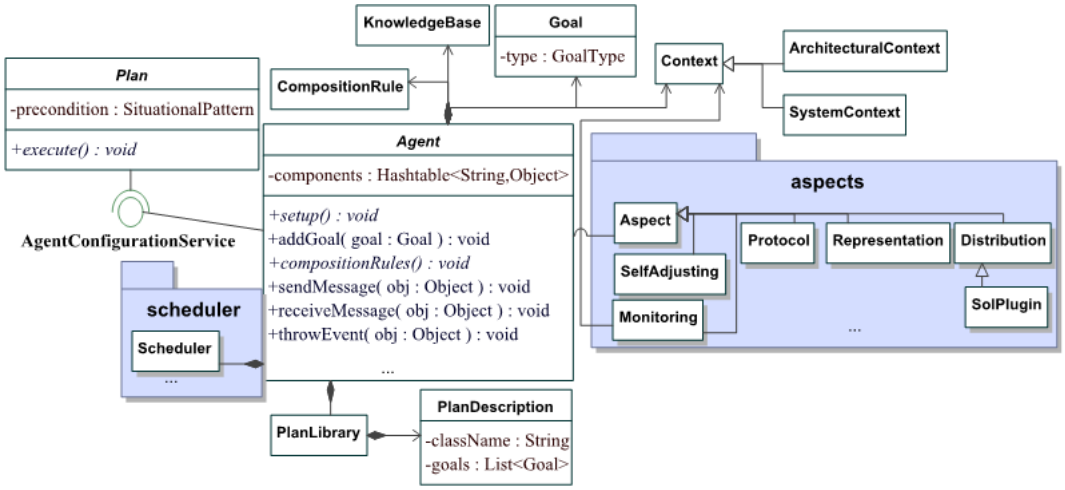

Figure 5.3: UML class diagram of the internal design of Goal Oriented MalacaTiny.

As depicted in Figure 5.3, the internal design of Goal-Oriented MalacaTiny agent shares classes with the MalacaTiny framework. The main difference is the absence of the *Mediator* class of MalacaTiny. In the case of the goal-oriented version, the functions of the *Mediator* are performed by the *Agent* class. This architecture borrows some concepts from BDI agents (it has goals, a knowledge base, and a plan library composed of plan descriptions).

Another distinguishing feature of our approach is that the agent maintains an explicit representation of its *architectural* context and the *system* context. The former captures the current state of the agent architecture by containing the set of components, aspects and relationships that are currently active (i.e. which are instantiated and in use). The latter context refers to the data and information derived from the system in which the agent is running (mainly the lightweight device), such as resource consumption.

Finally, the different elements involved in self-management, that compose the self-configuring loop in the *SecurityAgent* (see Figure 5.4), are explained in more detail in the following sections.

5.1.2 Aspects and aspect weaving

In this section, we explain the different aspectual properties considered by the MalacaTiny and Goal-Oriented MalacaTiny internal architectures and their aspect

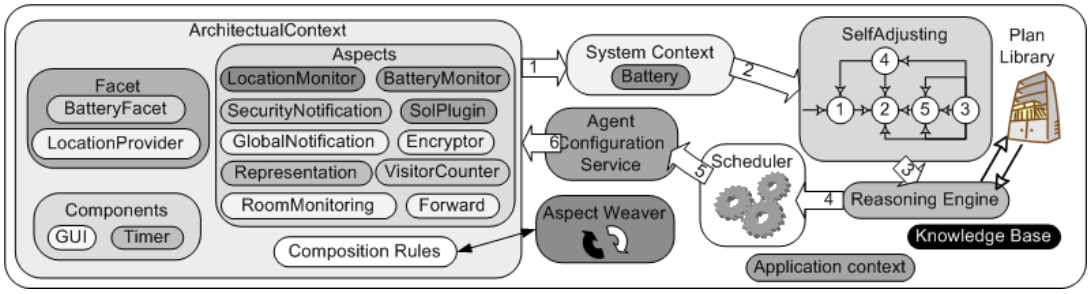

Figure 5.4: Self-management loop for *SecurityAgent*.

weaving processes.

5.1.2.1 Aspects weaving of MalacaTiny

As stated, the use of aspects helps achieve solutions for Challenges C2.1 (*Manage device and agent platform heterogeneity*) and C2.2 (*Cope with wireless network diversity*). Specifically, the issues related to the heterogeneity of communications are addressed by means of three different aspects inside the agent: (1) the *Distribution* aspect, which deals with message delivery; (2) the *Coordination* aspect, which deals with the exchange of messages according to a common interaction protocol; and (3) the *Representation* aspect, which formats ACL messages in a common representation format (such as String or Bit Efficient).

The *Distribution* aspect encapsulates how to use and access the MTS for message delivery. The implementation of the distribution aspect greatly depends on the services offered by the agent platform and the transport protocol used. In fact, this aspect supports platform-neutrality for FIPA compliant agent platforms: this aspect hides platform dependencies, and makes the rest of the classes of the agent architecture (components and aspects) independent from the agent platform and the communication service used at runtime. Indeed, the distribution aspect is an interface for the FIPA standard services provided by a remote agent platform.

The implementation of an aspect for the distribution of messages in a specific agent platform has to extend the *DistributionAspect* class and has to implement the methods defined in the *FIPAAgentPlatform* interface (Figure 5.5). The definition of this interface enables the use of different implementations of the distribution aspect, transparent to the rest of the elements of the architecture, and provides a

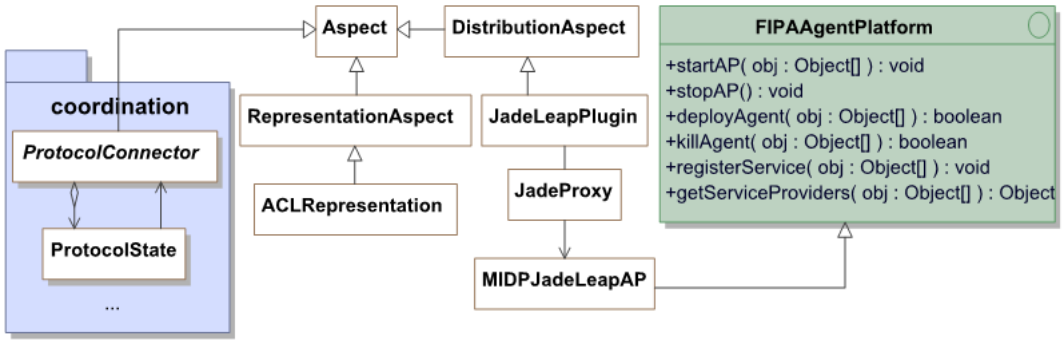

Figure 5.5: UML class diagram for aspects in MalacaTiny for MIDP devices.

unified interface to FIPA standard services (i.e. communication between agents, deploy or kill agents, start or stop the agent platform, register services and locate agents and services). The current implementation of MalacaTiny provides distribution aspects for Jade-Leap, *Sol* and Bluetooth. In Figure 5.5, the classes shown, implement the distribution aspect for the Jade-Leap agent platform: *JadeLeapPlugin* class extending *DistributionAspect* and *MIDPJadeLeapAP* class that implements *FIPAAgentPlatform* interface; both have a reference to *JadeProxy* class that acts as a proxy between Jade-Leap agent platform and these objects. Also, it is possible that an agent communicates with other agents through different agent platforms (part of Challenge C2.1 - *Manage device and agent platform heterogeneity*). There are other implementations of the *DistributionAspect* of MalacaTiny that will be presented in Chapter 6.

Another relevant concern separated as an aspect in MalacaTiny agents is coordination. The *Coordination* aspect is implemented by classes included in the *coordination* package. MalacaTiny agents have a *Coordination* aspect for each interaction the agent takes part in (supported by an interaction protocol). The design of this aspect is similar to the coordination aspect metamodel presented in Subsection 4.1.2. The behavior of the agent in the interaction (*ProtocolConnector*) is described and implemented using a finite state machine. *ProtocolConnector* contains: a declaration of the machine states (*ProtocolState*), the set of rules that describes state transitions, and the set of actions and activities (organized in plans) that the agent performs as a consequence of the transition.

The aspect composition process of MalacaTiny is driven by a set of composition

rules (see Figure 5.1), which set when and how aspects are woven. According to AOSD practices, aspects are composed when the system reaches certain execution points (join points). In AOSD, join points represent the execution points that can be intercepted (e.g. a message is received) and the predicate describing the set of join points intercepted by an aspect is called a pointcut (e.g. interception of each "cfp" message received). The join point model of MalacaTiny defines three join points: for the interception of the incoming (*RECV_MSG*) and outgoing (*SEND_MSG*) messages, and internal events (*THROW_EVENT*).

The composition rules of the MalacaTiny implementation are similar to those presented in the MalacaTiny metamodel (see Figure 4.2). Therefore, they include the following attributes: *pattern*, that models the predicate that must be met by the thrown event or the sent or received message in order to apply an aspect in the composition; *role*, that specifies the function of the aspect in the agent architecture, e.g. *Representation*; *instanceName* that contains the identifier of the aspect in the agent architecture, this value is set according to the *scope* of the aspect (see Subsection 4.1.1); the *relevance* attribute, which sets whether the application of the aspect is relevant for the agent architecture, if it is set as true, or if the application of an aspect fails, the composition process is stopped. Additionally, the rule sets the interception where the rule is applied (e.g. when a message is received) and the necessary information to instantiate the aspect, if this is required.

Composition rules are stored in the *Mediator* as an ordered list which is indexed by the interception point where the rule is applied. Each time a message is sent or received, or an event is thrown, the *Mediator* sequentially checks composition rules corresponding to this interception point (e.g. *RECV_MSG*). So, if the condition depicted in *pattern* is met by the message or the object attached to the event, then the aspect is applied to this element using the corresponding advice. Advices are methods that contain the aspectual behavior. MalacaTiny aspects have an advice by default for each interception point, which is included in the *AspectCompositionService* interface (see Figure 5.1). Therefore, when a message is received the advice *handleInputMessage* is applied in the aspect composition process, when a message is sent the advice applied is *handleOutputMessage* and when an event is thrown the advice used is *handleEvent*. When an aspectual rule is applied, the output produced by the process is used by the *Mediator* for the next composition

rule. For example, *VisitorAgent* has a set of composition rules for the interception point *RECV_MSG*. When this agent receives a message, a *Representation* aspect is applied, which transforms the message from the native format of the underlying agent platform to the format supported by the agent. Then, this transformed message is used with the next composition rule, which corresponds to a *Coordination* aspect. This aspect does not transform the message, which is used by the next rule in the ordered list of the composition rules.

In Figure 5.2, we can see the design of the *SensorAgent*, which comprises a set of aspects: *SinkPlugin* and *OrdinaryPlugin* carry on with the distribution of information in the WSN; *Monitor* is the aspect that captures data from facets and updates the knowledge with it; and *RequestRoomCondition* is the coordination aspect that answers data requests from *SecurityAgents*. *SensorSelfOptimizing* and *SensorSelfHealing* will be explained in the later paragraphs.

5.1.2.2 Dynamic weaving of Goal-Oriented MalacaTiny

Aspects of Goal-Oriented MalacaTiny has the same function as MalacaTiny agents. Therefore, they are mainly used to model and explicitly implement data and functions to facilitate communication and self-configuring activities.

Aspects of Goal-Oriented MalacaTiny has the same design as MalacaTiny aspects. Only the *Protocol* aspect (i.e. the *Coordination* aspect) differs a little because the transitions of its internal finite state machine does not describe actions that must be executed. On the contrary, transitions refer to goals that must be pursued. Figure 5.3 depicts an example of the *Distribution* aspect, the *SolPlugin*, which will be described in subsequent chapters.

The aspect composition process of Goal-Oriented MalacaTiny is similar to the process presented for MalacaTiny, the main differences between them are caused by the extensible join point model of the first. Goal-Oriented MalacaTiny has the same interception points as MalacaTiny and in addition, the developer can include new join points by extending the agent class with new methods to be intercepted and defining a *tag* that identifies the new interception point. Because of this extensibility, we cannot consider a fixed set of advices as for MalacaTiny and therefore, advices must be explicitly described in the composition rules that

drive the aspect weaving. In Goal-Oriented MalacaTiny, a composition rule is composed of the *tag* that identifies the interception point and a set of aspect descriptions. An aspect description includes the remainder information presented in MalacaTiny composition rules (*pattern*, *role*, *relevance*,...), but it also includes the advice that encapsulates the aspectual behavior.

The weaving process of Goal-Oriented MalacaTiny is depicted in Algorithm 1. In line 1, aspect descriptions that correspond to the specific interception point (identified by input *tag*) are selected and stored in the variable *A*. As explained, aspect descriptions contain the necessary data to weave an aspect at a specific interception point, including the restriction for the weaving of the aspect (an implementation of *PatternInterface*), the role that fulfills the aspect in the architecture or the corresponding advice to invoke. In lines 4 to 14, the weaving of the aspects contained in *A* is performed. As in the MalacaTiny weaving process, when the composition rule is selected, the first step is to check whether the input holds the predicate included in the *AspectDescription* (line 6). In this case, we need to know the role that the aspect plays in the agent architecture. If the aspect encapsulates the coordination of the agent using a coordination protocol (*Role.COORDINATION* in line 7), then the input is handled as a message otherwise it is handled as an event. This issue is important for getting the aspect for composition. At runtime aspects can be active or not, so in the methods in lines 8 and 10, aspects are selected if they are active or instantiated if not. An aspect is instantiated differently for a message then for an event. In the case of messages we can have multiple aspects of the same coordination protocol active, that correspond with different conversations of the agent. So we have to select/instantiate the aspect for the specific conversation. In the other case, there is a single instance of the corresponding aspect described in *AspectDescription*. When the aspect is selected, the advice is executed (line 12) and the output is assigned (line 15).

5.1.3 Implementation of the Self-management properties in MalacaTiny

We endowed MalacaTiny agents with self-management capabilities typical of autonomic systems. These functions not only enable the reconfiguration of the AmI

Algorithm 1 Aspect Composition

Input: the identifier the interception point *tag*, the aspect composition input *input*

Output: the output of the aspect composition process *output*

$A \leftarrow \{\forall AspectDescription \in CompositionRules | InterceptionPoint(AspectDescription) == tag\}$
$i \leftarrow 0$;
Aspect aspect;
while $i \leq size(A)$ **do**
  $Restriction\ rest \leftarrow A_i.getRestriction()$;
  if $rest.holdRestriction(input)$ **then**
    if $A_i.getRole() == Role.COORDINATION$ **then**
      $aspect \leftarrow getRequestAspectForMessage(A_i, input)$;
    else
      $aspect \leftarrow getRequestAspectForEvent(A_i, input)$;
    end if
    $input \leftarrow executeMethod(aspect, A_i.getAdvice(), input)$;
  end if
  $i \leftarrow i + 1$;
end while
$output \leftarrow input$;

system but also the agent's internal architecture. As defined in Chapter 2, self-management or autonomic computing [Kephart and Chess, 2003] is a term used to describe systems that assume their own management or are self-managed. From this general definition, we can consider a self-managed agent as an agent that is able to independently, and according to a set of rules or policies, take care of its own maintenance, configuration and optimization tasks, thus reducing the workload of the MAS's administrators. In order to develop a self-managed agent-based system, there are four so-called AFs that each agent should support [Dobson et al., 2010]: self-awareness, self-situation; self-monitoring and self-adjusting. Self-awareness is the capacity for introspection (be aware of their internal state), while self-situation is related with the awareness of current external operating conditions, which are commonly represented as the external environmental context; self-monitoring refers to the ability to detect changing circumstances in the agent environment. Finally, self-adjusting is the ability to accordingly adapt to these environment changes.

In MalacaTiny agents, the self-management is adapted according to the capacities of the device in which the agent is deployed. MalacaTiny agents have a simple self-management loop and policies are hard-coded, the function of the aspects is to improve the modularization and consequently facilitate the reconfiguration of the agent. The Goal Oriented MalacaTiny has a goal-oriented self-management control loop and also exploits aspect orientation to implement the AFs.

So, modeling the AFs by aspects allows us to: (1) add or remove the possibility of self-management from the agent; (2) optimize the resource usage, by adding only those aspects of self-management required by the MAS (e.g. monitor the battery and not the memory consumption); (3) improve the reasoning about each concern (e.g. awareness, security, etc.), since they are modeled separately; (4) reasoning on a limited number of aspects combined with the runtime weaving of aspects (i.e. system-level aspects) improves the scalability of the reconfiguration mechanism; (5) explicitly model the context awareness concerns as aspects, able to be used differently by each self-* (e.g. monitoring the network latency may be used to optimize the response time (self-optimizing), to fix a network communication problem (self-healing) and to change the communication protocol to adapt to the quality required by the user (self-configuring)). Any other functionality, non

crosscutting and which implements the application dependant behavior is modeled as a component or facet.

Additionally, since self-management may entail modifying the internal architecture of the agent, by, for example, adding new components or substituting others, the aspect-orientation presented in MalacaTiny and Goal-Oriented MalacaTiny facilitate the self-management process providing a loosely coupled architecture.

In the following subsections, we present how the self-management is integrated inside MalacaTiny and Goal-Oriented MalacaTiny agents.

5.1.3.1 Implementation of the SelfManagement class of MalacaTiny

The design of the MalacaTiny agent has a set of classes and interfaces to integrate self-management inside the agent. Self-management is performed by extending the abstract class *SelfManagement* (see Figure 5.1), that encapsulates the typical functions of a self-managing control loop (see Subsection 2.3). When the state of the agent requires the application of a self-management plan, this component analyzes the state, and according to a set of control policies, determines the plan (*SMAction* elements) to manage the situation, and executes the corresponding actions.

Plans are identified using tags that represent the purpose of the plan and are stored in an internal library of the *SelfManagement* class. A control policy defines the set of conditions that determine the execution of an action. These policies are encoded in the abstract method *analyze* (see Figure 5.1) of *SelfManagement*.

In the Intelligent Museum, the *SensorAgent* (see Figure 5.2) extends the *SelfManagement* class in *SensorSelfOptimizing* and *SensorSelfHealing*, that encapsulates policies depicted in Table 3.1.

5.1.3.2 Implementation of Self-management functions in Goal-Oriented MalacaTiny

In the subsections below, we explain the different elements involved in the self-management of Goal-Oriented MalacaTiny agents, stressing its relationship with the AFs (i.e. self-awareness, self-situation, self-monitoring and self-adjusting). As depicted in Figure 5.4, at runtime, there are two control loops, one is application-

specific and the other is for the self-management. The two loops generate goals that are processed at runtime by the Reasoning Engine. The BDI loop generates *ApplicationLevelGoals*, while the self-management loop generates *SystemLevelGoals*.

Implementing self-awareness and self-situation

Typically, in BDI architectures, there is a knowledge base that contains relevant information for the agent, i.e. the *belief* component. In order to carry out the self-management we need the internal of the agent (self-awareness) and that of the external world (self-situation). In our agent architecture, we consider that the internal information of the agent is its explicit architectural description, i.e. currently running components, aspects, composition rules and their current configuration, and the external world is the other information that may be considered as context, this information can be about the physical world (e.g. location) or about computational resources (e.g. memory occupation). The self-awareness is achieved using an explicit representation of the agent architecture, while the self-situation is achieved by modeling the external world as a set of context elements.

Self-awareness is achieved inside the agent architecture by the *ArchitecturalContext* element, which contains the set of aspects and components which compose the agent. For instance, the *SecurityAgent* (see Figure 5.4) needs components and aspects to send notifications to the people in the museum and control its rooms (interact with *GuideAgents*, *SensorAgents* and *VisitorAgents*). Additionally, it has support for a routing protocol encapsulated in the *Forward* aspect. *GuideAgent* has aspects to receive and send the messages (*SolPlugin*), an aspect to codify the messages in a readable format for the agents (*Representation*), an aspect to encrypt messages exchanged with other agents (*Encryptor*) and a set of aspects to rule the coordination between the different agents (*GlobalNotification*, *RoomMonitoring* and *SecurityNotification*). Moreover, the agent has a component to interact with the guard (*GUI*) and a *Timer* to coordinate activities internal to the agent. *ArchitecturalContext* also contains a set of composition rules (*CompositionRules*) to support the aspect composition process.

Self-situation is achieved inside the agent architecture by the *SystemContext* element (see Figure 5.4). The elements of *SystemContext* have a time stamp that allows the quality of the context or detection failures to be checked. The

information of the system context is gathered in the following way. In the case of the *SecurityAgent* that has to communicate with every agent in the system and has to ensure the device's correct functioning, the power consumption is the main concern for self-management. So, its *SystemContext* includes battery consumption.

The information which is specific to the application is stored in a *KnowledgeBase* as in most BDI architectures. For instance, *SecurityAgent* has a *LocationMonitor*, which is an aspect specific to the application and the information gathered by it is stored in the *KnowledgeBase*.

Implementation of the monitoring function

The monitoring function gathers information from the external world and the internal agent architecture and updates the *SystemContext* with it (Arrow 1 in Figure 5.4). The interface with the real world is modeled as a set of *Facet* elements, that encapsulates the sensing of a specific resource. There are facets that are passive and need to be explicitly sensed and others that transmit their values by means of events. Facets decouple the sensed resource from the sensing function thereby allowing multiple monitoring aspects specialized for a specific context. For example, in the case of the *VisitorAgent* which is aware of its battery level and consumption, it has a single facet element for the battery and two monitoring aspects associated with it, one for the current battery level and another for the battery life.

For an agent that is running in a lightweight device there are a number of issues that can be monitored: (i) a resource of the system, this can be computational resources, physical world or another agents in the MAS; (ii) the response time of a function, this comes from the time required to execute an operation; and (iii) the quality of context information, i.e. how old the information or accuracy is, this is done with time stamping. The usage of aspects for monitoring provides flexibility to our agent architecture as will be shown later.

The behavior of a *Monitor* aspect consists of periodically collecting data from a source of information. These aspects, are generic and are configured at runtime to monitor a given *Facet* element. So, each monitor aspect follows certain settings for measurement, which determines, for instance, the frequency of getting a new sample. These aspects can be configured modifying the sampling time in order to save resources or to increase the degree of accuracy of the context information.

The monitoring based on the periodic collecting of data is not enough as the source of information based on events. To collect this kind of information, our agent uses *ContextAware* aspects, that encapsulate the context dependant behavior of the agent. These aspects are usually affected by the interception point *THRW_EVNT* and the advice *handleEvent* described in Section 5.1.2.2. Using this kind of behavior, we can monitor components that warn of eventual failures in a component of system.

The monitoring of the response time of a system requires the modification of the source code introducing undesirable delays when this information is unnecessary. Aspect Orientation enables this kind of monitoring and provides a mechanism that disables the monitoring to avoid delays, even at runtime. As stated in Section 5.1.2.2, the weaving mechanism of our agent is extensible and allows new interception points to be added to the agent. An example of this is the case of the *VisitorAgent* that downloads information about the objects of an exhibition. The downloading process can be measured if we extend the agent with a new interception point and method (*GET_CONT* and *getContent* respectively). The visitor uses his/her mobile phone to scan a QR code attached to an object, this generates the goal *Get content for QR* that causes the execution of the plan depicted in Figure 5.6. This plan uses as input, the QR code and retrieves the content using the method *getContent* of the *VisitorAgent*. The calling of this function starts the aspect weaving process, that follows the composition rule depicted in the center of Figure 5.6. In order to measure the time needed to get the content, the aspect *MeasureMethodTime* is applied around (i.e. before and after) the aspect *GetContent*. It is the latter which accesses the database of contents and gets the data. After the aspect composition process, the plan has the content needed to update the user interface and the *SystemContext* is updated with the execution time of this method. Note that if we change the composition rule for this interception point (*GET_CONT*) removing the *MeasureTimeMethod* aspect, we disable the time measurement function and avoid the delay introduced by it.

Implementation of the self-adjusting aspect

The self-adjusting analyzes the *SystemContext* and applies the policies to solve the problems of the system (Arrow 2 in Figure 5.4). As stated, there are different kinds of policies [Kephart and Walsh, 2004], based on actions, goals or utility

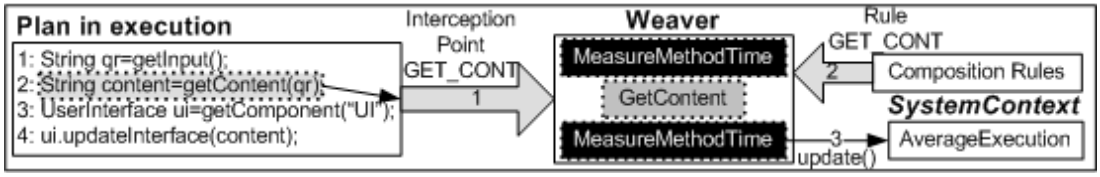

Figure 5.6: Graphical schema of the aspect weaving process example for response time monitoring in *VisitorAgent*.

functions, but we consider that a goal policy can be easily integrated inside a BDI agent. In order to avoid the combinatory explosion of states, we model the behavior of the self-adjusting as a State Transition Diagram (STD) . In each state, a specific number of *SystemContext* variables are considered and additionally, the reachable states from a given state are also determined. The states of the system that require self-management are modeled using situational patterns, which are a boolean expression about the *SystemContext*.

The self-adjusting is implemented as an aspect that is applied each time something in the *SystemContext* changes. This concern has been separated as an aspect because it clearly crosscuts the architecture of the agent. *SelfAdjusting* (see Algorithm 2) works as follows: for each transition associate to the current state (T_i), we check if its situational patterns (P_i) is held (lines 4-10). If P_i is held, then the STD associated to the *SelfAdjusting* aspect transits and the aspect generates goals to solve the situation (lines 11-12). These goals are transmitted to the Reasoning Engine of the agent, which selects plans to accomplish them in the *Plan Library* (see Figure 5.4). The algorithm assumes that the zero or one pattern is held and if it is the second, nothing in the system changes. In the case that more than one pattern is held, only the first of the patterns is considered.

The use of STD to model the behavior of the self-adjusting is inspired by the Strategy Realization System (SRS) presented in [Janik and Zielinski, 2010]. In our approach, the states of the STD represent the correct configuration of the agent and when the STD transits, new aspects and rules for composition are enabled and disabled in order to achieve a goal. The programming of the STD has been checked at the modeling stage using APPEL and the UMC Model checker. The self-management strategy of the *SecurityAgent* is an example of this (see Figure 5.7).

Algorithm 2 $SelfAdjustment(C, S)$

Input: the *SystemContext* C, the self-adjusting state $S = \{T_1, T_2, \ldots, T_n\}$ where $T_i = (P_i, G_i, S'_i)$ and P_i is a situational pattern, G_i are a set of goals and S'_i is the next state of the self-adjusting algorithm.

Output: a set of system level goals G

1: $match \leftarrow false$
2: $i \leftarrow 0$
3: $G \leftarrow \emptyset$
4: **while** $!match \wedge i < n$ **do**
5: **if** $P_i(C)$ **then**
6: $match \leftarrow true$
7: $G \leftarrow G_i$
8: **end if**
9: $i \leftarrow i + 1$
10: **end while**
11: **if** $G! = \emptyset$ **then**
12: $S \leftarrow S_{i-1}$
13: **end if**

By default, *SecurityAgent* has an *Encryption* aspect enabled to send information to *GuideAgents* (non-priority encryption) and to *SecurityAgents* (priority encryption) (State 1), but the main priority is that the device in which the agent is running continues to work. When the battery is running low, to save energy the first step is to disable the non-priority encryption. This is not a trivial task, the encryption takes place on both sides of the communication and a *GuideAgent* can receive information from various *SecurityAgents*. Therefore, it is not reasonable for all the *GuideAgents* in the museum to disable the *Encryption* aspect for a specific agent. In order to resolve this issue, we modify the route of the messages of *SecurityAgent* using a special *SecurityAgent* named *SuperAgent*. This agent acts as a bridge between an agent and a group of agents, indicating to the group of the agent that it will be the new receptor of the messages for a specific agent. So, before disabling the non-priority encryption, the *SecurityAgent* sends a request to *SuperAgent* (State 2) and when it receives the confirmation from this agent (State 3), it generates goals to disable the encryption. We accomplish these tasks by modifying the composition rules of the *SecurityAgent*, i.e. adding and removing aspects and modifying its transitions for its application. When the system transits

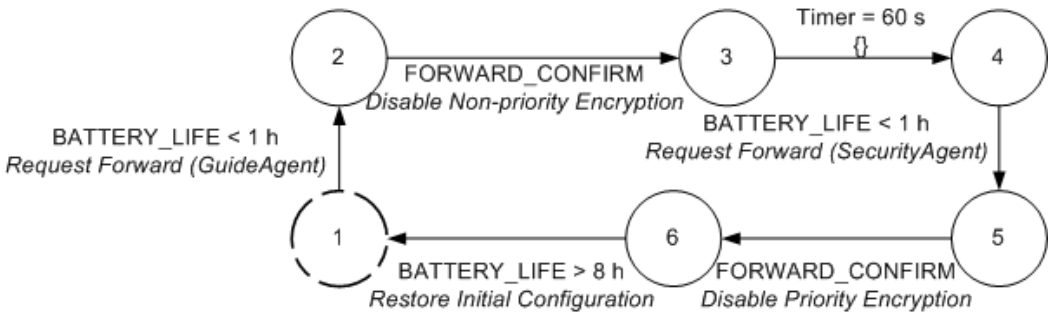

Figure 5.7: State Transtion Diagram of the self-management strategy for *SecurityAgent*.

to State 4 and the battery is low we follow a similar procedure as the one used to disable the non-priority encryption, but for the priority encryption (States 4, 5 and 6).

Plans that accomplish *SystemLevelGoal* (showed in italic tags in arrows in Figure 5.7) are selected by the Reasoning Engine from the Plan Library. This stores a set of Plan Description elements that specifies the class of the plan, the preconditions for the execution of such a plan and the goals that they can fulfill. Plans are sent to the *Scheduler* (Figure 5.3 and Arrow 4 in Figure 5.4), which executes them. Plans for *SystemLevelGoals* use the *Agent Configuration Service*. This interface has the same methods and purpose as the *AgentConfigurationService* of MalacaTiny (see Figure 5.1).

5.2 Code generation process of the MalacaTiny agents

In order to generate executable code for AmI devices, a final step in the process is necessary. Given the XMI file (corresponding to the modeling of our MAS), this is translated to Java code by a set of M2T transformations. We have a M2T transformation process for each type device and version, involved in MalacaTiny. That results in 4 M2T transformation processes implemented using xPand (see Subsection 2.4.4.3). These processes are very similar and the main differences between them are the self-management concern of MalacaTiny and Goal-Oriented Mala-

caTiny. So, in this Section we focus on the generation of MalacaTiny for devices with MIDP profile and point out their differences to Goal Oriented MalacaTiny.

5.2.1 Code generation of the internal architecture of agents

Firstly, each *Agent* element in the MalacaTiny model file is equivalent to a *MalacaTiny* class. So, for each *Agent* a *MalacaTiny* class with the same name, knowledge, context, components, facet, aspects and composition rules is generated (see Figure 5.1). With the information provided by the Pineapple metamodel, it is only possible to derive empty Java classes for knowledge, context, components and facets with the corresponding name. Therefore, these elements are added but they are linked to artifacts (classes, interfaces, attributes) that must be filled in by programmers.

For the aspects, there is a default set of aspects, principally corresponding to agent communication (such as *JadeLeapPlugin* or *SolPlugin* for the distribution aspect in Figures 5.5 and 5.3), that have already been implemented. The incorporation of these aspects and their composition rules are derived from the M2M transformation process from Pineapple to Malaca, and the use of predefined aspect patterns. For these aspects, the programmer only needs to indicate the IP address and port in which the agent platform is running.

Returning to our case study, Example 5.2.1 shows the generated implementation of the composition rules for *SensorAgent*. The composition rule for sending messages (*SND_MSG*) requires the composition of the representation aspect named *AURORA_ADAPTOR* aspect (line 10) and then the *ORDINARY_ADAPTOR* distribution aspect (line 11). In the interception point *RECV_MSG* requires, firstly the composition of the Representation aspect (line 14) and then one of the following coordination aspects or self-adjusting aspects (lines 15-17), if the message belongs to the corresponding protocol, determined by an instance of the class *MessagePattern*: *dataProvider_MP* (line 2), *sensorSelfOptimizing_MP* (line 4), *or sensorSelfHealing_MP* (line 6). This code is generated automatically from the agent in the MalacaTiny metamodel. Generated values for *RoleInstance* and *Scope* usually have to be changed to guarantee the correct behavior of the system. By default, *Scope* for coordination aspects is *PROTOCOL_SCOPE*, but it depends on

the application whether this is applicable or not. If it is necessary to change the *Scope*, then *RoleInstance* is modified using the Malaca aspect identification mechanism [Amor and Fuentes, 2009]. Finally, for the *THROW_EVENT* interception point (lines 19-22), all the coordination and self-adjusting aspects are applied (all the aspects catch the event). If an aspect is not supposed to handle the event, it simply does nothing (i.e protocol connector does not change its internal state and an exception is not thrown), but for greater efficiency it is preferable to remove the lines of code that do not match the *THROW_EVENT* interception point.

Example 5.2.1. Section of code of composition rules for the *SensorAgent* agent.

```
1     // Message patterns.
2     MessagePattern dataProvider_MP=new MessagePattern();
3     dataProvider.setProtocol("RequestRoomCondition");
4     MessagePattern sensorSelfOptimizing_MP=new MessagePattern();
5     sensorSelfOptimizing_MP.setProtocol("SensorSelfOptimizing");
6     MessagePattern sensorSelfHealing_MP=new MessagePattern();
7     sensorSelfHealing_MP.setProtocol("SensorSelfHealing");
8
9     // Message sending
10    applyAspect(InterceptionPoint.SEND_MSG, Role.REPRESENTATION,
          null, "AURORA_ADAPTOR", null, true, Scope.AGENT_SCOPE);
11    applyAspect(InterceptionPoint.SEND_MSG, Role.DISTRIBUTION,
          null, "ORDINARY_ADAPTOR", null, true, Scope.AGENT_SCOPE);
12
13    // Message reception
14    applyAspect(InterceptionPoint.RECV_MSG, Role.REPRESENTATION,
          null, "AURORA_ADAPTOR", null, true, Scope.AGENT_SCOPE);
15    applyAspect(InterceptionPoint.RECV_MSG, Role.COORDINATION,
          dataProvider_MP, "RequestRoomCondition",
          RequestRoomConditionDataProvider.class.getName(), true,
          Scope.AGENT_SCOPE);
16    applyAspect(InterceptionPoint.RECV_MSG, "SelfAdjusting",
          sensorSelfOptimizing_MP, "SensorSelfOptimizing",
          SensorSelfOptimizing.class.getName(), true, Scope.
          AGENT_SCOPE);
17    applyAspect(InterceptionPoint.RECV_MSG, "SelfAdjusting",
          sensorSelfHealing_MP, "SensorSelfHealing",
          SensorSelfHealing.class.getName(), true, Scope.AGENT_SCOPE)
```

```
        ;
18
19      // Event throwing
20      applyAspect(InterceptionPoint.THROW_EVENT, Role.COORDINATION,
            dataProvider_MP, "RequestRoomCondition",
            RequestRoomConditionDataProvider.class.getName(), true,
            Scope.AGENT_SCOPE);
21      applyAspect(InterceptionPoint.THROW_EVENT, "SelfAdjusting",
            sensorSelfOptimizing_MP, "SensorSelfOptimizing",
            SensorSelfOptimizing.class.getName(), true, Scope.
            AGENT_SCOPE);
22      applyAspect(InterceptionPoint.THROW_EVENT, "SelfAdjusting",
            sensorSelfHealing_MP, "SensorSelfHealing",
            SensorSelfHealing.class.getName(), true, Scope.AGENT_SCOPE)
            ;
```

Composition rules generated for *GuideAgent*, *SecurityAgent* and *VisitorAgent* have the same structure. The only difference is the composition rules for these agents (i.e. Goal Oriented MalacaTiny agents) include the advice to access aspect behavior.

5.2.2 Code generation of aspects

The initial *Coordination* aspect element is transformed to a set of Java classes; at least one class for each *Coordination* element is included, and additionally, one class for each *Transition* element. The resultant code for initial setup of *RoutePlanningRequester* behavior can be seen in Example 5.2.2. This example shows the setup method of this class, where the states and transitions of the FSM are defined. There is a direct mapping between the XMI of Figure 4.5 and the Java code in Example 5.2.2. Line 13 of the XMI file that defines the message pattern *messagePattern_10* is used to generate line 21 in the Java code. The *InstancePattern* defined in lines 8-9 is mapped to *initialEventRoutePlanning_IP* in the Java code (line 20). Lines 6-14 describe the transitions of the finite state machine at model level, that are mapped to lines 25-27 in Java code.

Example 5.2.2. Section of code in Java of the RequestRoomConditionDataPro-

vider coordination aspect.

```
1 public class RoutePlanningRequester extends ProtocolConnector {
2
3
4   public String getProtocolName() {
5     return "RoutePlanning";
6   }
7
8   @Override
9   protected void setup() {
10    // States
11    ProtocolState initialRoutePlanning=new ProtocolState(this,"
          InitialRoutePlanning");
12        ProtocolState forkRequestInformation=new ProtocolState(
              this,"ForkRequestInformation");
13        ProtocolState forkMyRoute=new ProtocolState(this,"
              ForkMyRoute");
14
15        // Goals
16        Goal sendRouteRequestGoal=new Goal("SendRouteRequest",
              GoalType.APPLICATION);
17        Goal sendMyRouteInformationGoal=new Goal("
              SendMyRouteInformation",GoalType.APPLICATION);
18
19        // Patterns
20        InstancePattern initialEventRoutePlanning_IP=new
              InstancePattern(new GUIEvent());
21        MessagePattern messagePattern_10=new MessagePattern();
22        dataProvider.setProtocol("RoutePlanning");
23
24
25        // Transitions
26        registerTransition(initialEventRoutePlanning_IP,
              initialRoutePlanning, forkRequestInformation,
              sendRouteRequestGoal);
27    registerTransition(messagePattern_10, forkRequestInformation,
          forkMyRoute, sendMyRouteInformationGoal);
28
29        // Setup
```

```
30      setInitial_state(initialRoutePlanning);
31      setCurrent_State(initialRoutePlanning);
32    }
33  }
```

One important difference between the code section of Example 5.2.2 and the model depicted in Figure 5.2.2 is the presence of goals. Pineapple and the MalacaTiny metamodel do not explicitly define agent goals, neither does MalacaTiny. However, this is not the case of Goal-Oriented MalacaTiny. The solution that we have adopted is to automatically generate goals from plans. If a plan can accomplish more than one goal, these goals are added to the plan after the M2M transformation using the Ecore utility provided by the Eclipse Modeling Framework. Therefore, the code generation for agents with MIDP profile has a direct mapping, while the transformation for Goal Oriented MalacaTiny is a bit more complicated.

Plans attached to goals or coordination aspects are generated in the same way for MalacaTiny and Goal Oriented MalacaTiny. The code section shown in Example 5.2.3 is generated from the SendRouteRequest plan of Figure 4.5. This plan requires the execution of the following internal tasks: receive the event that causes the execution of the plan (line 7), compose the route request (line 7) and compose and send the message requesting information to other agents (lines 9-14). Plans implement an interface that contains methods that must be directly implemented by programmers. In our example, the interface *SendRouteRequestInterface* (line 1) contains the methods *composeRequest()* and *composeSendRequestInformation(...)*.

Example 5.2.3. Section of code in Java of the *SendRouteRequest* plan.

```
1 public class SendRouteRequest extends Plan implements
     SendRouteRequestInterface{
2
3   @Override
4   protected void execute() {
5         Object receiveEvent=getInput();
6
7         Room source=composeRequest();
8
```

```
 9          ACLMessage sendRequestInformation=new ACLMessage();
10          sendRequestInformation.setOntology("
                IntelligentMuseumOntology");
11          sendRequestInformation.setPerformative(ACLMessage.REQUEST)
                ;
12          composeSendRequestInformation(sendRequestInformation,
                source);
13
14          getAgent().sendMessage(sendRequestInformation);
15    }
16       ....
17 }
```

5.2.3 Code generation of self-management

The code generation for self-management is where the M2T transformation for MalacaTiny and Goal Oriented MalacaTiny presents their biggest difference. As stated in Subsubsection 5.1.3.1, the MalacaTiny policies are hard-coded in the *analyze()* method of the abstract class *SelfManagement*. While Goal Oriented MalacaTiny uses goal policies implemented in an STD (see Subsubsection 5.1.3.2). However, the finite state machine of the SelfAdjusting aspect in the MalacaTiny metamodel (see Figure 4.4) is more similar to an STD. So, although the *SelfAdjusting* aspect of Goal Oriented MalacaTiny is more complex than the implementation of the *analyze()* method in MalacaTiny, the first code generation process is simpler.

In the case of the self-management for MalacaTiny agents, a class is generated that extends *SelfManagement* for each *SelfAdjusting* aspect of the MalacaTiny model. The behavior of the *SMMachine* included in the aspect is encoded in the *analyze()* method using a combination of the switch and the If-then-else control structure. Each state of finite state machine is a case of the switch structure (lines 29 and 43) and each transition for one state is an if-condition included in the case (lines 34 and 37 for *InitialSelfOptimizingState* and line 48 for *DecreaseLightMonitoringFrequencyPolicyState*). Plans associated with the *SMMachine* are stored in an internal plan library (lines 16-20) with an automatic generated goal that corresponds to the plan name (lines 2-5).

Example 5.2.4. Section of code in Java of the *SendRouteRequest* plan.

```
1     ...
2     public static String DecreaseLightMonitoringFrequencyPlan="
          DecreaseLightMonitoringFrequencyPlan";
3     public static String DecreaseNoisetMonitoringFrequencyPlan="
          DecreaseNoiseMonitoringFrequencyPlan";
4     public static String AcceptLightProposalPlan="
          AcceptLightProposalPlan";
5     public static String AcceptNoiseProposalPlan="
          AcceptNoiseProposalPlan";
6     ...
7     public static int InitialSelfOptimizingPolicyState=0;
8     public static int DecreaseLightMonitoringFrequencyPolicyState
          =1;
9     public static int DecreaseNoiseMonitoringFrequencyPolicyState
          =2;
10    ...
11    private int currentState;
12
13    public SensorSelfOptimizing(){
14
15        currentState=0;
16        this.addSMAction(DecreaseLightMonitoringFrequencyPlan,
              DecreaseLightMonitoringFrequencyPlan.class.getName());
17        this.addSMAction(DecreaseNoisetMonitoringFrequencyPlan,
              DecreaseNoisetMonitoringFrequencyPlan.class.getName());
18        ...
19        this.addSMAction(AcceptLightProposalPlan,
              AcceptLightProposalPlan.class.getName());
20        this.addSMAction(AcceptNoiseProposalPlan,
              AcceptNoiseProposalPlan.class.getName());
21        ...
22    }
23
24    public String analyze(Object input) {
25
26        String goal="none";
27
28        switch (currentState) {
```

```
29              case InitialSelfOptimizingPolicyState:
30                  boolean lowBattery=service.getKnowledge("
                        lowBattery");
31                  boolean activeLightMonitoring=service.getKnowledge
                        ("activeLightMonitoring");
32                  boolean lightMonitoringFrequencyModifiable=service
                        .getKnowledge("
                        lightMonitoringFrequencyModifiable");
33                  ....
34                  if (lowBattery && activeLightMonitoring &&
                        lightMonitoringFrequencyModifiable){
35                      currentState=
                            DecreaseLightMonitoringFrequencyPolicyState
                            ;
36                      goal=DecreaseLightMonitoringFrequencyPlan;
37                  }else if (lowBattery && activeNoiseMonitoring &&
                        noiseMonitoringFrequencyModifiable){
38                      currentState=
                            DecreaseNoiseMonitoringFrequencyPolicyState
                            ;
39                      goal=DecreaseNoiseMonitoringFrequencyPlan;
40                  }
41                  ....
42                  break;
43              case DecreaseLightMonitoringFrequencyPolicyState:
44                  boolean lowBattery=service.getKnowledge("
                        lowBattery");
45                  boolean activeLightMonitoring=service.getKnowledge
                        ("activeLightMonitoring");
46                  boolean lightMonitoringFrequencyModifiable=service
                        .getKnowledge("
                        lightMonitoringFrequencyModifiable");
47                  ...
48                  if(lowBattery && activeLightMonitoring &&
                        lightMonitoringFrequencyModifiable){
49                      currentState=RequestLightMonitoringState;
50                      goal=RequestLightMonitoringPlan;
51                  }
52                  ...
```

```
53                  break;
54              ...
55          }
56          return goal;
57      }
58      ...
```

In the case of self-management for Goal Oriented MalacaTiny agents, the mapping is direct. But in order to achieve policies with loops like the policy depicted in Figure 5.7 the developer must modify the MalacaTiny model. This is because ECA policies does not consider states. Example 5.2.5 depicts a code section of the *EncriptionOptimizer* self-adjusting aspect. The self-management goals of the policy are mapped from plans included in the transitions of *SMMAchine* (lines 5-7). *GoalState* elements, that represent states of the policy, are generated from *States* of the *SMMachine* element (lines 5-7). *SMTransitions* from MalacaTiny metamodel are transformed into *GoalTransition* objects, with the same states and patterns for transitions (lines 14-22).

Example 5.2.5. Section of code in Java of the *SendRouteRequest* plan.

```
1 public class HMSelfAdjusting extends SelfAdjusting{
2
3   @Override
4   public void setup() {
5     GoalState initialEncryptionOptimizerPolicyState=new GoalState(
          this);
6     GoalState requestGuideForwardPolicyState=new GoalState(this);
7         GoalState disableNonPriorityEncryptionPolicyState=new
              GoalState(this);
8     ...
9
10    Goal requestGuideForwardPlanGoal=new Goal("
          RequestGuideForwardPlan",GoalType.SYSTEM);
11        Goal disableNonPriorityEncryptionPlanGoal=new Goal("
              DisableNonPriorityEncryptionPlan",GoalType.SYSTEM);
12    ...
13
14        GoalTransition transition_12=new GoalTransition();
```

```
15          transition_12.setNextState(requestGuideForwardPolicyState,
                requestGuideForwardPlanGoal);
16          registerTranstion(initialEncryptionOptimizerPolicyState,
                new LowBatteryPattern(),transition_12);
17
18          GoalTransition transition_13=new GoalTransition();
19          MessagePattern messagePattern_18=new MessagePattern();
20          messagePattern_18.setPerformative(ACLMessage.CONFIRM);
21          transition_13.setNextState(
                disableNonPriorityEncryptionPolicyState,
                disableNonPriorityEncryptionPlanGoal);
22          registerTransition(requestGuideForwardPolicyState,
                messagePattern_18,transition_13);
23
24          ...
25
26      this.setCurrentState(initialEncryptionOptimizerPolicyState);
27    }
28 }
```

Finally, plans to accomplish self-management goals are mapped using the same procedure as described in Subsection 5.2.2, used for regular plans. Additionally, new mapping rules are required to generate the code for associated actions for self-management.

5.3 Summary

In this chapter we have presented the code generation process for MalacaTiny and Goal Oriented MalacaTiny. Using the metamodel presented in Chapter 4 we are able to generate code for the different versions of MalacaTiny. This requires having 4 M2T transformation process that has been implemented using the xPand technology. We are able to generate code for MalacaTiny and Goal Oriented MalacaTiny because the MalacaTiny metamodel, like the Pineapple metamodel, can accommodate agents with different architectures. However, the MalacaTiny metamodel does not have explicit goals that must be automatically generated in M2T process for Goal Oriented MalacaTiny.

In this chapter we have also shown the generation of the code for self-management. For MalacaTiny, the finite state machines presented in Chapter 4 are transformed in the implementation of the *analyze()* method from the *SelfManagement* class. In order to translate the states and transitions, we have codified a switch structure with if-conditions. The code generation for Goal Oriented MalacaTiny is simpler because it only requires translating the *SMMachine* model to Java code.

Chapter 6

The communication concern

In this chapter we present the two distribution aspects that we have developed specifically for MalacaTiny and Goal Oriented MalacaTiny agents, *Blue* and the *Sol* agent platform. *Blue* is a distribution aspect specific for mobile phones with MIDP profile that we developed to study the energy consumption of MalacaTiny agents. On the other hand, the *Sol* agent platform is our solution to communicate heterogenous devices with heterogenous communication means. This platform has been specially important for the development of the Intelligent Museum presented in Chapter 3.

This chapter is structured as follows: Section 6.1 describes the internal design of the *Blue* agent platform; Section 6.2 presents the *Sol* agent platform; and Section 6.3 summarizes the contributions of this chapter.

6.1 The *Blue* agent platform

To illustrate how we accomplish the Challenge C2.2 (*Cope with wireless network diversity*), we will demonstrate how to extend the distribution aspect to enable the communication of agents with different wireless technologies and without the use of agent platform distribution services. In AmI environments, the communication of neighboring devices is very popular. In this scenario, Bluetooth allows devices to wirelessly communicate over short distances. This wireless technology enables ad hoc networks to be formed dynamically between Bluetooth-capable de-

vices. Bluetooth makes it possible to use the Bluetooth network interface to send and receive TCP/IP transport data. However, Bluetooth's specification defines four major transport protocols, nearly all of which are special purpose. Of these protocols, the Radio Frequency Communications (RFCOMM) protocol is often the best choice, and sometimes the only choice (some environments only support this protocol). RFCOMM is a general-purpose reliable stream-based protocol. It provides, roughly, the same service and reliability guarantees as TCP. The Logical Link Control and Adaptation Protocol (L2CAP) is a packet-based protocol that is also widely used when the streaming nature of RFCOMM is not needed.

In order to enable MalacaTiny to communicate with other agents using Bluetooth RFCOMM connections, we just need to implement a new plugin. From the agent perspective, its functionality remains the same: the distribution of ACL messages is provided by the distribution aspect. However, this implementation of the distribution aspect encapsulates all the particularities of Bluetooth: device and service discovery, and the establishment of RFCOMM connections. Prior to data transmission, Bluetooth devices need to perform device inquiry and service discovery. When these processes have finished, a Bluetooth device can establish a connection that enables communication with devices nearby. The device that starts the search plays a client role in the communication and the devices discovered are servers. This initial process is not so different to the agent management service and the directory facilitator query in agent platforms[1].

A FIPA compliant agent platform offers a directory facilitator for service discovery, but a service must be previously registered in the directory facilitator. In a Bluetooth network, service discovery is very similar, since each Bluetooth-enabled device running an agent publishes services that can be found by clients (i.e. agents running in other Bluetooth devices). The published services allow the agents to communicate. Then the distribution aspect uses the Service Discovery Protocol (SDP) to discover devices (i.e. agents), and, for each device, it seeks a specific service that allows agents to exchange ACL messages (the approach in Bluetooth is to assign every single service a unique identifier). The service record returned by the SDP contains the URL that allows the distribution aspect, acting as a client, to establish a RFCOMM connection with the distribution aspect of the other agent

[1] `http://www.fipa.org/`

(that plays the role of a server). In parallel, the distribution aspect publishes its own service for communicating with other agents. Therefore, each agent that uses the distribution aspect for Bluetooth plays a server role and additionally, if it seeks other agents it is considered to be an inquiring client. Finally, both roles are a way to get Bluetooth connections, which are identified by the Bluetooth address of the other side's device. As a Bluetooth address is the unique identifier for each device, this guarantees that each connection can be univocally identified.

The design of this aspect is shown in Figure 6.1. The *BluetoothPlugin* class is in charge of the communication between agents using Bluetooth. By default it sets up a *BTHost* object that waits for requests from *BTClient* objects (a Bluetooth device never inquiries in itself). When one of these artifacts sets a connection, it creates a *BluetoothConnection* object that is transmitted to *BluetoothPlugin*, which keeps a *hashtable* with all connections indexed by the Bluetooth address as a key. The purpose of *BluetoothConnection* is to hide the complexity of the message exchange from *BluetoothPlugin*. The class *BlueAgentPlatform* implements the *FIPAAgentPlatform* interface, providing agent platform services to the MalacaTiny agent and additionally it is in charge of initializing the internal objects of the distribution aspect (e.g. the *BluetoohPlugin* object). Since the data transmitted between Bluetooth devices using the RFComm sockets cannot be Java objects, then a message representation aspect based on *String* format has also been developed. This format is called *BlueMessage* and the aspect that processes messages in this format is called *BlueMRepresentation* (see Figure 6.1).

The effort required to develop this plug-in and integrate it as part of the agent internals to support a new communication protocol was not very high, thanks to the good modularization of the MalacaTiny architecture that uses aspects orientation. In addition, the plug-in can be reused in any MalacaTiny agent, regardless of whether the MAS of the agent is involved. As a result, agents can decide to use this communication protocol, to support the communication of neighboring agents. Moreover, it is possible to provide plug-ins to enable the discovery and communication of agents using other wireless technologies (e.g. IEEE 802.15.4/Zigbee for agents running in Sun SPOT sensors), or the services provided by new agent platforms as they appear. In addition, the composition of aspects inside the agents allows the use of more than one plug-in at a time. This feature makes the com-

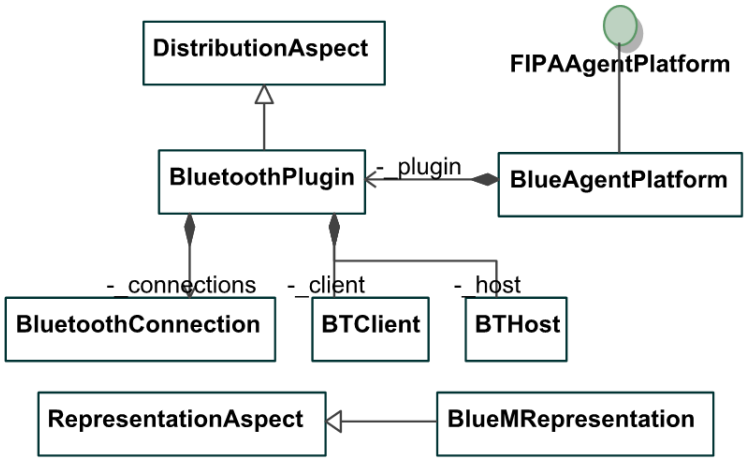

Figure 6.1: UML class diagram of the distribution aspect for Bluetooth in MalacaTiny.

munication of agents in multiple heterogeneous devices possible, even at the same time.

6.2 The *Sol* Agent platform

In this Section we present the internal design of the Sol agent platform. This platform addresses limitations of agent technology in the AmI environment at two different levels: (i) improving the design of a communication subsystem inside the agent architecture (i.e. agent level) to facilitate the reconfiguration of an agent communication mechanism to adjust it to different contexts; and (ii) endowing the agent infrastructure with the necessary means to manage interoperability limitations because of device and communication protocol heterogeneity, and extending the message transport service provided by the agent infrastructure to support an efficient group communication (i.e. the agent platform level). At the agent level, we provide agents with the capacity to self-configure their internal architecture in order to use different communication protocols, taking into account the context and the necessities of the application. This flexibility inside the agent's internal design also makes the simultaneous use of different message distribution mechanisms easier. Sol facilitates (1) the communication and interoperation of agents running

in heterogeneous sets of devices (such as SunSPOT sensor motes, Android-based lightweight devices and other mobile phones) and is even able to use different communication protocols; (2) group message delivery (one-to-many communication) efficiently. Sol supports the native communication protocols of each device (e.g. ZigBee, WiFi) and acts as a gateway, performing specific functions in order to ensure interoperability. In order to address the efficient group message distribution, the *Sol* agent platform provides support for membership management (joining and leaving members) and adequate communication mechanisms when possible (e.g. using IP multicast). We illustrate the benefits of our approach for several scenarios in our case study.

6.2.1 An agent platform for AmI applications

6.2.1.1 The *Sol* agent platform services

The *Sol* agent platform (see Figure 6.2), partially implements the FIPA[1] abstract architecture for lightweight devices. The main goal of *Sol* is to support the interoperability of agents deployed in different AmI devices, through heterogeneous communication protocols. The current version of the platform works with wireless personal and local area networks (WPAN and WLAN) mainly composed of phones with MIDP profile, Android devices, Sun SPOT sensor motes, and recently also Libelium waspmotes[2]. This agent platform acts as a middleware that provides a set of services to the agents that are deployed on it, and behaves as a gateway to support communication heterogeneity (fulfilling the Challenge C2.2-*Cope with wireless network diversity*). Specifically, our agent platform supports:

1. The registering and discovering of agents (AMS).

2. The registering and discovering of services (DF).

3. The registration and membership of groups (Group Management Service-GMS).

[1] `http://www.fipa.org/`
[2] `http://www.libelium.com/products/waspmote`

4. A message transport service (MTS), which allows the communication between agents registered in the agent platform, extended to facilitate the distribution of group-based communication.

The distribution of communication messages is supported internally by the Internal Platform Message Transport (IPMT), which realizes the MTS and resolves communication interoperability issues. Note that the AMS, DF and MTS are classical services provided by any agent platform (these services are defined by the FIPA Abstract Architecture Specification [for Intelligent Physical Agents, 2002]). Apart from these services, we have extended the MTS to support an efficient group communication, which is complemented with the new GMS service.

The internal design of *Sol* is shown in Figure 6.3. The main class (named *Sol*) encapsulates the provision of the services enumerated below and stores information about the agents and groups deployed in the platform and also about the services provided by agents that are signed up, in the MAS. However, agents do not interact with this class directly to access these services. Instead, all the interactions between *Sol* and the attached agents is ACL message-based. The ACL messages are represented in a special string-based format named *SolMessage* (see Fig. 6.3).

Once an agent starts its execution, its first interaction is to join the agent platform (i.e. register in *Sol*). Requests for registration are attended to by *Listeners*. The platform provides specific listeners for different protocols and technologies (TCP, SunSpot, Bluetooth listeners in Figure 6.3). Internally, these classes instantiate threads which have sockets that listen to requests from hand-held devices (Android enabled devices and mobile phones with MIDP profile) and Sun SPOT sensor motes. Agents send a request message to register in the agent platform through this listener. In the registration message they specify their type, their identifier and the set and type of transport protocols (e.g. Bluetooth) supported. This information is stored in an instance of the *AgentProfile* class. Agent profiles are stored in a hash table (attribute *profiles*) indexed by the agent identifier. This data will be used to help the agent access the MTS. Additionally, *Sol* adds each agent to a group composed of all the agents with the same type. Both MTS access and group management are described further in this section. Registration has to be performed by every agent in the Museum (*Guide*, *Visitor* and *Sensor* agents).

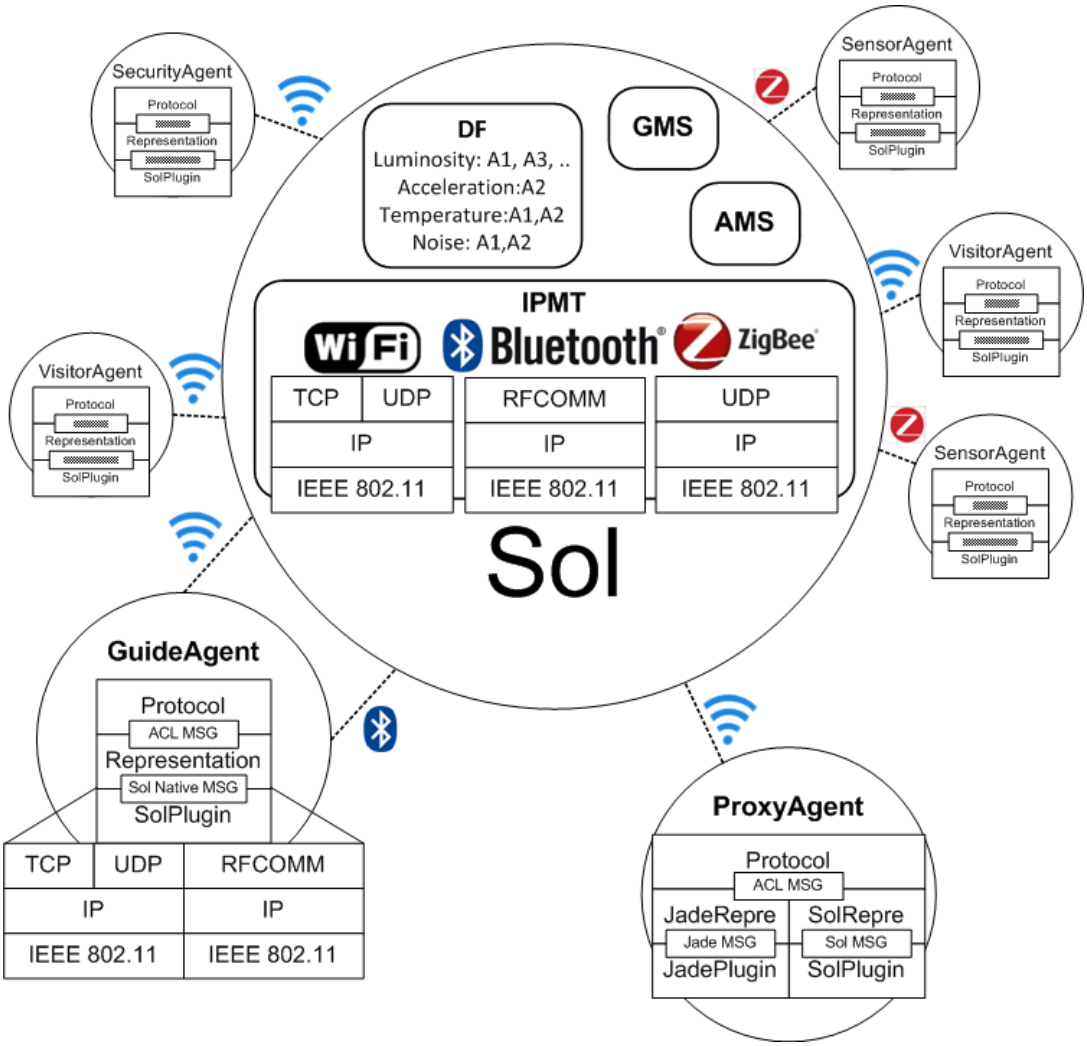

Figure 6.2: Schema of the communication in the *Sol* agent platform

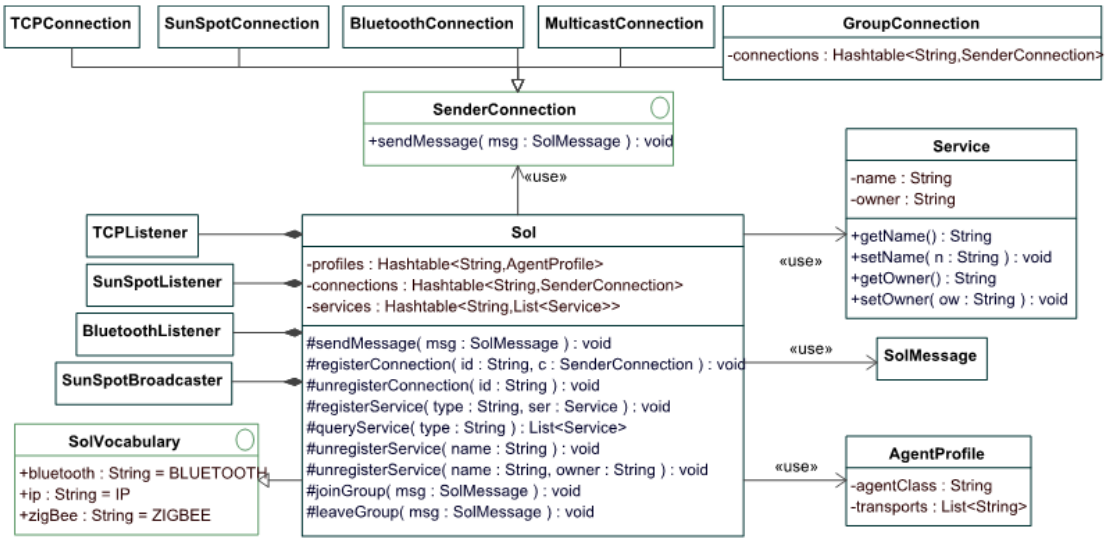

Figure 6.3: UML class diagram of the *Sol* agent platform.

The DF provided by *Sol* supports the main functions of a FIPA DF. This service is also supported by a set of specific listener classes. Agents may register their services with the DF or query the DF to find out what services are offered by other agents. Service descriptions provided by agents when they register with the DF are stored in an internal hash table (*services* attribute in Figure 6.3). In the IM, each agent registers the services that it can provide to the system. Service exchange between agents registered in the platform is illustrated in the following scenario.

Scenario 1. *In our museum, Room 2 has different types of sensors with an embedded SensorAgent, which can measure and provide data on acceleration, luminosity, person detection, and temperature. Let us consider that it is near closing time, so the Security agent has to close this room, but before doing so they need to check whether Room 2 is empty or not. This action requires monitoring the sensor dedicated to person detection.*

Each SensorAgent is registered in the agent platform (DF) as a provider of a certain type of sensing data for Room 2. So, the SecurityAgent queries the DF of Sol to find out the identifier of the agent providing this service (person detection in Room 2). Finally, it sends a request for data, to this sensor agent, which periodically sends inform messages with data on person detection until it detects

that the museum is completely empty and can be closed.

6.2.1.2 Managing Groups at *Sol*

Another of the contributions of *Sol* is the support for different communication paradigms and technologies. Specifically, *Sol* supports peer to peer communication (the usual communication paradigm in FIPA compliant MAS) and multicast communication, which is often required by ubiquitous systems. Multicast communication facilitates the distribution of the same information to clustered components of the system. In order to introduce this kind of communication in the MAS, the *Sol* agent platform incorporates the concept of group. A group is formed by a set of agents that share features. By default, there is a group for each type of agent that comprises the system, but additionally the user can define its own groups taking into account the role that the agent plays in the MAS and the application's communication needs. The different groups that are formed in the museum are illustrated in the following scenario.

Scenario 2. *In the museum, there are groups for each category of staff: a group formed by the guides, a group that agglutinates the security staff members, and the visitors are also organized into different groups. Individual visitors (i.e. VisitorAgents) only belong to the global visitors group, but visitors of organized groups (e.g. the "Colegio El Atabal" group) form a separate group with the corresponding guide composed of a GuideAgent (e.g. Inmaculada) and a number of Visitor agents (those registered in the "Colegio El Atabal").*

Note that membership of a group could change over time and according to the context. The members of the group ("being in Room 2") depends on the current location of *Guide*, *Visitor*, and *Security* agents. The creation of groups can be done at any time. With *Sol* it is easier to advertise information related to the museum, relevant for visitors and staff, such as "closing time is near", or "there is a special offer in the cafeteria"). Figure 6.4 depicts the user interface for notifications to a visitor in the museum. How group communication can make the functioning of the museum more efficient is illustrated in the following scenario.

Scenario 3*. Since it is usual to communicate simultaneously with a set of service providers, we can define a group to include all the agent providers of the same*

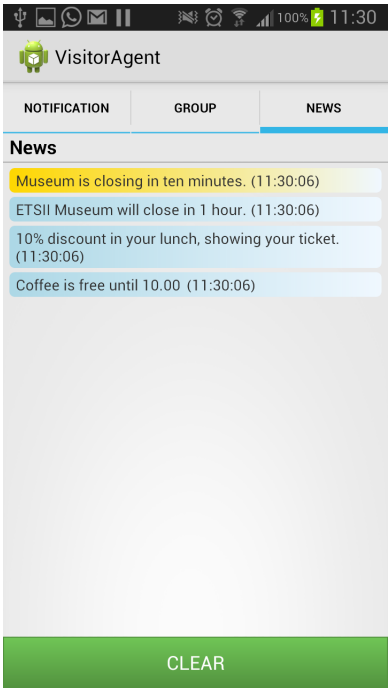

Figure 6.4: Visitor user interface for the reception of notifications.

service. A group is defined to facilitate the propagation of information between agents, and to make service provision more efficient. In the museum, there is a group for all the SensorAgents that monitor a specific room. Although these agents monitor and provide different data (e.g. presence or humidity), they are physically situated in the same room, which is considered one of the common features used to define the group (i.e. their location in the museum). The other feature is "being a sensor agent". This way, when Room 2 of the museum is empty (i.e. no users are in it) a message "room empty" can be sent to the group and SensorAgents can decrease their activity in order to save energy, which is crucial to making the system sustainable in terms of energy.

From Scenarios 2 and 3, we can see the high demand of group creation in our system, and how it facilitates the communication between the agents. The creation and the maintenance of groups are left to the GMS provided by *Sol*. This service supports group creation and allows agents to join and leave groups. Agents request to join a group, usually as part of the AmI application functionality. If it is the first member of the group, then a new group is created represented by an instance

of *GroupConnection* in the GMS. This data structure maintains information about the active members of the group which makes a reliable group communication possible. The joining of an agent to a group and the information that is stored in the corresponding *GroupConnection* depends on its profile (detailed in the *AgentProfile* class in Figure 6.3). If the agent supports TCP/IP and multicast IP, a *MulticastConnection* is created (even if there is no *MulticastConnection* in this group) and added to the *GroupConnection*. A *MulticastConnection* is assigned a multicast IP address. Once the joining process ends, the agent platform sends a message to the agent that includes the IP multicast address and port (how the agent uses this information to complete the joining process is described in Section 6.2.2.2) associated that group. If the agent does not support TCP/IP (or multicast IP) then the individual (and preferred) agent transport address (represented by an implementation of *SenderConnection* interface) is included in the group connection object. In Figure 6.5, we can see a UML sequence diagram that depicts the registration process of the first *VisitorAgent* that joins a group in the IM. In this case, the agent has support for multicast, so the registration process retrieves the IP and port of the new *MulticastConnection* inside the *GroupConnection*. The sequence of actions that an agent performs to join different groups is illustrated in Scenarios 4 and 5.

Scenario 4. *Ernesto has decided to visit the museum. When he enters the main hall, he is invited to download and install the museum application on his mobile phone. When the application starts up, a new VisitorAgent is created. The new agent is registered in the AMS and joins a group that represents new visitors. A minute later, a message announces that a new guided tour will start in a few minutes and invites him to join.*

The user is unaware of how this information is disseminated, but the fact is that Ernesto's mobile phone only supports Bluetooth connections. In the case of Ernesto's visitor agent, when it joins the newcomers group, an active BluetoothConnection object is added to the corresponding GroupConnection (if the connection is not established, then it is created).

Scenario 5. *A new sensor for Room 2 is going to be installed. Zigbee-based sensors communicate using IPv6 on top of IEEE 802.15.4 access protocol. Sensors transmit data on special UDP datagrams named radio datagrams. The agent that*

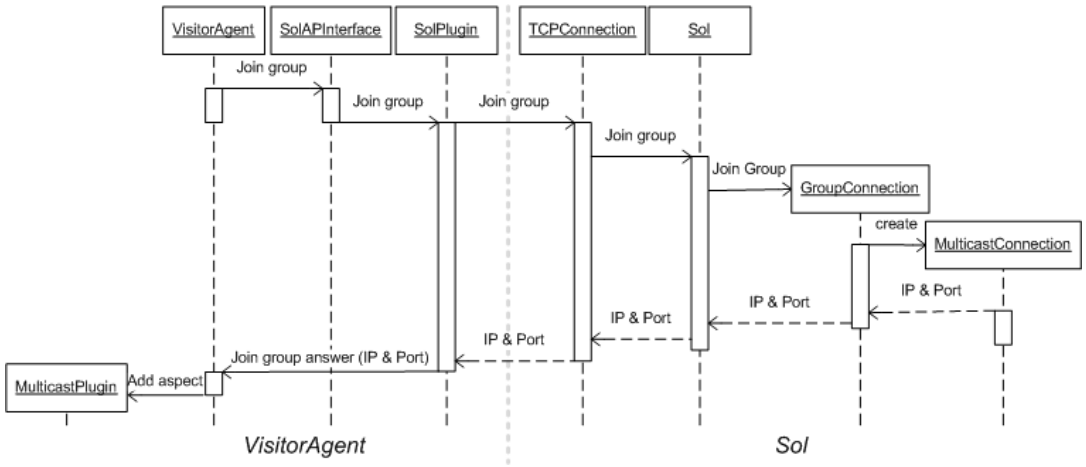

Figure 6.5: UML sequence diagram of the joining of an agent to a group in *Sol*.

will run in this new sensor is programmed to join the group of "sensors in Room 2". However, the joining of the new agent does not require any special type of connection (i.e. MulticastConnection) because the sensor agents already use UDP-based datagrams to communicate, so it just needs to obtain the IP multicast address and the corresponding port to correctly configure its own SolPlugin instance.

6.2.1.3 Extending the MTS to support multicast

The MTS delivers messages between agents registered in *Sol*. All the agents have access to at least one specific MTS provided by the agent platform. *Sol*'s MTS is supported by a set of connections. For each agent that is registered in the platform, the platform maintains a connection. After registering the agent through the AMS, a connection between the agent and the agent platform is established using the technology access that has been detailed at the registration stage. A connection is supported by concrete implementations of the *SenderConnection* interface. All the active connections are stored in a hash table. For each connection, the information stored is the identifier of the agent or the group in the agent platform and a class that implements the interface *SenderConnection*. *Sol* supports five types of connections (i.e. implementation of the interface): *TCPConnection*, *BluetoothConnection*, *MulticastConnection*, *SunSpotConnection* and *GroupConnection*.

The *TCPConnection* and *BluetoothConnection* are used by agents running in

hand-held devices to send and receive messages by means of TCP sockets and Bluetooth connections. These devices can receive multicast messages using UDP sockets by means of *MulticastConnection* objects. The *SunSpotConnection* is used to send UDP datagrams in order to communicate with Sun SPOT sensor motes. Finally, *GroupConnection* represents a group of devices and stores an internal list of connections (*connections* attribute), which can refer to the other types of connections that have been described. With this design it is easy to add new devices and communication protocols to the MTS of the agent platform, since we only have to implement a listener (for the AMS and the DF service) and the *SenderConnection* interface for the new type of connection or device specific communication mechanism. The features that are described at this point address the Challenge C2.2 (*Cope with wireless network diversity*), because *Sol* has support for heterogeneous communication means.

In addition, we have already checked how this design meets Challenge C2.2, paying particular attention to the flexibility of the communication infrastructure. The case of Sun SPOT sensor motes is special and different to Bluetooth and classical TCP/IP because these devices connect to *Sol* by means of the so called Sun SPOT base station. The problem is that each time the base station is plugged into the system it is bound with a different IPv6 address, which must be known by the sensor motes before connecting to it. Therefore an initial discovery process is necessary. This discovery process is implemented in the *SunSpotBroadcaster* class, which is a thread that periodically sends broadcast messages with the IPv6 address of the base station. Finally, *SunSpotListener* is an UDP socket to listen/sending datagrams from/to Sun SPOT sensor motes.

As mentioned, the distribution of messages to groups is also implemented as another type of connection represented by the class *GroupConnection* (in Figure 6.3). An instance of *GroupConnection* has an internal hash table of *SenderConnection* implementations. The reason is that, although the best way to multicast a message to a group is to use a multicast IP address, we can not be sure that all the devices in the group support TCP/IP, UDP or IP multicast. So, although *Sol* defines an IP multicast address to identify each group and use UDP sockets to send multicast messages, the agent platform makes sure that the group's members, that do not support IP multicast, can also receive group messages by sending unicast

messages. In this way, we still fulfill C2.

6.2.1.4 Services for MAS administrator

The current version of *Sol* offers a graphical user interface (GUI) that allows system administrators to visualize and manage the MAS, agents and groups deployed in the agent platform. The GUI, which is called *SolGUI*, provides the MAS administrator with access to the main agent plaform services (i.e. AMS and DF), in a similar way as does the administrator GUI of the Jade agent platform [Bellifemine et al., 2001]. In addition, the GUI allows access to the GMS (see Figure 6.6). Therefore, we can see in the main interface, each agent deployed in the agent platform with its identifiers (tree view on the left side). The left hand side is dedicated to each service's (AMS, DF and GMS) facilities, which are assigned to different tabs. AMS and DF views allow the visualization of the agents and services registered in these services, respectively. The button of the main window allocates a specific interface to visualize the platform's status messages.

SolGUI provides an interface (tab labeled "GMS") to monitor and control the different groups registered in the platform (see Figure 6.6). This panel shows all the groups, and, when a group is selected, its members and the different transport addresses associated with the group are shown. Additionally, it offers the possibility to create and remove a group (buttons labeled "Add Group" and "Remove Group", respectively), and add and remove members of the existing groups (buttons labeled "Add Agent" and "Remove Agent" respectively). In addition, it supports sending a message to the selected group (button labeled "Send Message").

6.2.2 Supporting interoperability between heterogeneous devices

In this section we describe how our approach manages heterogeneity and enables the communication of agents through different network technologies, transport protocols and agent platforms. Figure 6.2 provides a graphical schema of the set of access network technologies and transport protocols used by the agents involved in our AmI application, and how the *Sol* agent platform is able to support and manage such heterogeneity.

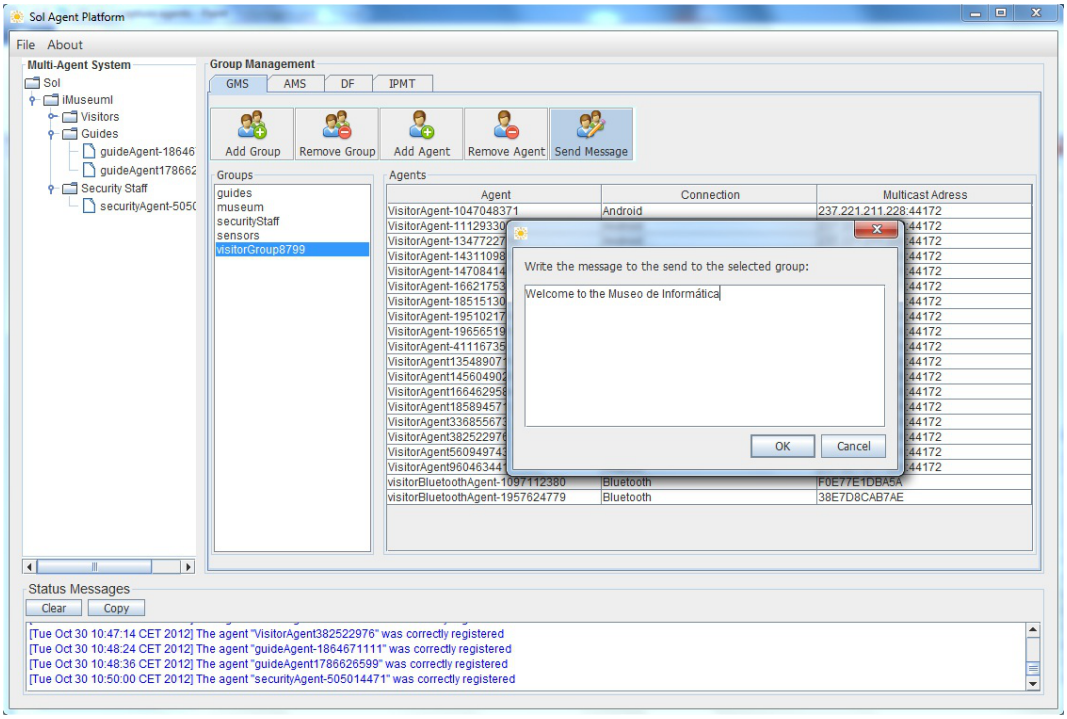

Figure 6.6: Interface for the visualization and management of groups in the *Sol* agent platform.

6.2.2.1 Supporting interoperability in the agent infrastructure

Currently, the *Sol* IPMT supports Bluetooth RFCOMM, UDP and TCP protocols at the transport level (protocol stacks for these network connections are detailed in Figure 6.2). This feature makes it possible to use *Sol* as a gateway since it can resolve the incompatibilities of different communication protocol stacks (mainly at the data link level). Sensors are normally connected, forming a WSN, using the IEEE 802.15.4 standard (also known as ZigBee), while the connectivity of typical personal hand-held devices (such as smart phones, mobile phones and tablets) is supported by WiFi (IEEE 802.11) and Bluetooth (IEEE 802.15.1). So, *Sol* allows Sun SPOT sensor motes to communicate with each other and with the so-called Sun SPOT base station using ZigBee, but it also allows them to send and receive data from hand-held devices, behaving as a gateway. Apart from the differences in the data link level protocol, the *Sol* implementation has to deal with the limitations imposed on the TCP protocol implementation, by the 802.15.4 network interface, especially those related to the number of active connections. The heterogeneity of our case study and how this can be solved using the *Sol* agent platform is illustrated in the following scenario.

Scenario 6. *There is a SensorAgent monitoring the presence of people in Room 2. Each time a visiting group or a person enters or leaves Room 2, the SensorAgent sends data to the SecurityAgent that runs in Luis's smartphone (who is member of the security staff and is assigned to control Room 2). However, Luis's smartphone only supports WiFi and Bluetooth connectivity, while sensors communicate through Zigbee networks. In this case, the SensorAgent in the sensor sends the data in a UDP datagram, which is received and processed by the Zigbee interface of the* Sol *agent platform. The IPMT extracts the ACL message (which contains the target agent and the data) from the UDP datagram and sends it to Luis's SecurityAgent. The ACL message is now encapsulated as a byte stream of a TCP connection and sent to the WiFi interface to be received by the Security agent.*

6.2.2.2 The self-configurable communication concern

The basis for achieving Challenge C2.2 flexibly is an agent architectural design and implementation that endows software agents with enough flexibility to communicate using different access technologies and communication protocols (even simultaneously). By doing this, we solve the interoperability issue at the agent level (as part of the agent architecture itself), which gives more control to the agent developer to adapt the agent to communicate through any network interface supported by the device in which it is embedded. The agent platform presented has been designed to work with the family of MalacaTiny agents. Anyway, *Sol* can provide useful services to any agent implementation interested in providing self-configurable communication facilities. As we stated in Chapter 5, MalacaTiny agents can self-configure their communications easily, by using different distribution aspect implementations to distribute messages. These distribution aspect implementations can be changed at runtime and several of them can even be used simultaneously whenever necessary.

The distribution aspect for *Sol* (i.e. *SolPlugin* in Figure 6.7) has the same design for the agents running in the different devices (sensors, hand held devices and desktop computers). This aspect allows the agent to communicate by means of different MTSs and network technologies through *Sol*. The main functionality of this aspect is the distribution of agent communication messages: it receives the incoming messages and delivers outgoing messages through a specific network access technology (Wi-FI or Bluetooth) or communication transport protocol (TCP, UDP or RFCOMM). This aspect hides any infrastructure dependency (derived both from the use of a specific agent platform or wireless technology) defining a high-level interface to send and receive messages to and from different communication technologies.

For each network interface and protocol the agent can access, this aspect is in charge of instantiating the corresponding device and API-dependent functionality. For example, Sun SPOT sensor motes use UDP datagrams to distribute sensed data. When the agent joins a group, this aspect deals with the instantiation of a new UDP socket for receiving group messages. The IP multicast address and the corresponding port is provided by the GMS of the *Sol* agent platform when

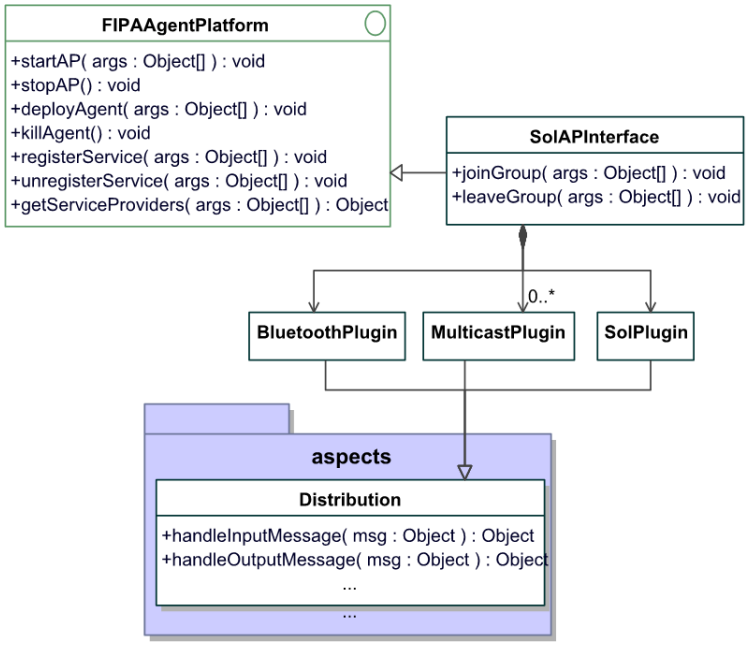

Figure 6.7: UML class diagram of the *SolPlugin*

the agent joins the group. The aspect also notifies the transport layer that it has joined a multicast IP address.

However, these devices require a discovery process before being able to send any data through the network interface (similar to Bluetooth). The distribution aspect realizes this discovery process when the agent is initializing. Moreover, the implementations of the *SolPlugin* distribution aspect for Android devices and mobile phones with MIDP profile differ, mainly in how each API sets and establishes the TCP connection.

Although the implementation of each distribution aspect differs, all of them together make up the *FIPAAgentPlatform* interface. This interface is common to all the distribution aspects and allows a uniform access to FIPA compliant agent platforms. The *SolAPInterface* class extends the services of *FIPAAgentPlatform* with services to allow the joining and leaving of groups.

Internally, the functionality of this communication concern is divided into two main parts (see Figure 6.7): the access to the *Sol* platform services (i.e. the AMS, DF and GMS services); and sending and reception of communication messages through the MTS of the platform (*SolPlugin*, *MulticastPlugin* and *BluetoothPlu-*

gin).

The *SolPlugin* and the *BluetoothPlugin* classes extend the *Distribution* aspect and allow sending messages through the *Sol* agent platform using a specific transport or access technology. They are implemented as threads that listen to messages. In the case of the *SolPlugin*, from a TCP socket connection established with the *Sol* agent platform, and in the case of the *BluetoothPlugin* from a RFCOMM Bluetooth connection to a service also running in *Sol*. The case of *MulticastPlugin* is special because it is a thread for only listening to messages targeted at multicast groups. As stated before, agents can ask the platform to join a group and this is done by means of the *SolAPInterface*. If the technology for the connection is IP then *Sol* answers the request with a multicast IP address and a transport port (see Figure 6.5). With this information a new *MulticastPlugin* is instantiated and added to the agent architecture. If an agent wants to send a message to a group (see Figure 6.8), this is done via *SolPlugin* or *BluetoothPlugin* and uses, as receivers, the identifier of the group. When the message arrives in *Sol*, this sends the message using the corresponding *GroupConnection* as described in Section 6.2.1.1. Figure 6.8 depicts the situation in which a *GuideAgent* (that uses RFCOMM Bluetooth) sends a message to a group of *VisitorAgents*. As you can see, when the message arrives to *Sol*, this sends the message using the *GroupConnection* that sends the message to the different connections included in the group.

As stated before, aspect orientation gives the MalacaTiny agents the possibility of using different agent platforms and mechanisms for communication. But as a counterpart it also requires the transformation of the messages during sending and reception in order to compose messages in the format used by the underlying agent platform. This task can be defined as a translation, which is encapsulated in the *Representation* aspect [Amor and Fuentes, 2009]. For our case study, *SensorAgent* and *SecurityAgent* are involved in an interaction ruled by a protocol that consists of *SensorAgent* send lectures of the light sensor to the *SecurityAgent* under request. The *SensorAgent* sends an ACL message that is transformed by the *Representation* aspect to a *SolMessage* and is sent to the *Sol* agent platform as a datagram via *SolPlugin*. When the message arrives at the platform, this is sent by means of the *TCPConnection* that sends the message as a stream of bytes to the *SecurityAgent*.

A differentiated feature of MalacaTiny agents is that they can simultaneously

use different network access technologies, transport protocols, and even agent platforms. This is possible simply by instantiating more than one *Distribution* aspect, one each per communication mechanism used. In addition, an agent with two or more distribution aspects instantiated can act as a proxy between agents in different agent platforms, using different transport protocols or access network technologies. However, given that *Sol* resolves heterogeneity of transport protocols and access network technologies satisfactorily, the use of the MalacaTiny agent as a proxy is left to resolve interoperability between agent platforms. The use of proxy agents is illustrated in the following scenario.

Scenario 7. *As a first approach in the Museum application, we had previously developed an agent-based solution implemented in Jade-Leap to organize groups (given a set of visitors, the agent-based application distributed them in groups depending on user profile, such as age and physical disabilities) and calculate the optimized route to visit the museum. The Jade agent in charge of organizing groups (named GroupOrganizer) is still useful for our system, so we wanted to reuse it. But, Jade-Leap agents communicate using WiFi, and TCP/IP, so the MTS of Jade-Leap cannot interoperate with the IPMT of the* Sol *agent platform one.*

We were not too concerned with resolving such interoperability problems in Sol, because it had already been solved inside the MalacaTiny agents. So, we developed an agent (ProxyAgent in Figure 6.2), which acts as a proxy that is able to interact simultaneously using two different agent platforms (Sol and Jade/Leap by instantiating two Representation aspects and two Distribution aspects (SolPlugin and JadePlugin respectively). Thanks to this proxy agent, it was possible to reuse the GroupOrganizer agent. Once the GroupOrganizer agent organizes the groups, it has to communicate them to each GuideAgent. But, this communication is performed through the proxy agent, which receives the ACL message from the GroupOrganizer Jade agent (through the JadePlugin) and sends it to the GuideAgent Self-StarMAS agent through the SolPlugin). Another possible solution, is to instantiate the JadePlugin as part of each GuideAgent. But, since we are trying to optimize the performance of agents in lightweight devices, we do not want to introduce unnecessary overhead. The proxy agent could be executed in a device with more capacity to instantiate many distribution aspects, maintaining good response

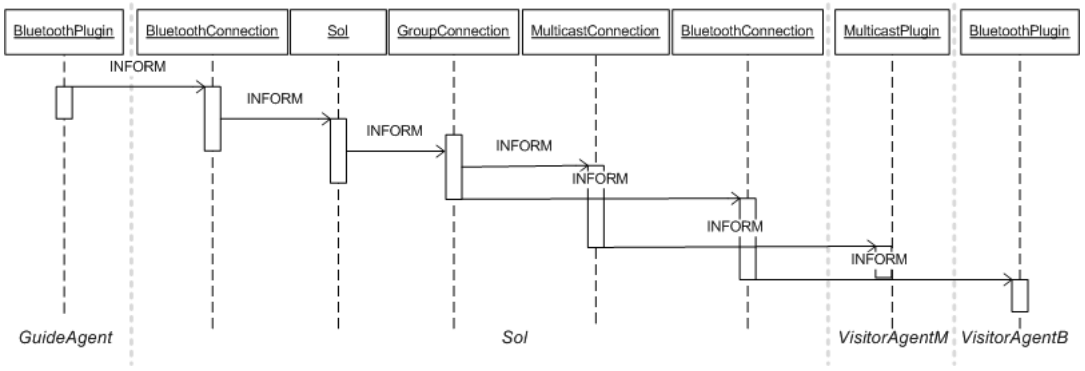

Figure 6.8: UML sequence diagram of sending a message to a group in *Sol*.

times.

6.3 Summary

In this Chapter we have presented our progressing in enabling the communication between MalacaTiny agents. Firstly, we have presented the internal design of the *Blue* agent platform, that enables Bluetooth-based communications between agents deployed in mobile phones with MIDP profiles. Secondly, we have presented the *Sol* agent platform that contributes to achieving at agent platform level some of the challenges raised in Chapter 1. *Sol* facilitates (1) the communication and interoperation of agents running in a heterogeneous set of devices (such as Sun SPOT sensor motes, Android-based lightweight devices and mobile phones) and it is even able to use different communication protocols; (2) group message delivery (one-to-many communication) efficiently. The *Sol* agent platform supports the native communication protocols of each device (e.g. ZigBee, WiFi) and acts as a gateway, performing specific functions in order to ensure interoperability.

Chapter 7

Validation

In order to evaluate the work presented in this thesis, we present and discuss some results with regard to the automatic generation process of agents for AmI systems and the energy consumption and performance of MalacaTiny. Additionally, we present the results of the self-management functionality and distribution aspects implemented for MalacaTiny; Jade-Leap, *Blue* and the *Sol* agent platform.

This chapter is structured as follows: Section 7.1 presents the results for the Degree of Automation of our MDD process; Section 7.2 presents the results of the power consumption, performance and scalability of MalacaTiny agents; Section 7.3 studies the benefits and performance of the self-management functionality; Section 7.4 studies the performance of the *Sol* agent platform; and Section 7.5 summarizes the contributions of this Chapter.

7.1 Degree of automation

For measuring Productivity, Automation and Reuse in systems based on *application frameworks* (such as MalacaTiny or Jade) we cannot use the Line Of Code metric (LOC) as it cannot be directly used to compare the size of the generated MAS. Instead we use an adaptation of the approach presented in [Harrington and Cahill, 2011], called degree of automation.

The effort (of development) required to implement a case study such as the IM is recorded using the number of elements (NE) a metric of the elements (concepts

from the metamodel) required for modeling the agents, coordination protocols, plans and self-management functionality of the case study with the Pineapple metamodel. Our experiments extend the IM by adding new agents with new capabilities, new states to protocols and plans that complete protocol specification and new rules to self-management policies. After each experiment is performed, the NE metric is calculated again for the extended model. The increased agent, coordination and self-management data produced by the transformation process is recorded and used as an alternative measure for the increased functionality in the extended scenario.

The ratio of the increased Pineapple model development effort can be compared to the ratio of the increased number of features in protocols and agents for MalacaTiny. These ratios indicate the expressive power of the Pineapple abstractions and their contribution to the transformation process to generate application functionality. This ratio is measured as:

$$\delta(Model)/\delta(Feature\_Size) \tag{7.1}$$

where

$$\delta(Model) = \frac{ExtendedPineappleModel(NE)}{OriginalPineappleModel(NE)} \tag{7.2}$$

and

$$\delta(Feature\_Size) = \frac{ExtendedFeatureSize}{OriginalFeatureSize} \tag{7.3}$$

The degree of automation metric is useful for measuring the related evaluation criteria of productivity gain, automation and reuse. The extension of agent specifications, protocols and self-management policies, and the re-running of the transformation process show the reuse of domain model components. Since we principally transform three elements (agents, protocols and self-management policies), we apply these metrics three times. On the one hand, for the transformation process that generates agents (knowledge, components, aspects and composition rules), the feature studied is the number of these elements. The NE measure is applied to the Pineapple model that contains these features (MAS, organization,

collaboration, agent). On the other hand, the second transformation process corresponds to the protocol description by means of a finite state machine for each participant role. In this case, the feature studied is the number of the transition rules; plans executed in these transitions (transition descriptions) and atomic actions contained in these plans. The NE measure is applied to the protocol concept and associated plan in the Pineapple model. Finally, for the transformation process of self-management policies, the feature studied are similar to the case of the coordination protocols. So, we study the number of transitions of the *SMMachine*, plans executed in these transitions and atomic actions. Additionally, we consider the knowledge and plans included in *SelfManagement* role capabilities. The NE measure is applied to the *SelfManagement* role concept (condition, triggers, plans and knowledge).

Scenario 1. Firstly, we add a new agent to our MAS for IM named *ExhibitAgent* that represents an exhibit in the museum and gives information about the item that it represents. In the current implementation of the museum, *VisitorAgent* queries an internal data base to find out what exhibits could be interesting to a each user. However, this solution is difficult to maintain because each time a new exhibit is added to the collection, the internal database of *VisitorAgents* must be updated. Our solution is the *ExhibitAgent*, these agents provide information of the items and are members of a group that is composed of the items that are located in the same room.

When the user enters a new room in the museum, its *VisitorAgent* automatically sends a message to the corresponding group requesting information on the exhibits and receives information about the items in order to make a recommendation. So, when a new item is added to the exhibition, we only need to program and instantiate a new *ExhibitAgent* while the *VisitorAgent* remains unchanged. This solution is advantageous when we want to add more information to an exhibit. In this case, we only need to update the *ExhibitAgent* that represents the exhibit. Adding the *ExhibitAgent* means adding the following elements: we add a new agent, which is a member of the *IntelligentMuseum* organization (Figure 3.3) through a new role named *Exhibit*; a component that stores the information of the exhibit that must be shared; a new protocol to model the interaction between *ExhibitAgent* and *VisitorAgent*; a plan for the visitor agent to model the sharing

of the information; we modify the plan to recommend items in a room. We recalculate the NE for the extended design model. The number of elements of agent features in the original model is 142, while the NE for the extended model is 166. The number of generated entities in the original model for these features is 55 and for the extended model is 65. Therefore, the value of the measure is:

$$\delta(Model) : \delta(Feature\_Size) = \frac{144}{132} : \frac{126}{112} \tag{7.4}$$

An increase of 9% in agent development effort results in a 13% increase in the application functionality (measured by the number of coordination aspects, default aspects, distribution and representation, composition rules, components and knowledge).

Scenario 2. In order to check the Degree of Automation in the coordination protocol transformation process, for the case study presented above, we transform the *SearchGuide* protocol, which is a classical FIPA-Request[1] protocol, into a FIPA-ContractNet[2] protocol (this protocol is similarly modeled in the Pineapple metamodel to the *RoutePlanning* protocol (see Figure 3.4), but we add new messages, messages flows and sub-actors associated with the actors). This protocol involves one initiator and a set of participants, so it is a more complex protocol, with more messages and states. The Initiator solicits an m proposal from other agents by issuing a call for proposals (cfp for short) communicative act, as well as any conditions the Initiator is placing upon the execution of the task (in our example the Initiator requires the room in the museum where the guide is).

In order to modify the *SearchGuide* protocol to follow the FIPA Contract Net protocol it is necessary to modify the corresponding protocol diagram and the associated plans. The number of entities obtained for these elements in the original model is 130 and for the extended model is 220. The number of generated entities in the original model for these features (coordination aspects, interchanged messages, plans and atomic actions in plans) is 203 and for the extended model is 358. Therefore, the value of the degree of automation is:

[1] http://www.fipa.org/specs/fipa00027/
[2] http://www.fipa.org/specs/fipa00029/

$$\delta(Model) : \delta(Feature\_Size) = \frac{226}{130} : \frac{371}{184} \tag{7.5}$$

This result shows that an increase of 73% in the protocol development effort results in an increase of 101% in agent functionality (in terms of the number of transition rules, transition descriptions and atomic actions generated). So, in this scenario, the percentage of generated model is 28%, showing that the grade of automation is high for complex change scenarios, justifying the use of an MDE approach.

Scenario 3. In order to check the Degree of Automation in the transformation of self-management policies, we extend the policies associated with *VisitorAgent*. In the original model, *VisitorAgent* has a policy that enables a cache for the contents of the museum when the response time of the application is low. In the extended model, *VisitorAgent* is aware of the power consumption and when the battery level is low disables the context-aware behavior that recommends exhibits when the user enters a new room (*LocationAwareRecommendation* in Figure 3.5).

In order to add this new self-management functionality, we modify the policy associated with the *ContentOptimizer SelfManagementRole* (see Figure 3.3). This policy is a *ComposedRule* (see Figure 3.7) that uses the *Parallel RuleOperation* and is composed of two *BasicRule* (for enabling the cache and for disabling the cache). Then, in the extended model, we add two extra *BasicRule* elements to enable or disable the *LocationAwareRecommendation*. At the PIM level, the number of elements that compose the *ContentOptimizer SelfManagementRole* is 22 and in the extended model 56. The number of generated entities in the original model for these self-management features (*SelfAdjusting*, *Knowledge* and *SMPlan*) is 46 and for the extended model 126. Therefore, the value of the measure is:

$$\delta(Model) : \delta(Feature\_Size) = \frac{56}{22} : \frac{146}{46} \tag{7.6}$$

This result shows that an increase of 154% in the protocol development effort results in an increase of 217% in agent functionality (in terms of the number of elements that compose the *SelfAdjusting* aspect, atomic actions included in *SMPlans* and knowledge associated with self-management). Therefore, in this last scenario, the percentage of the generated model is 63%.

7.2 Validation of MalacaTiny agents

This section presents some results showing the energy efficiency and performance of MalacaTiny agents for devices with profile MIDP and Android devices compared with Jade-Leap. Agents implemented in Java ME and in Android have been widely used to develop AmI systems [Bromuri et al., 2010; Lech and Wienhofen, 2005; Muldoon et al., 2006; Sánchez-Pi et al., 2008]. In our case study *GuideAgent*, *SecurityAgent* and *VisitorAgent* runs on the mobile phone of the users.

Although, the experiments for Java ME were performed in a 5630 XpressMusic [Developer, 2013b] and in a N96 [Developer, 2013a], in this section we present only those results obtained for XpressMusic. The mobile phone parameters have been monitored with the Nokia Energy Profiler. This application offers real time values collected while the mobile phone is functioning. The experiments for Android were performed in a HTC Desire [Encyclopedia, 2013] and in a Samsung Galaxy [Mobile, 2013], but in this section we only present the results obtained from the HTC Desire. However, the memory footprint of a process is difficult to measure in Android because a lot of memory is actually shared across multiple processes [Hackborn, 2010], so we only present static results captured in the Android emulator in order to provide a notion of the size of the agents.

7.2.1 Resource consumption

The goal of the experiments presented in this section is twofold: (1) To quantify the memory footprint and power consumption of MalacaTiny agents in MIDP and Android versions and using (for all of them) two different communication mechanisms, i.e. a Jade-Leap agent platform and Bluetooth (we will call them JL-MalacaTiny and B-MalacaTiny, respectively); and (2) to compare them, in terms of memory usage and power consumption, with a Jade-Leap version of the same agent. We have made this comparison using Jade-Leap because, as can be seen in Table 2.2, it is the only FIPA-compliant agent technology that supports both Android and MIDP devices. Memory usage (in KiloBytes) is measured both in MIDP and Android versions. Power consumption is only evaluated in milliamperes (mA) in the MIDP versions using the Nokia Energy Profiler, since the Android

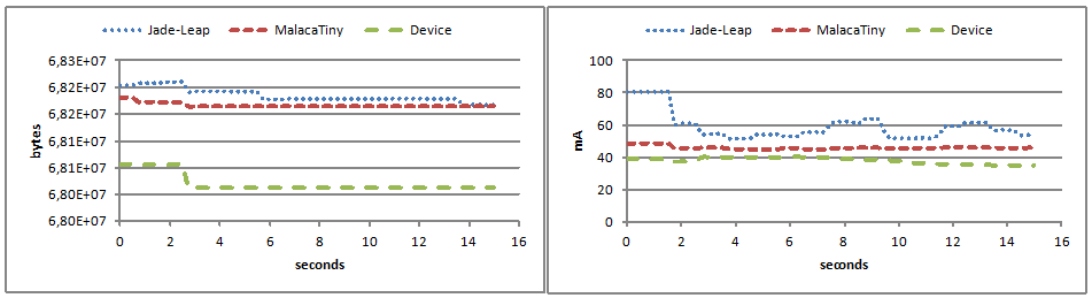

Figure 7.1: Memory occupation (left) and power consumption (right) averages for *IdleAgent*.

emulator does not provide these data. In order to measure the resource consumption of MalacaTiny agents and perform the comparison with the corresponding Jade-Leap agents, we have conducted a battery of tests based on different executions of the JL-MalacaTiny, the B-MalacaTiny and Jade-Leap agents in MIDP and Android mobile phones. For each MIDP agent (in Jade-Leap, JL-MalacaTiny and B-MalacaTiny), each test took fifteen seconds and was repeated ten times. The average of the results for each test is shown graphically. For Android agents, the tests for memory footprint (in KiloBytes) were performed in the emulator and the results shown are the average of the result obtained after ten executions. Table 7.1 summarizes the results of the different tests for Android agents.

Firstly, we considered memory and power consumption for idle agents (i.e without performing any action). Therefore, we monitored mobile phone resources while an idle agent was running (*IdleAgent*) and then we compared these results with the mobile resources consumption without agents. This test was carried out for JL-MalacaTiny and Jade-Leap agents. The results are shown in Figure 7.1. As expected, resource consumption is lowest when there is no agent running. However, the MalacaTiny agent resource consumption is lower than the resource consumption exhibited by the Jade-Leap agent. So, the MalacaTiny implementation is more efficient in terms of memory and power consumption than Jade-Leap. In MalacaTiny we have optimized the resources by enabling and disabling aspects, depending on the agent's needs. The same test was performed in Android phones with similar results (as can be seen in Table 7.1, row labeled *IdleAgents*), i.e. the

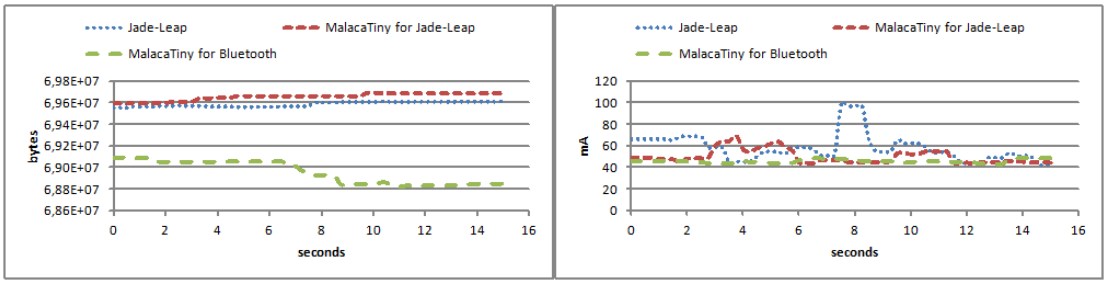

Figure 7.2: Memory occupation (left) and power consumption (right) for *ChattyAgents* during reception.

memory size of the Jade-Leap agent is 3915kB while the memory required by a MalacaTiny agent is lower, 3290kB.

Memory and power consumption have also been measured during agent interaction. Some tests were performed where multiple messages were sent or received. For this experiment, two paired agents were developed: an agent which waited ten seconds and sent ten messages, and an agent that waited for these messages. These agents are named *ChattyAgents*. Figures 7.2 and 7.3 show the memory occupation and power consumption results for the time slot in which a message is sent or received. For message reception (Figure 7.2), Jade Leap has good results for memory occupation (although higher than B-MalacaTiny), but the worst results for power consumption. For MalacaTiny, the memory occupation is the highest for JL-MalacaTiny, but the lowest in the B-MalacaTiny implementation. Regarding power consumption, MalacaTiny is the most energy efficient with the lowest power consumption in JL- and B-MalacaTiny versions, something so important in AmI systems as it means it is possible to extend the lifetime of devices. Again, the B-MalacaTiny agent had the best results for message sending in memory and power consumption tests (Figure 7.3). The difference is notable for power consumption with an average of 64'8mA, while the average of Jade-Leap is 116'4mA and for JL-MalacaTiny is 121'9mA. The second best results for both measures are Jade-Leap but they are quite similar to those of JL-MalacaTiny. Different implementations of the *ChattyAgent* agents are also executed on Android mobile phones. Memory footprint results are given in Table 7.1 (row labeled *ChattyAgents*). For reception, Jade-Leap has a memory footprint of 4721kB and MalacaTiny of 4780kB. For

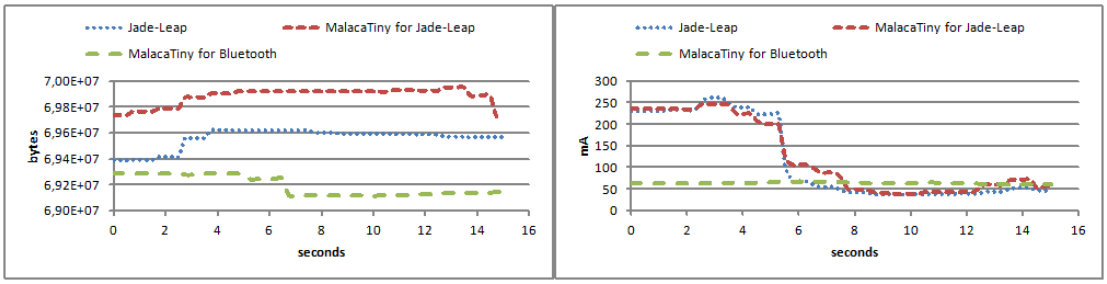

Figure 7.3: Memory occupation (left) and power consumption (right) for *ChattyAgents* during sending.

Table 7.1: Memory-footprint of Jade-Leap and MalacaTiny agents in Android devices for resource consumption tests in KiloBytes.

	Jade-Leap		*MalacaTiny*	
	Reception	*Sending*	*Reception*	*Sending*
IdleAgent	3915		3290	
ChattyAgent	4721	4798	4780	4925

sending, the results are similar and Jade-Leap achieves 4798kB, while MalacaTiny 4925kB.

Finally, to summarize, the conclusions of these tests are: (1) MalacaTiny does not introduce a critical overhead when it is used on top of an agent platform, and therefore developers can benefit from the advantages offered by the MalacaTiny approach at design and implementation with a minimum resource penalty, and (2) the implementation of B-MalacaTiny is the one which better scales in memory and power consumption, showing that it is worth having agents with the capacity of interacting through different communication protocols. This offers the possibility of choosing the optimal communication protocol, taking into account the expected resource and power consumption.

7.2.2 Scalability

In this section we include some of the tests that show how MalacaTiny for MIDP scales with respect to the number of *aspects* instantiated by the agent. The results

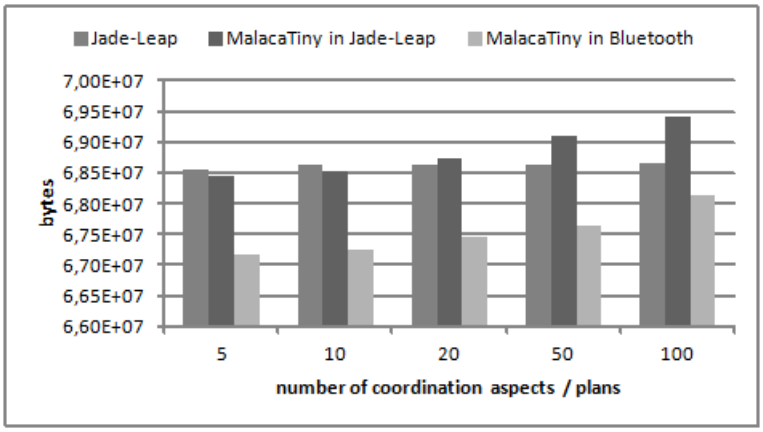

Figure 7.4: Memory occupation with different numbers of coordination aspects.

of the tests for Android agents are not shown since they are not very reliable because the memory between Android processes is mostly shared and it is difficult to measure the real memory footprint of a process. We chose to measure *aspects* since the functionality in MalacaTiny is encapsulated in components implemented by classes, as with Jade-leap. Furthermore in MalacaTiny each agent conversation implies the instantiation of a coordination aspect. If we measure how well MalacaTiny scales with regard to the number of aspects, implicitly we are measuring how well it scales regarding the active behaviors of the agent. The tests measure the increment of the agents memory footprint when the number of active aspects for MalacaTiny or behaviors for Jade-Leap increases. We measured the average memory footprint for fifteen seconds with a different number of coordination aspects or conversations for Jade-leap (5, 10, 20, 100), repeating each test ten times and calculating the average of these results (Figure 7.4).

As the results show, B-MalacaTiny again scales better than the other agent implementations. For a low number of active aspects, results are very similar for Jade-Leap and JL-MalacaTiny but for a high number of aspects Jade-Leap performs much better.

Table 7.2: Average and standard deviation for message sending and reception in MIDP devices in milliseconds.

	Jade-Leap	*JL-MalacaTiny*	*B-MalacaTiny*
Average	139.4	176.4	257.8
Standard deviation	39.8	68.1	9.6

Table 7.3: Average and standard deviation for message sending and reception in Android devices in milliseconds.

	Jade-Leap	*JL-MalacaTiny*
Average	411.3	360.2
Standard deviation	217.64	180.01

7.2.3 Performance

In this section the performance of MalacaTiny agents is evaluated and compared with Jade-Leap agents. The evaluation and comparison are presented in Tables 7.2 through 7.5. The numbers given correspond to the average and the standard deviation of the results that come from the multiple execution of each experiment. Our first experiment measured the time required for sending and receiving a message (round-trip delay time). Two paired agents were implemented: an agent that sends a message and waits for a response, and an agent which receives a message and sends a reply. These agents were implemented using Jade-Leap and MalacaTiny for MIDP and Android. MalacaTiny agents have been executed using Jade-Leap and Bluetooth. Each experiment was repeated thirty times. Table 7.2 shows the results for the agents implemented in MIDP. As can be seen, the results for Jade-Leap agent and the MalacaTiny agent deployed in this agent platform are quite similar (a difference of just 37 milliseconds). The result for the agent using Bluetooth shows that this is the slowest communication system, as expected. The results for agents implemented for Android (showed in Table 7.3) are different. In this experiment, the times measured are a little better for MalacaTiny (51.1ms).

Table 7.4 shows the latency of an interaction following the FIPA-Query pro-

Table 7.4: Average and standard deviation for FIPA-Query protocol execution in MIDP devices in milliseconds.

	Jade-Leap	*JL-MalacaTiny*	*B-MalacaTiny*
Average	676.7	530.5	22418.6
Standard deviation	39.1	76.8	18045.9

Table 7.5: Average and standard deviation for FIPA-Query protocol execution in Android devices in milliseconds.

	Jade-Leap	*JL-MalacaTiny*
Average	262.3	372.73
Standard deviation	142.1	292.06

tocol. The result includes message delivery, internal message processing and the realization of a simple activity. In the implementation of the FIPA-Query protocol used in our experiment, the initiator requests the participant to perform some kind of informing action by sending a query-if message. Once the participant receives and processes the query-if message, it decides whether to accept or refuse the query request (in this implementation the query is always accepted) and sends the corresponding message to the initiator.

In Table 7.4, we can see the average and standard deviation of ten executions of the agents implemented in Jade-Leap, JL-MalacaTiny agent and the B-MalacaTiny agent for MIDP. The highest result is achieved for the B-MalacaTiny agent (as expected). The reason is that both service discovery and message delivery (see Table 7.2) is slower in agents using Bluetooth than in the agents using the Jade-Leap agent platform with MIDP. In addition, we want to highlight that JL-MalacaTiny exhibits the lowest result (despite message delivering being slightly slower in JL-MalacaTiny agent). Finally, as in the previous experiment, the results are different for the Android versions of the agents, being the performance of the Jade-Leap agent for Android better than MalacaTiny agents for Android. From these results we can conclude that the internal design of MalacaTiny agents does not introduce a critical overhead and the differences in the performance are not notable.

7.3 Performance of self-management functionality

In this section we are going to show the results of the different tests that have been carried out in order to validate the performance and efficiency of the self-configuring capacity inside the agent. Due to the fact the response time is a major concern in AmI systems, these tests focus on the times required to accomplish the different tasks of the self-configuring loop inside MalacaTiny agents. All the experiments has been repeated fifty times and the average and the standard deviation have been calculated. The tests have been performed in HTC Desire mobile phones and Sun SPOT sensors.

The current implementation of our agents is able to do the following self-configuring tasks: (T1) to change the sampling frequency of a monitoring component/aspect; (T2) to change the distribution aspect at runtime; and (T3) to require a new source of data in case of failure of the data provider. The following experiments focus on measuring the time that is required for the different self-configuring tasks, which is, in our agent architecture, independent of the specific application scenario (medication prompting, health monitoring, location based services, etc.). This is because the application/system in which agents are deployed only acts as a trigger for the different self-configuring tasks.

In order to measure T1, we use a *SensorAgent* that when it detects its battery level is low, changes the sampling time of an internal monitoring service (e.g. temperature monitoring). This task is useful because it can contribute to increasing the lifetime of the sensor. The time required for T1 in a *SensorAgent* is 1219 milliseconds with an standard deviation of 1 millisecond. The accomplishment of this task requires localizing the monitoring aspect and calling the method that changes the temperature sampling in the hardware device. This self-management policy can increase the life of a Sun SPOT sensor mote by 4%.

Secondly, in order to measure the time required for T2, we focus on a *GuideAgent* that changes its distribution aspect at runtime from the Sol agent platform to Jade-Leap. This task only requires 316 milliseconds with an standard deviation of 66 milliseconds. This can be considered the reference time for substituting an aspect with an alternative implementation.

The substitution of the distribution aspect can have great benefits in power saving if we change between a distribution aspect based on WiFi (e.g. Jade-Leap) and another based on Bluetooth (e.g. *Blue*). The platform over Bluetooth is slower than Jade-Leap, but is much more energy efficient. So, we define a reconfiguration to replace the Jade-Leap with the Bluetooth communication, in the mobile device used in our case study. This device can manage the reconfiguration of many devices, thereby minimizing the energy expenditure during the message delivery and reception which is useful. In this example the agent being executed in the mobile device uses a FIPA Request protocol[1] to supervise the replacement of the agent platform in all the devices managed by this agent. In our experiment we have five agents managed by the supervisor agent. Considering that the cost of sending a message with Jade-leap is 121.91mA and of receiving a message is 50mA (see Subsection 7.2.1), the total cost of reconfiguration is 862.3mA. On the other hand, the cost of sending a message via Bluetooth is 64.798mA and of receiving a message is 45.93mA. Let us consider that the master agent sends and receives 50 messages (for example, with sensed data) per hour, using Jade-Leap we would expend 8550mA, but with Bluetooth this cost is reduced to 5536.4mA. Adding to the Bluetooth cost the self-configuring cost, this means that in just one hour we have saved 28.79% in energy. Over the following hours the cost of reconfiguration is no longer present so the energy saving increases by up to 64.65%, in all the devices (five in our experiment), increasing the lifetime of both the AmI system and the batteries.

Finally, in order to measure T3 we have deployed different environmental monitoring systems, principally composed of one *SecurityAgent* (executing in the smart phone of a member of the security staff) and a variable number of *SensorAgents* agents (used to monitor different elements of the same room in the museum). The number of sensor agents can vary from just one sensor (for instance providing acceleration data) to various. All the *SensorAgents* of the system are registered as service providers of the specific service which fails and causes the T3 reconfiguration. With these experiments we measure both the time required for T3 and the scalability of this self-configuring task. This reconfiguration task is the most complex one, implying several sub-task detailed below. When a *SecurityAgent* is

[1] `http://fipa.org/specs/fipa00026/SC00026H.html`

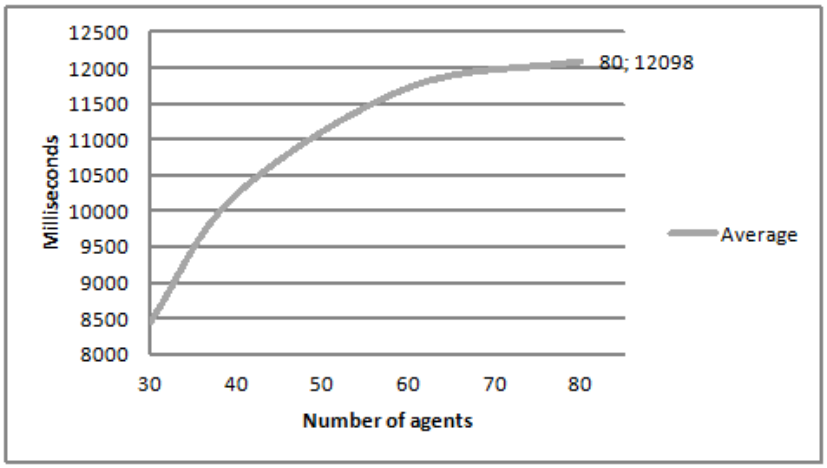

Figure 7.5: Times (in milliseconds) for T3 for different numbers of agents.

waiting for a service from a *SensorAgent* and this service fails, the *SecurityAgent* queries the directory facilitator of the Sol agent platform (see Chapter 6) in order to get a new service provider. Then, it sends request messages to all the *SensorAgents* that can provide the required service and chooses one, sending a *confirm* message, discarding the others, sending a *refuse* message. The results of this experiment (see Figure 7.5) shows that time for self-configuring scales up with the number of agents, following a logarithmic curve. The resulting times are affordable and the scalability of the self-configuring task is good. Note that in an extreme situation, when the number of sensors-and agents is increased by more than 250%, the time required for self-managing increases by 140%.

However, note that T3 requires more time than the other self-configuring tasks, it being the worst case. This is mainly because the time to accomplish T3 is affected by having to exchange a lot of messages. The exchange of messages is usually the most time consuming task in any networked system. In the following section, we will show the benefit of the group mechanism of Sol for this experiment.

The conclusion to be drawn from these experiments is that our system has a reasonable response time for self-management.

7.4 Performance of the *Sol* agent platform

In this section, we validate how our platform accomplishes the challenges presented in Chapter 1. We have illustrated how *Sol* supports device and communication heterogeneity (Challenges C2.1-*Manage device and agent platform heterogeneity* and C2.2-*Cope with wireless network diversity*). Therefore in Subsection 7.4.1, we show the performance of *Sol* when it enables communication between heterogenous devices. Additionally, we illustrate the accomplishment of the Challenge C2.1, showing the performance of the *ProxyAgent* presented in Subsection 6.2.2.2. In Subsection 7.4.2, we validate the group communication showing its benefits for agents running in Sun SPOT sensor motes and Android. Experiments in this section for Android devices have been carried out using the Goal Oriented version of MalacaTiny. We also provide an idea of the size and performance of these agents. All the experiments presented in this section have been repeated fifty times and the average and standard deviation have been calculated. The time is measured in milliseconds (ms) and to accomplish these experiments we have used Android Google Nexus [Inc., 2013] devices.

7.4.1 Interoperation between heterogeneous agents

In this section, we show the times for communication between the devices that can compose the Intelligent Museum and that communicate through the *Sol* agent platform. Specifically, we measured the performance in the communication between an agent that uses WiFi and another agent that uses Bluetooth and an agent that uses WiFi and another agent that uses ZigBee. Firstly, we focus on the case of Bluetooth and WiFi, e.g. the communication between a *GuideAgent* and a *VisitorAgent* where the first one can use Bluetooth or WiFi to exchange messages. In this experiment, we measured the round-trip delay time of a message sent by the *GuideAgent* to a *VisitorAgent*. The goal of these experiments is twofold: (1) to show that the time spent in communicating through *Sol* is reasonable and (2) to show that the communication through the *Sol* agent platform does not introduce a critical overhead. The round trip delay time for a *GuideAgent* and *VisitorAgent* using only Bluetooth and exchanging messages of 207 Bytes size is 413 ms with a

standard deviation of 157 ms. In this situation, the communication requires two RFCOMM sockets using the same network technology (Bluetooth). On the other hand, the result for communicating these agents using the *GuideAgent* WiFi is 647 ms with an standard deviation of 115 ms. The round trip delay time for two agents exchanging messages of 207 Bytes size and using WiFi is 823 ms with a standard deviation of 136 ms. In this situation, the communication requires two TCP connections using the same network technology. On the other hand, the result for communicating one agent that uses WiFi (e.g. *SecurityAgent*) and a *SensorAgent* is 619 ms with an standard deviation of 60 ms. This situation involves the use of two transport protocols (UDP and TCP). These results show that the round trip times in these scenarios are reasonable for communicating smart phones and sensors making up the IM. We also show that the communication between mobile phones and sensors through the *Sol* agent platform using different transport protocols does not introduce a critical overhead.

An additional problem that we have found is that interoperability is not possible in all cases (for instance, sensors use IEEE 802.15.4 standards to communicate, which are still not supported by personal lightweight devices). In order to achieve interoperability we need additional solutions and one of them is using proxies. In this situation our approach can enable the interoperation by means of designing an agent to act as a proxy (see Section 6.7). This is invaluable if we consider the development of specific purpose agents and their subsequent integration with a MAS composed of general purpose agents such as Self-StarMAS.

In the following experiment, we measure the performance of an MalacaTiny agent acting as a proxy between *Sol* and Jade-Leap agent platforms as depicted in Scenario 7. As stated before, the experimental scenario (see Figure 7.6) has 3 agents, one deployed in Jade-Leap agent platform (*GroupOrganizer* agent), another in *Sol* agent platform (*GuideAgent* agent) and a third agent which is deployed in both the *Sol* and Jade-Leap agent platform. This agent, which acts as a gateway and is called *ProxyAgent*, is a MalacaTiny agent that has instantiated two different distribution aspects (*JadePlugin* and *SolPlugin*). The behavior of this agent is very simple; each message it receives from the *GroupOrganizer* agent through the *JadePlugin* is processed until an internal representation of the ACL message has been obtained and forwarded through the *SolPlugin* to the *Guide-*

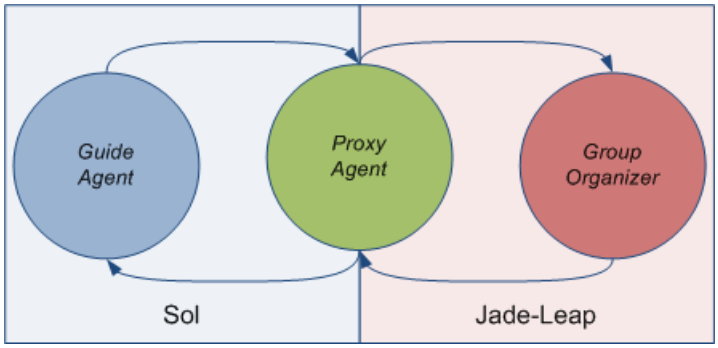

Figure 7.6: MalacaTiny agent acting as a proxy between agents deployed in *Sol* y Jade-Leap agent platforms.

Agent. The reverse is true, each message it receives from *GuideAgent* through the *SolPlugin* is processed until an internal representation of the ACL message has been obtained and forwarded through the *SolPlugin* to the *GroupOrganizer* agent. As we stated in Scenario 7 it has the possibility to operate in both agent platforms simultaneously. In the experiment, we measured the round-trip delay time of a message that is sent by the *GuideAgent*, is forwarded by *ProxyAgent* and received by *GroupOrganizer*. To compare the work of *ProxyAgent* acting as proxy, we developed 2 additional scenarios with the same schema of communication, but the final agents (i.e. *GroupOrganizer* and *GuideAgent* agents) are deployed in the same agent platform (i.e. Jade-Leap or *Sol*). Despite this, messages pass through the *ProxyAgent*. Additionally, we measured the heap memory of the *ProxyAgent* in the three scenarios.

Table 7.6 shows the results of this experiment, the standard deviation is around 200 ms for the 3 scenarios considered. As you can see regarding the delay of message delivery, the worst result is for the real proxy (row "Sol+Jade-Leap"), while memory usage does not increase significantly due to the translation process (the differences in the heap memory of the three proxy agents are not notable). These results were expected, because of the additional coding and decoding from one agent platform message transport format to another required by the inter-platform communication. However, we did not expect the difference between the only-*Sol* and only-Jade-Leap experiments. In order to detect why *Sol* is almost double, the

Table 7.6: Average round-trip delay time (in milliseconds), heap memory (in MegaBytes) for the proxy experiment.

Target Platforms	*Time (ms)*	*Memory (MB)*
Sol+Jade-Leap	2031	11.250
Sol	1490	11.133
Jade-Leap	866	11.07

profiling tool of the Android DDMS[1] was used to analyze and determine the source of such an overhead, finding that the time-consuming task was the *Representation* aspect behavior (see Section 6.2.2.2) that deals with the codification of transport messages of the *Sol* platform. The problem is that *Sol* agent platform uses a common message format based on String (to ensure the interoperability between its multiple, supported communication mechanisms), and MalacaTiny agents format messages prior to their delivery, while Jade-Leap agents directly use object serialization to transport ACL messages. Therefore, the work of the *Representation* aspect when it has to translate a *SolMessage* is a time-consuming task (Jade-Leap agents serialize the internal representation of ACL messages). In the future, we plan to optimize the formatting of *SolMessage* in order to enhance communication times.

7.4.2 Group communication

As stated before, MalacaTiny agents can accomplish different tasks for self-configuring. The T3 task, requires a new service data provider of data in case of failure of the service provider and can greatly benefit from *Sol*. In Section 7.3, we evaluate this task considering that the agent, each time it loses the service provider, queries the directory facilitator of *Sol* in order to get a new service provider. Then, it sends request messages to all the service provider agents that can provide the required service and chooses one, sending a confirmation message and discarding the others, sending a refuse message. In this section, the set of service provider

[1] `http://developer.android.com/tools/debugging/debugging-tracing.html`

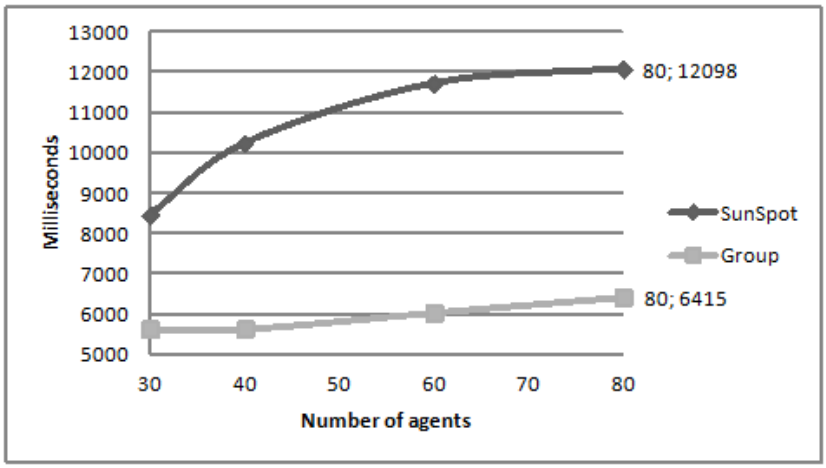

Figure 7.7: Times (in milliseconds) for the T3 task for different numbers of agents.

agents are replaced by a group (*GroupConnection*, see Subsection 6.2.1.2) and we can see the benefits of the group mechanism by comparing it to the classical unicast messaging (*TCPConnection*, *SunSpotConnection* or *BluetoothConnection*). Specifically, we compare one *GroupConnection* with a set of *SunSpotConnections* to see the benefits for the T3 self-configuring task.

In order to evaluate T3 with the two communication mechanisms (a single *GroupConnection* and a set of *SunSpotConnections*), we repeat the experiment presented in Subsection 7.3, but this time replacing the set of *SunSpotConnection* with a single *GroupConnection*. As in the previous experiment, the number of *SensorAgents* can vary from just one sensor (for instance providing luminosity data) to several. All the *SensorAgents* of the system are registered as service providers of the specific service which fails and causes the T3 reconfiguration. In Figure 7.7, we can see the mean of the results for the two experiments. Although the times obtained are affordable and the scalability of the self-configuring task using the set of *SunSpotConnection* is good (*SunSpot* label), the benefit of using the mechanism for group-based communication (*Group* label) is clear. The time when using the *GroupConnection* is lower in all scenarios. Additionally, note that when the number of sensors and agents increases by more than 250%, the time required for self-managing increases 140% in the case of the set of *SunSpotConnections* and 120% in the case of the *GroupConnection*.

The group mechanism offers a great advantage when communicating a dynamic MAS whose agents can vary at runtime. In our case study, the *GuideAgent* can send information to its group of visitors without it being necessary to know what the identifiers of the agents that compose the group are, or when the members join or leave the group. Additionally, it is not necessary for the agent to control the group member's presence in the MAS. We validated the performance of the group mechanism for Android and compared it with unicast standard communication. The experiment consists of a group composed of 20 *VisitorAgents* (17 of them running in virtual devices and 3 agents in real devices) and a *GuideAgent*. We have measured the round-trip-delay time for a message sent by the *GuideAgent* using 3 mechanisms: (1) a single *GroupConnection*; (2) sending a unique message including 20 target agents as in the receivers field (the identifiers of the members of the group); and (3) sending 20 different messages (with the same content data) from the *GuideAgent* to each member of the group. The difference between mechanisms (2) and (3) is that in (2) *Sol* only receives a message and then it sends the message through each *TCPConnection* associated with an agent depicted in the receiver field of the message (1+20 sendings in total), while in mechanism (3) *GuideAgent* sends 20 different messages to *Sol* that only have one agent depicted in their receiver fields (20+20 sendings in total). In (1) the number of sendings is just 2, because the sending of individual messages to the group members is left to the multicast facility of the network level (i.e. IP). The average and standard deviation results in ms are 935 and 93 for (1), 1023 and 214 for (2), and 1340 and 312 for (3). These results are slightly better for the *GroupConnection*, but as stated before, the advantage of using it is not only an improvement in performance. The results for (2) and (3) show that there is a penalty in the performance if the agent sends messages with a single receiver. This is because in MalacaTiny agent aspects are composed each time a message is sent or received, or when an event is thrown. So in (3) aspects are composed 20 times (for each member of the group of visitors), while in (2) only once. However, this overhead is easily avoided by sending messages with multiple receivers such as in (2).

7.5 Summary

In this chapter we have presented the validation of our approach, taking into consideration the M2M generation process, the MalacaTiny agents and the self-management functionality. We have evaluated the convenience of applying a model driven approach by assessing the Degree of Automation of the MDD process presented in Chapter 4. The results have shown that it is possible to automatically generate more or less 40% of the model in complex AmI systems. We have presented and discussed an evaluation of the MalacaTiny implementation for MIDP devices and Android enabled phones by assessing different parameters: performance, memory consumption and energy efficiency comparing Jade-Leap and using different communication protocols. The results have demonstrated that the internal design of MalacaTiny is very efficient, so using our framework even over another agent platform like Jade-Leap has very little or no penalty in resource consumption. We have shown the feasibility of our self-management functionality in terms of the response time of reconfiguration. Finally, we have presented an evaluation of the *Sol* agent platform. This platform has a reasonable response time in terms of wireless data exchange. Additionally, *Sol* enhances the self-configuration of agents thanks to the group communication.

Chapter 8

Conclusions

This chapter summarizes the proposal that has been explained throughout this dissertation, highlighting the contributions of our work, in Section 8.1. Then, in Section 8.2, we detail the main publications obtained from these contributions. A section discussing lessons learned is also provided (Section 8.3). Finally, we outline our prospective future work in Section 8.4.

8.1 Summary and conclusions

After the Introduction (Chapter 1) and Background (Chapter 2), this thesis presents an MDD process to automatically generate agents-based systems using the MalacaTiny platform-neutral framework, that provides agent technologies especially suitable for AmI systems. The model driven solution proposed, covers the design and implementation phases by the transformation of a design model of the AmI system in Pineapple. This is a general purpose agent metamodel that we adapted to support an explicit modeling of context aware systems and the modeling of the self-management functionality. Pineapple models are transformed into a set of self-managed MalacaTiny agents, able to be executed in heterogeneous lightweight devices.

We have defined Pineapple, a PIM to design self-managed AmI systems based on agents (Chapter 3). The foundation of Pineapple is the PIM4Agents metamodel, that we have adapted to support the modeling of context-awareness and the specification of policies for self-management. We have defined a new modeling

viewpoint called *Self-Management*, which permits the roles for self-management to be modeled (including the knowledge related with self-management) and the policies that drive the self-managed behavior of the AmI system. Policies are described using a domain specific language that follows the APPEL syntax. The use of APPEL enables conflict between policies to be detected using the UMC model checker. Finally, the new viewpoint also includes specific actions for modeling the autonomic functions that a self-managed system should support: self-awareness, self-situation; self-monitoring and self-adjusting.

We have defined the MalacaTiny metamodel, the PSM of the MalacaTiny agents (Chapter 4). This metamodel translates the advantages of the Malaca agent architecture to the metamodel level. The main feature of the internal architecture of a MalacaTiny agent is that it represents separately, application-specific functions from extra-functional agent properties. This separation improves the internal modularization of the agent architecture, which is based on the composition of components and aspects, and enhances the adaptation, reuse and maintenance of the software agent. Additionally, aspects encapsulate agent platform dependant functionality that makes MalacaTiny platform-neutral for FIPA compliant agent platforms.

We have defined an MDD process from the Pineapple metamodel to a set of MalacaTiny agents, able to be executed in heterogeneous lightweight devices (Chapters 4 and 5). With MalacaTiny at modeling and upon implementation, the model derivation process is simplified. In a typical MDD process, in order to add a new agent technology, it is necessary to implement a new model-to-model transformation process and a new model-to-text transformation process. In contrast, in MalacaTiny, this only implies the implementation of a specific plug-in, that is included in the MalacaTiny agent at deployment.

We have presented the internal design of the MalacaTiny agents and the goal oriented MalacaTiny (Chapter 5). MalacaTiny is a family of lightweight implementations of the Malaca agent architectural model. These agents are able to be executed in Sun SPOT sensor motes, mobile phones with MIDP profile and Android enabled devices. Additionally, they can use different mechanisms for communication, that include the Jade-Leap agent platform, a mechanism based on Bluetooth called the *Blue* agent platform or our platform to communicate he-

terogeneous agents, *Sol*. Goal Oriented MalacaTiny is a more advanced version of these agents, specifically for Android devices, that is goal oriented and has a dynamic aspect weaving process.

We have presented our implementation for the communication concern of MalacaTiny agents (Chapter 6). We have implemented a communication mechanism based on Bluetooth to directly communicate devices with MIDP profile. Additionally, in order to provide a workable solution to implement AmI applications based on agents, we have developed the *Sol* agent platform. *Sol* is a FIPA compliant agent platform, which can communicate agents from heterogeneous devices (Sun SPOT sensor motes, Android devices and mobile phones with MIDP profile) using heterogenous communication means (ZigBee, Bluetooth and WiFi). This platform acts as a gateway that facilitates interoperation between agents running in different and interoperable devices. Additionally, it provides services for the distribution of information to a set of related nodes. Many of the applications and services deployed in AmI environments require the dissemination of data to a set of group-related nodes.

Self-management is a concern of great importance in AmI environments. In this thesis, we have provided a framework to model self-management policies using the APPEL notation, to validate such policies using the UMC model checker and to configure the agent at runtime (Chapters 3, 4 and 5).

Finally, in Chapter 7 we have shown the evaluation of the different implementations that we have developed for this thesis. We have evaluated the convenience of applying a model driven approach by assessing the degree of automation of the proposed MDD process. The results show that it is possible to automatically generate more or less 40% of the code in complex AmI systems. We have presented and discussed an evaluation of the MalacaTiny implementation for MIDP devices and Android enabled phones using Jade-Leap and the *Blue* agent platform by assessing different parameters: performance, memory consumption and energy efficiency compared with Jade-Leap agents and using different interaction protocols. The results have demonstrated that the internal design of MalacaTiny is very efficient in terms of performance, memory occupation and energy efficiency, so using our framework even over another agent platform like Jade-leap has very little or no penalty in resource consumption. In particular, with the use of Bluetooth-

based communication we obtained very good results: MalacaTiny/Bluetooth is the combination that scales the best in terms of memory and power consumption, showing it is worth having agents with the capability to act through different communication protocols and technologies. Additionally, we have had to carry out other experiments to show that the self-management process implementation of MalacaTiny agents is also feasible. Chapter 7 concludes with the evaluation of our agent platform for AmI environments, the *Sol* agent platform. We have shown the feasibility and advantages of this agent platform in terms of the wireless data exchange, so important in the AmI.

Summarized, the main contributions of our work are as follows:

1. The Pineapple metamodel to model AmI systems based on agents with self-management capacities.
2. The MalacaTiny metamodel, an agent metamodel for platform neutral agents based on aspects.
3. Model Driven Development process from Pineapple to MalacaTiny agents.
4. Implementation and evaluation of the family of MalacaTiny agents for mobile phones with MIDP profile, Android and Sun SPOT sensor motes.
5. The goal oriented version of MalacaTiny for Android devices.
6. The *Sol* Agent platform to communicate heterogeneous agents using heterogenous communication means.
7. Implementation and evaluation of distribution aspects for Jade-Leap, *Blue* and the *Sol* agent platform.
8. Self-management modeling and functionality integrated inside the agent using aspect-orientation.

8.2 Publications

The results of the work involved in this research have been published in international journals, conferences and workshops, with peer-review, as follows:

Journals

- Inmaculada Ayala, Mercedes Amor, and Lidia Fuentes. A model driven engineering process of platform neutral agents for ambient intelligence devices. *Autonomous Agents and Multi-Agent Systems*, pages 1-42, 2013.

- Inmaculada Ayala, Mercedes Amor, and Lidia Fuentes. Self-configuring agents for ambient assisted living applications. *Personal and Ubiquitous Computing*, pages 1-11, 2012.

Conferences

- Inmaculada Ayala, MercedesAmor Pinilla, and Lidia Fuentes. Enhancing agent metamodels with self-management for ami environments. In *Progress in Artificial Intelligence volume 8154 of Lecture Notes in Computer Science*, pages 420-431. Springer Berlin Heidelberg, 2013.

- Inmaculada Ayala, Mercedes Amor and Lidia Fuentes. Exploiting dynamic weaving for self-managed agents in the IoT. In Ingo J. Timm and Christian Guttmann (editors) *Multiagent System Technologies, volume 7598 of Lecture Notes in Computer Science*, pages 5-14. Springer Berlin Heidelberg, 2012.

- Inmaculada Ayala, Mercedes Amor, and Lidia Fuentes. Self-starmas: A multi-agent system for the self-management of AAL applications. In *Proceedings of the 2012 Sixth International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing, IMIS'12*, pages 901-906, Washington, DC, USA, 2012. IEEE Computer Society.

- Inmaculada Ayala, Mercedes Amor, and Lidia Fuentes. Self-management of ambient intelligence systems: a pure agent-based approach. In *Proceedings of the 11th International Conference on Autonomous Agents and Multiagent Systems-Volume 3, AAMAS'12*, pages 1427-1428, Richland, SC, 2012. International Foundation for Autonomous Agents and Multiagent Systems.

- Inmaculada Ayala, Mercedes Amor, and Lidia Fuentes. Autonomic agents for mobile ambient assisted living applications. In José Bravo; Diego López

de Piña; Sergio Ochoa y Jesús Favela (editors). In *Proceedings of the 5th International symposium on ubiquitous computing and ambient intelligence. UCAmI 2011.* Riviera Maya, Mexico, december 5-9, December 2011.

- Inmaculada Ayala, MercedesAmor Pinilla, and Lidia Fuentes. Modeling context-awareness in agents for ambient intelligence: An aspect-oriented approach. In Luis Antunes and H. Sofia Pinto (editors). *Progress in Artificial Intelligence, volume 7026 of Lecture Notes in Computer Science*, pages 29-43. Springer Berlin Heidelberg, 2011.

- Inmaculada Ayala, Mercedes Amor, and Lidia Fuentes. A model driven development of platform-neutral agents. In Jürgen Dix and Cees Witteveen (editors). *Multiagent System Technologies, volume 6251 of Lecture Notes in Computer Science*, pages 3-14. Springer Berlin Heidelberg, 2010.

- Mercedes Amor, Inmaculada Ayala, and Lidia Fuentes. A4VANET: Context aware Jade-Leap agents for vanets. In Yves Demazeau, Frank Dignum, Juan M. Corchado, and Javier Bajo Pérez (editors). *Advances in Practical Applications of Agents and Multiagent Systems, volume 70 of Advances in Intelligent and Soft Computing*, pages 279-284. Springer Berlin Heidelberg, 2010.

- Mercedes Amor, Inmaculada Ayala, and Lidia Fuentes. A4VANET: una aplicación basada en agentes Jade-Leap para redes VANET. *In XIII Conferencia de la Asociación Española para la Inteligencia Artificial. CAEPIA TTIA 2009.* Sevilla, 9-13 de Noviembre de 2009. Actas, pages 561-570. Asociación Española para la Inteligencia Artificial, February 2009.

Workshops

- Inmaculada Ayala, Mercedes Amor, and Lidia Fuentes. An agent platform for self-configuring agents in the Internet of Things. In *Proceedings of the Thirds International Workshop on Infrastructures and Tools for Multiagent Systems. ITMAS 2012.* June 5, 2012 Valencia, Spain, pages 65-78.

- Inmaculada Ayala, Mercedes Amor, and Lidia Fuentes. Towards the automatic derivation of Malaca agents using MDE. In Wiebe Van der Haek, et. al. (editors). *The Eleventh International Workshop on agent oriented software engineering. AOSE 2010.* Toronto, Canada, 10 of May 2010, pages 61-72, May 2010.

8.3 Lessons learned

Our approach is intended for the development of AmI systems in which agents are the meaningful entities of the application. The related work has shown that the use of agent technology is not required by all AmI systems. However, once the use of the agent technology is justified by the specific requirements of the AmI system, we want to discuss the lessons learned, which include the pitfalls and main limitations of our approach from two view points: the MDE process, and considering integration and interoperability.

8.3.1 Model Driven Engineering

We are conscious of the fact that our approach may not be attractive to AmI developers that are unfamiliar with agent technologies; and the election of Pineapple as the PIM of the MDD process could also reduce the number of potential users of our approach, even those users who are experts on agent technologies. The latter, are normally familiar with agent toolkits (e.g. Jade), but in our approach they must learn how to design a MAS using Pineapple. This inconvenience is mitigated because: (1) Pineapple offers a syntax and concepts that are familiar to agent developers; (2) the proposal can be easily extended with other PIMs just by providing corresponding M2M transformations to the MalacaTiny metamodel; and (3) our MDD process starts in the design phase, with Pineapple as the solution adopted.

On the other hand, it is not possible to specify AmI general requirements at requirements level. What we have done is to extend the Pineapple agent metamodel to incorporate some properties specific to AmI systems. A more complete

solution would be to include in our MDD process, the definition of a CIM for AmI. A set of M2M transformation rules would transform the CIM to the corresponding (extended) Pineapple model, including AmI specific requirements. In addition, the CIM would be useful for identifying the weaknesses of other agent metamodels when designing AmI systems.

Finally, the development tools that our approach uses (mainly the Eclipse Modeling Framework) do not carry out the validation, evaluation or optimization of the design. Therefore, we must assume that developers would provide the best possible design of the MAS.

8.3.2 Integration and Interoperability

Another important lesson learnt is that the performance of our approach cannot compete with specific agent technologies or ad-hoc solutions developed for environments with special requirements (both hardware and software). Our approach is an alternative to general purpose agent technologies such as Jade-Leap. The added value of our approach is that it offers the possibility of modifying the agent to meet specific requirements imposed by the agent platform or the communication mechanism used by the system.

On the other hand, a good modularization of the agent's internal architecture makes the adaptation and the evolution of the agent and the MAS for new technologies and requirements easier. Currently, the appearance and disappearance of network technologies and lightweight devices (i.e. mobile phones technologies, sensors and actuators) is becoming a regular occurrence. Therefore, promoting and facilitating the evolution of AmI systems for new technologies offers a great advantage.

Nowadays the number of applications and services for lightweight devices (like tablets, smartphones) is gaining market share. In this scenario, wireless network technologies play an important role, providing full connectivity to these devices. Some years ago, the cost and the immaturity of the wireless technologies limited their use in lightweight devices. Currently this is no longer the case, most of the personal devices and sensor technologies have multiple wireless interfaces that can be successfully exploited in AmI systems. However, an additional problem

that we have found is that interoperability is not possible in all cases (for instance, sensors use IEEE 802.15.4 standards to communicate, which are still not supported by personal lightweight devices). In order to achieve interoperability we need additional solutions. One solution is the use of gateways. In this situation, our approach can enable the interoperation by generating of an adequate *Distribution* aspect and in the design of the agent to act as a gateway. This is of great value if we consider developing of specific purpose agents (using the corresponding toolkit) and later integrating them with a MAS composed of general purpose agents.

Regarding the use of aspect-orientation in the internal design of MalacaTiny, it provides an enhanced modularization, resulting in greater flexibility, which MalacaTiny exploits especially for separating the communication-related concerns. As we have shown in the preceding chapters, the weaving process introduces an additional overhead (affecting the consumption of time and memory) as it is the weaver which effectively invokes the agent functionality, introducing an extra level of indirection. But, as shown in the experiments described in Chapter 7 the differences in performance are not noticeable, especially as due to the limitations of most device platforms (e.g. Java ME), the implementation of the weaving is static. A special case is Goal Oriented MalacaTiny, where it was possible to implement dynamic weaving. In this case, the overhead introduced by the dynamic weaving is a little bit higher, but the flexibility and extensibility that agent implementations are endowed with more than compensate this penalty in performance. The conclusion is that the results shown throughout Chapter 7 demonstrate that the overhead is acceptable and it does not prevent the use of aspect-oriented agents in real AmI applications.

8.4 Future work

Having obtained good results from this thesis, we could say that our work is done, however, a thesis is probably never strictly finished. Thus, there are still several areas of our approach that could to be further investigated and improved upon. In this section, we discuss some of these issues.

In this thesis we have presented an MDD process, however we have not ac-

complished some of the goals of the MDD. Models are used as the first entities in our MDD process, however our process only has one direction. We have generated code from the PSM, and the PSM from the PIM, but we have not developed the opposite transformation. In the different phases of the development of a project it is normal that knowledge about the problem changes. So, having a bijective transformation process would allow us to keep the consistency between the different models of the system.

Related to the MDD process, we plan to develop a graphical Domain Specific Language for Pineapple and MalacaTiny. Currently, we model the system using tools provided by the EMF, but this process can be difficult for developers who are unfamiliar with this Eclipse tool. The great advantage of the EMF is that it provides ready to use tools for manipulating models. Graphical syntax for Ecore models (like our approach) are generally developed using the Graphical Modeling Framework (GMF) of Eclipse. There is a strong relationship between Ecore metamodels and its graphical representation, so any change in the underlying metamodel makes it impossible to use the graphical tool. If we use EMF tools, this problem does not exist. So, before providing a graphical language, we have waited for a more stable version of our metamodels. Now, we think is the appropriate moment to start this task. Another future development for our approach is the addition of the CIM level to the MDD process.

The development of the "Museo de la Informática" has been a great challenge, that has inspired some ideas to extend the work of *Sol* and to make this agent platform a workable solution to develop real AmI applications. Some ideas are currently under development while others are planned for the near future.

Performance is a continuous concern in the development of technologies for lightweight devices. As mentioned before, we are improving both the *Sol* agent platform and the implementation of MalacaTiny agents to make the formatting process performed by the *Representation* aspect more efficient. We are aware that our agent approach introduces an overhead that is not introduced in approaches that provide specific-purpose agents, but we are working hard to make this overhead minimal and affordable. To accomplish this task, we are using software profiling and testing tools (provided by Software Development Kits such as Android) to analyze the implementation and optimize the code accordingly.

As stated before, AmI applications must be able to be executed in a diversity of devices, with a variable set of physical features and software services availability. In order to cover the largest number of devices and technologies, we are extending our proposal with Libellium Waspmotes. This has been necessary because Sun SPOT sensor motes only offer a limited set of monitoring capabilities. These new sensors have fewer computational resources than Sun SPOT and although of Waspmote motes use ZigBee, the implementation of this specification is incompatible with the Sun SPOT base station (Waspmotes and Sun SPOT sensors form different sensor networks). So, we are endowing *Sol* with the means necessary to interact with these kinds of sensors considering their low computational capacities. Additionally, we are adapting a lightweight MalacaTiny agent for TinyOS (which already run in MicaZ motes) to execute in Libellium Waspmotes (achieving this task is well underway).

The spacial distribution of the rooms of the "Museo de la Informática" has been another concern for us. As stated in Chapter 3, the different rooms of the museum are distributed in different halls and rooms in the "E.T.S.I. en Informática" buildings. The problem is to communicate devices with low-range communication technology to the node in which *Sol* is running. Our initial solution has been to introduce the use of Libellium Meshlium Routers[1] that have support for connecting handheld and lightweight devices, supporting WiFi and Bluetooth, and Libellium Waspmotes that use ZigBee. However, this is an expensive solution, so we are working on a distributed version of our agent platform with the capacity of having remote nodes which communicate internally. These remote nodes are being implemented in a new type of connection of the IMTP.

Finally, in future work we plan to provide a more flexible mechanism for defining groups and managing their membership. Capabilities such as self-management and context awareness are considered a high priority for the AmI. We have moved these capabilities to groups. In the current implementation of *Sol*, agents join groups using a previously-known group identifier, that in some situations can be a mechanism which is much too rigid. For example, an agent wants to join a group with a specific feature (e.g. a group whose number of members is lower than 10, a group of female visitors, a group of older visitors, or a group of English-speaking

[1] `http://www.libelium.com/products/meshlium`

visitors) but it does not know its group identifier, so for this agent it becomes impossible to join such a group.

In addition, we want to facilitate the automatic joining of agents to a group once it has been created. For instance, a group of foreign student visitors could be created to tell them that they are to be given a gift. So, all the *VisitorAgents* whose users are foreign students would be automatically added to the group as soon as it is created. In addition, we plan to extend the GMS to facilitate group search queries (like a group directory service). In the current implementation of *Sol*, the GMS does not support query information of groups registered in the platform. However, extending the GMS is not an easy task because we have to extend group description (to include the features that characterize the group) and define an ontology with the necessary terms for AmI and specific domain applications.

Appendix A: Resumen

La Inteligencia Ambiental (AmI por sus siglas en inglés, Ambient Intelligence) propone una nueva visión de la tecnología que fomenta una relación más natural entre las personas y la electrónica que las rodea. En la AmI, los dispositivos electrónicos se vuelven invisibles gracias a que se integran de manera natural en nuestro entorno. En general, los sistemas AmI son distribuidos y están formados por una plétora de dispositivos que se interconectan a través de una gran diversidad de tecnologías de comunicación. El desarrollo de estos sistemas plantea nuevos retos que deben de hacerse frente utilizando tecnologías software apropiadas. En esta tesis se explora la mejora del desarrollo de las aplicaciones AmI a través de los agentes software, el desarrollo dirigido por modelos (MDD por sus siglas en inglés, Model Driven Development) y la orientación a aspectos.

Los agentes software y los Sistemas Multi-Agente (MAS por sus siglas en inglés, Multi-Agent System) son considerados buenas alternativas para el desarrollo de aplicaciones en el dominio de la AmI [Sadri, 2011]. La inteligencia, la reactividad, la proactividad y el comportamiento social de los agentes software satisfacen los requisitos de los sistemas AmI. Sin embargo, para hacer de los agentes una tecnología ampliamente aceptada para el desarrollo de sistemas AmI, sería necesario facilitar tanto el diseño como la implementación de estos sistemas, proporcionando herramientas de desarrollo adecuadas que automaticen algunas de las tareas del desarrollador, mejorando la productividad de este tipo de aplicaciones.

El MDD [Stahl and Völter, 2006] es una propuesta para el desarrollo software que promueve tanto el uso de modelos para representar de manera formal conceptos específicos de un dominio de aplicación, como la automatización de las tareas implicadas en el desarrollo de programas a través de la transformación de modelos

del sistema. Estos modelos siguen sintaxis que son establecidas por unos elementos llamados metamodelos. Actualmente, existen varios metamodelos especficos para agentes, pero los sistemas AmI tienen particularidades que deben de ser modeladas desde las fases más tempranas del desarrollo del sistema y no son tenidas en cuenta por dichos metamodelos. Las más relevantes son la necesidad de ser consciente de ciertas propiedades del entorno (sensibilidad al contexto) y la capacidad de reaccionar a cambios en estas propiedades (auto-gestión) [Kephart and Chess, 2003].

En esta tesis se define un proceso dirigido por modelos adaptado a las necesidades del desarrollo de agentes con capacidades de auto-gestión que pueden ser ejecutados en los dispositivos más usuales de los entornos AmI, teléfonos inteligentes o sensores. Nuestra solución está centrada en una arquitectura de MAS totalmente distribuida y descentralizada, gracias a la integración de los agentes en los dispositivos heterogéneos que suelen formar parte de un sistema AmI. La principal motivación para promover la integración de los agentes en dispositivos ligeros es que es posible adaptar la funcionalidad del agente al hardware y a sus recursos computacionales. Algunas de las propuestas de agentes más conocidas han lanzado nuevas versiones específicas para dispositivos ligeros (Jade-Leap [Bellifemine et al., 2001; Bergenti and Poggi, 2002], μFIPA-OS [Laukkanen et al., 2002]) y además, han aparecido propuestas concretas para estos terminales (Andromeda [Agüero et al., 2009], MAPS [Aiello et al., 2009]). Sin embargo, estas propuestas presentan serias limitaciones y deficiencias a la hora de enfrentarse con la diversidad de dispositivos y tecnologías de red tan presentes en la AmI. Estas limitaciones son abordadas como parte de esta tesis.

El MDD nos ayuda a separar las propiedades independientes de la plataforma de las que son específicas de una plataforma concreta. De esta manera, es posible modelar los requisitos especiales de los sistemas AmI independientemente de la plataforma de ejecución en la que será desplegado el sistema final. Sin embargo, en la AmI la generación de agentes es necesaria para distintos tipos de dispositivos (teléfonos móviles, diferentes tipos de sensores,...), con diferentes sistemas operativos (Android, TinyOS,...) e interconectados con distintas tecnologías de red. Para hacer frente a esta problemática, hemos usado técnicas de desarrollo software orientado a aspectos (AOSD por sus siglas en inglés, Aspect-Oriented Software De-

velopment) que nos permiten separar los elementos del agente concernientes a la comunicación de la funcionalidad específica del sistema.

A.1 Inteligencia ambiental: motivación y retos

El siglo XX fue testigo de numerosas ficciones que reflejaban la visión de escritores y cineastas sobre como serían nuestras vidas en el presente milenio. Escritores tan famosos como Philip K. Dick o Isaac Asimov capturaron en sus historias un futuro cuyo desarrollo tecnológico hace que científicos e ingenieros de hoy en día se ruboricen. ¿Dónde están los robots que respetan las 3 leyes de la robótica? ¿Dónde está HAL 9000? ¿Dónde está mi coche no tripulado? Por supuesto, están más cerca que nunca, pero aún, lejos. Actualmente, los robots no tienen una conciencia que haga necesarias unas leyes de la robótica, pero es una realidad que pueden hacer muchas cosas por nosotros, entre ellas, ayudar a los ancianos en su día a día [Pollack et al., 2002]. Tampoco tenemos computadores con tendencias psicópatas como HAL 9000, pero tenemos sistemas que se auto-configuran teniendo en cuenta nuestras preferencias para hacer nuestros hogares más confortables [Hagras et al., 2004]. El coche no tripulado no está todavía en nuestras calles, pero el *Google Driverless Car* [Guizzo, 2011] y la *DARPA Robotic Challenge*[1] demuestran que estos vehículos pueden ser una realidad en nuestras vidas. El objetivo de la AmI es hacer que estos desarrollos tecnológicos dejen de ser ciencia ficción.

En línea con estas ideas, el termino AmI fue acuñado por el IST Advisory Group (ISTAG) en el año 2001 [Ducatel et al., 2001] y más tarde revisado en [Ducatel et al., 2003]. Estos informes técnicos proporcionaban una serie de escenarios en los que la AmI tenía un rol muy importante en hacer nuestras vidas más confortables y seguras. En estos documentos se destaca el papel que debe de cumplir la tecnología de agentes para hacer realidad esta visión. Además, se recalca la necesidad de adaptar la tecnología de agentes a este nuevo entorno compuesto de dispositivos heterogéneos conectados mediante redes de comunicación diversas. Por lo tanto, para convertir la computación basada en agentes en una tecnología ampliamente aceptada para desarrollar sistemas AmI es necesario hacer frente a unos **retos**

[1] `http://www.theroboticschallenge.org/`

específicos.

R1 Modelado de agentes para sistemas AmI: La programación con tecnología de agentes para dispositivos ligeros debe facilitarse proporcionando la capacidad de expresar conceptos del dominio de la AmI con un nivel de abstracción alto. Además, deben proporcionarse herramientas de desarrollo que automaticen algunas de las tareas del programador. El objetivo de estas utilidades es simplificar la programación de los agentes, mejorando la productividad de aplicaciones de AmI, independientemente de las características de la plataforma donde se desplegará el sistema (tipo de dispositivo, protocolo de comuniación o tecnología inalámbrica). Como se ha discutido en la introducción, el MDD parece ser la tecnología más adecuada para hacer frente a este reto. Algunas propuestas ya han probado los beneficios del MDD para agentes [Agüero et al., 2009; Hahn et al., 2009; Pavón et al., 2006]. Con el MDD es posible diseñar un sistema AmI basado en agentes especificando conceptos de alto nivel en un modelo de agente independiente de la plataforma (fijándonos en el modelo del dominio), para después transformarlo automáticamente en diferentes modelos de implementación, cerrando la brecha entre diseño e implementación. Por lo tanto, un reto es proponer procesos MDD novedosos para la generación automática de agentes que pueden ser ejecutados en dispositivos heterogéneos típicos de la AmI. Sin embargo, con respecto al proceso MDD, la aplicación de esta solución para desarrollar agentes de AmI presenta los siguientes retos:

R1.1 *Facilitar el modelado de alto nivel de las característcas de las aplicaciones AmI*: Estudios de tecnologías de AmI [Cook et al., 2009; Sadri, 2011] muestran que la mayoría de los sistemas AmI actuales que están basados en agentes proporcionan soluciones ad-hoc, sin considerar el modelado de alto nivel de las propiedades de estos sistemas. En un proceso MDD, estas propiedades espcíficas del dominio deberían ser especificadas como parte de un metamodelo. Este metamodelo tendría que modelar tanto las propiedades genéricas de los agentes como aquellas que son específicas de la AmI. Aunque hay una gran cantidad de metamodelos de agentes (PIM4Agents [Hahn et al., 2009], FAML [Bey-

doun et al., 2009],...) que pueden ser utilizados para modelar aplicaciones muy diversas, estos deberían ser extendidos para incorporar propiedades específicas de entornos AmI, como la sensibilidad al contexto y la auto-gestión.

R1.2 *Facilitar la extensión del proceso MDD*: El proceso de generación debe de considerar la continua aparición de dispositivos con nuevos sistemas operativos y plataformas de agentes para ellos. Por lo tanto, un requisito crucial para un proceso de generación automática de agentes en entornos de AmI es facilitar la extensibilidad de estos procesos para incorporar nuevas tecnologías. El reto, es definir un proceso que partiendo del mismo modelo de agente, permita generar agentes que puedan ser ejecutados en plataformas distintas, en lugar de definir procesos específicos para cada plataforma o tecnología de agentes (Jade-Leap o μFIPA-OS) implicada en el sistema de AmI.

R2 Integración eficiente de agentes en dispositivos heterogéneos típicos de la AmI: Las propuestas basadas en la integración de los agentes en dispositivos típicos de la AmI proponen normalmente soluciones ad-hoc que son específicas de un sistema concreto. Integrar los agentes en estos dispositivos nos proporciona las siguientes ventajas: (i) pueden proporcionar servicios personalizados a los recursos del dispositivos en el que se están ejecutando [Stock et al., 2007]; (ii) encapsulan datos y computación de manera que permanecen ocultos de otros agentes en el MAS; (iii) permite variar los componentes, es decir los agentes, de un sistema sin modificar su arquitectura [Cook et al., 2006]; (iv) proporciona una mayor flexibilidad a la hora de modelar sistemas abiertos porque permite modelar soluciones genuinamente descentralizadas. En esta tesis nos centramos en una solución de este tipo. Los agentes son integrados en dispositivos heterogéneos, que se comunican a través de distintas de tecnologías de red y cuya ejecución debe de ser eficiente, considerando las limitaciones en recursos de algunos dispositivos AmI. A la hora de hacer frente a este desafío hemos identificado los siguientes retos:

R2.1 *Gestionar la heterogeneidad a nivel de dispositivo y plataforma de agentes*: La mayoría de los sistemas AmI están compuestos de un con-

junto heterogéneo de dispositivos. Sin embargo, las tecnologías de agentes actuales para dispositivos ligeros solamente pueden ser utilizadas en un conjunto reducido de dispositivos y no pueden interactuar con agentes desplegados en una plataforma de agentes distinta a la suya [Ayala et al., 2013b]. Esta es una limitación importante ya que hace imposible el desarrollo de algunos sistemas AmI utilizando agentes. Este caso se da si el dispositivo utilizado no soporta la plataforma de agentes que hemos elegido para nuestro sistema. Debería ser posible que agentes que pertenecen al mismo MAS sean capaces de interactuar con independencia de la plataforma y los dispositivos en los que se están ejecutando.

R2.2 *Hacer frente a la diversidad de tecnologías de red*: Normalmente, los dispositivos AmI, como los teléfonos móviles, pueden utilizar varias tecnologías de red para comunicarse. Actualmente las plataformas de agentes que pueden ser utilizadas para el desarrollo de sistemas AmI están limitadas a utilizar sólo una interfaz de red, y además, no están diseñadas para ser extendidas con nuevas tecnologías de manera sencilla [Bellifemine et al., 2001; Muldoon et al., 2006].

R2.3 *Conseguir eficiencia en el código generado*: En un sistema AmI basado en agentes, la ejecución de los agentes en dispositivos ligeros como teléfonos inteligentes, tabletas o sensores, con unos recursos computacionales limitados debe ser posible y asequible en recursos computacionales. Por lo tanto, el proceso de generación de código debe gestionar estas limitaciones con el objetivo de producir un código que haga un uso óptimo de los recursos computacionales disponibles.

R3 Agentes auto-gestionados: La mayoría de los dispositivos AmI presentan síntomas de degradación de su funcionamiento, tales como pérdida de energía o fallos de alguno de los nodos de su red. Este tipo de problemas requiere que se tomen acciones de manera explícita, como por ejemplo, ahorrar energía de un determinado componente para asegurar la supervivencia del sistema. En este contexto, la auto-gestión es de gran importancia para estos sistemas. Sin embargo, integrar la auto-gestión que es requerida por los sitemas AmI

en los agentes que los componen supone un gran reto. Concretamente, hemos encontrado los siguientes:

R3.1 *Auto-gestión descentralizada*: La descentralización y la naturaleza empotrada de los sistemas AmI dificulta la aplicación de estrategias de control sobre cada uno de los dispositivos que forman el sistema. Esto hace que las propuestas centralizadas que usan un agente o un conjunto fijo de los mismos para controlar el sistema sean difíciles de aplicar en este dominio, además de inadecuadas y económicamente inviables. Por lo tanto, el reto al que debemos de hacer frente es proponer soluciones de auto-gestión descentralizadas y autonomas. La solución propuesta debe de considerar que en un sistemas AmI coexisten dispositivos simples (unidades sensoras) con otros más complejos (teléfonos inteligentes de última generación). Por lo tanto, deben considerarse distintos tipos de auto-gestión adaptados a las capacidades de cada dispositivo y que sean capaces de interactuar cuando sea necesario.

R3.2 *Modelado e implementación de agentes auto-gestionados*: El reto en este punto es proporcionar al diseñador del sistema AmI con abstracciones de alto nivel que le permitan especificar el comportamiento de la auto-gestión como parte del metamodelo fuente de un proceso MDD. Las ventajas que nos proporcionaría lograr este reto son las siguientes: (i) modelar de manera explícita la auto-gestión mejora la capacidad de razonar sobre esta propiedad y su relación con el resto de elementos del modelo de agente considerado; (ii) el diseñador no debe de preocuparse sobre los detalles de implementación de la auto-gestión, ya que la funcionalidad es generada automáticamente por el proceso MDD; y (iii) es posible comprobar el correcto funcionamiento de la auto-gestión antes de desplegar el sistema final. Sin embargo, los metamodelos de agentes actuales no disponen de los mecanismos que permitan modelar de manera adecuada esta propiedad del sistema [Bernon et al., 2005; Beydoun et al., 2009].

A.2 Visión general

En consonancia con el reto **R1**, se ha definido un proceso MDD que permite generar de manera automática agentes que pueden ser integrados en distintos dispositivos típicos de los entornos AmI. Estos agentes pueden interactuar a través de plataformas de agentes en consonancia con el estándar para agentes FIPA, utilizando distintos protocolos de red (**R2**). Adicionalmente, nuestros agentes disponen de capacidades de auto-gestión que consideran los limitados recursos presentes en los dispositivos que suelen componer un sistema AmI (**R3**).

La Figura 1 proporciona una visión general de nuestro proceso MDD. Con el objetivo de abordar **R1.1**, se ha definido a nivel de modelado (etiqueta *Modeling* en la parte superior de la Figura 1) un Metamodelo Independiente de la Plataformateracción) como la auto-gestión (**R3**). La propiedad de auto-gestión es modelada de forma separada al resto de los elementos del MAS utilizando un lenguaje de especificación de políticas (**R3.2**). La base de Pineapple es el metamodelo PIM4Agents, que unifica los conceptos más comunes en el desarrollo orientado a agentes en un mismo metamodelo [Hahn et al., 2009]. Nosotros hemos extendido de manera significativa este metamodelo con nuevos conceptos que permiten diseñar las capacidades de auto-gestión del agente. Estos conceptos de modelado están inspirados por el lenguaje de especificación de políticas APPEL [Turner et al., 2009]. Además, hemos separado estos conceptos del resto de las propiedades del MAS añadiendo un nuevo punto de vista específico para la auto-gestión de los agentes. Por lo tanto, el primer paso de nuestro proceso MDD es el modelado del sistema AmI utilizando Pineapple. Hay que considerar, tal como se ha explicado en el reto **R3.1**, que el mismo MAS puede estar compuesto por agentes con diferentes capacidades de auto-gestión. Así que el modelado en el punto de vista de la auto-gestión dependerá de los requisitos del dispositivo donde el agente va a ser desplegado.

El reto **R1.2** está abordado en nuestro proceso por la definición e implementación de un modelo de agente de plataforma neutral (MalacaTiny), es decir, un modelo de agente que es independiente de una tecnología de desarrollo específica o de una plataforma de agentes concreta. El uso de MalacaTiny simplifica notablemente el proceso MDD si lo comparamos con otras propuestas similares

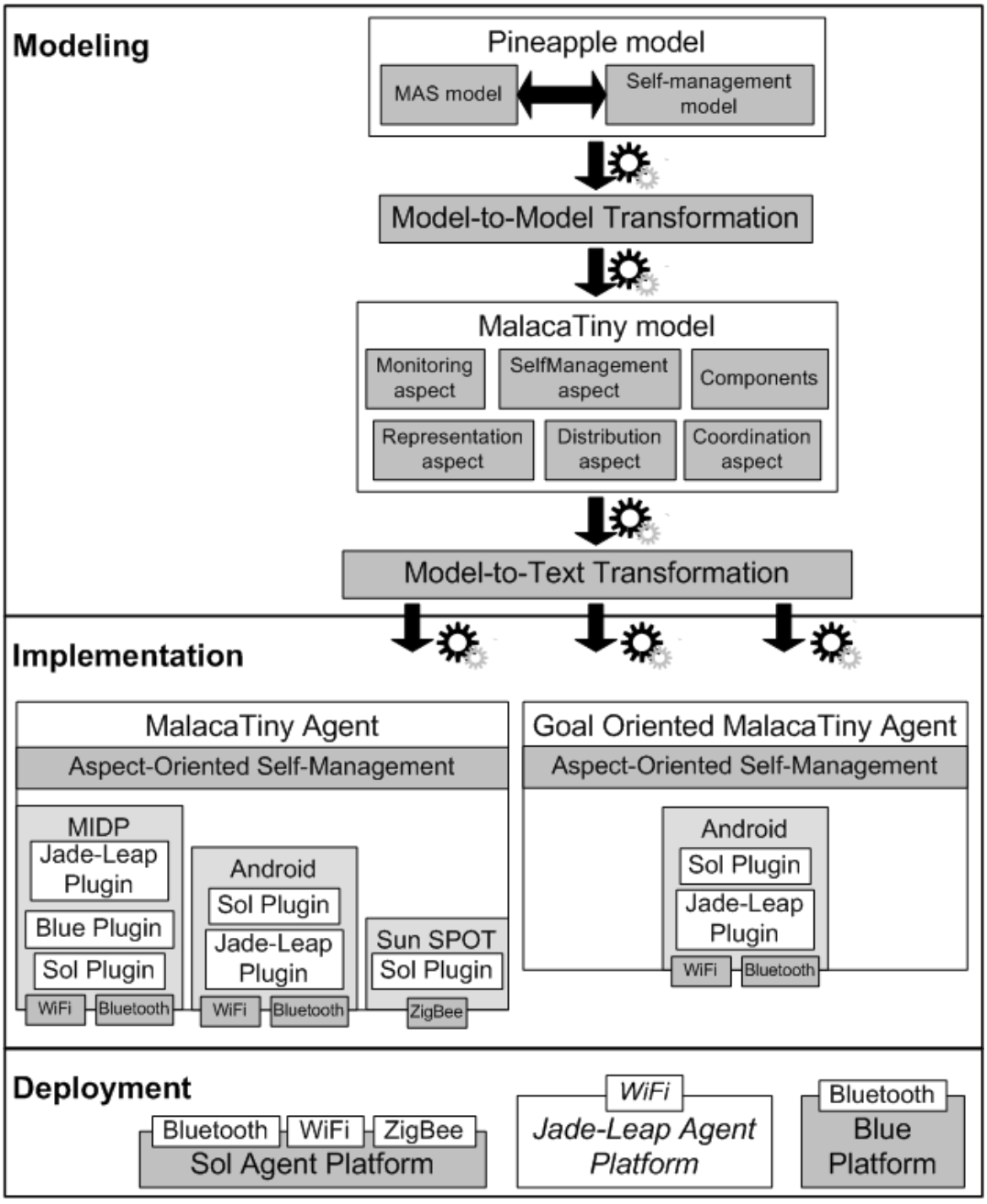

Figure 1: Visión general del proceso MDD para agentes de entornos AmI

para agentes. En un proceso dirigido por modelos estándar, para cada una de las plataformas de agentes implicadas en el proceso, se requieren 2 procesos de transformación, de modelo-a-modelo (M2M por sus siglas en inglés, Model-to-Model) y de modelo-a-texto (M2T por sus siglas en inglés, Model-to-text) [Gascueña et al., 2012]. En nuestra solución, en lugar de generar un agente que sólo puede ser desplegado en una plataforma (por ejemplo Jade-Leap), nuestro proceso genera agentes que pueden ser desplegados en distintas plataformas, incluso de manera simultánea. De esta forma, simplificamos el proceso MDD, dado que solo necesitamos 2 procesos (M2M y M2T), independientemente del número de plataformas de agentes implicadas en el proceso. La neutralidad a nivel de plataforma de MalacaTiny se logra utilizando orientación a aspectos. Los conceptos relativos a la plataforma de comunicaciones (e.g. distribución de mensajes, codificación,...) son modelados utilizando aspectos. Esto hace posible generar agentes adaptados a una plataforma de agentes concreta simplemente *tejiendo* el aspecto adecuado.

Una vez que el diseñador ha modelado el sistema AmI en Pineapple, la transformación M2M es ejecutada y se genera automáticamente un modelo en MalacaTiny que contiene el conjunto de agentes que forman el sistema AmI. Hemos evaluado los beneficios de aplicar un enfoque dirigido por modelos a nuestro proceso MDD aplicando la métrica Grado de Automatización [Harrington and Cahill, 2011]. Los resultados muestran que es posible generar automáticamente aproximadamente un 40% de el código en sistemas AmI complejos.

Con respecto a la propiedad de auto-gestión, algunos dispositivos AmI como las motas (i.e. unidades sensoras) son muy simples, y no tiene sentido implementar políticas de auto-gestión complejas para ellas. Por lo tanto, nuestro proceso MDD considera la generación de agentes con distintas capacidades de auto-gestión, pero la interacción entre estas capacidades sigue siendo posible (**R3.1**). Para aquellos agentes MalacaTiny con necesidades de auto-gestión simples, es aplicado un proceso de transformación M2T que genera agentes MalacaTiny reactivos. Para agentes que requieren una auto-gestión más sofisticada es aplicado un proceso de transformación diferente, que genera agentes cuyo comportamiento es orientado a objetivos. La orientación a objetivos nos permite implementar comportamiento de auto-gestión basado en objetivos de alto nivel (**R3.2**). Consideramos que un agente es orientado a objetivos cuando las acciones que lleva a cabo son a causa

de la realización de un objetivo.

En nuestro modelo de implementación (etiqueta *Implementation* en la Figura 1), el agente puede ajustar su funcionamiento para hacer frente a circunstancias cambiantes o a fallos en el hardware o en el software. Ambas implementaciones, tanto la reactiva como la orientada a objetivos, usan orientación a aspectos en su diseño interno. En consecuencia, los agentes MalacaTiny están compuestos de componentes y aspectos, que contribuyen a la mejora de la adaptación del agente, la reutilización de sus componentes y el mantenimiento de su arquitectura. Con todo ello, nuestra propuesta hace frente a **R2.3** (eficiencia del código generado) ya que: (i) genera implementaciones adaptadas a los recursos del dispositivo, es decir, tenemos una versión reactiva con un consumo de recursos bajo, y una versión orientada a objetivos que consume más recursos, pero puede realizar una gestión más sofisticada; (ii) el código generado por la transformación de modelo a texto está optimizado para el dispositivo objetivo. Hemos evaluado la eficiencia y el rendimiento de MalacaTiny comparadolo con Jade-Leap y los resultados muestran que el diseño de MalacaTiny es muy eficiente. Usando nuestra propuesta, incluso sobre otra plataforma como Jade-Leap, la penalización que se añade en el consumo de recursos es mínima. Las distintas versiones de MalacaTiny pueden ser integradas en varios dispositivos y desplegados en distintas plataforma que cumplan el estándar FIPA. Concretamente, los agentes MalacaTiny pueden ser ejecutados en dispositivos Android, dispositivos con perfil MIDP (teléfonos móviles que soportan Java ME [Oracle, 2013]) y motas sensoras Sun SPOT [Labs, 2013].

La orientación a aspectos también nos ayuda a hacer frente al reto **R2.1** (Gestionar la heterogeneidad a nivel de dispositivo y plataforma de agentes) y **R2.2** (Hacer frente a la diversidad de tecnologías de red). El aspecto de distribución de mensajes encapsula cómo usar y acceder al servicio de transporte de mensajes (MTS por sus siglas en inglés, Message Transport Service), su implementación por tanto depende de los servicios ofrecidos por la plataforma y el protocolo de transporte utilizado. Este aspecto mantiene ocultos las dependencias a nivel de plataforma y hace el resto de las clases de la arquitectura del agente (componentes y aspectos) independientes de la plataforma de agentes (**R2.1**) y la tecnología de red usada en tiempo de ejecución (**R2.2**). Además, es posible que un agente se comunique con otro a través de distintas plataformas de agentes. Para ello, sólo

es necesario instanciar el aspecto de distribución adecuado para cada plataforma (e.g. Jade-Leap).

Gracias a la introducción de aspectos, la modularización de la arquitectura es mejorada y se facilita añadir una nueva plataforma de agentes a nuestra propuesta, sólo es necesario la implementación de un nuevo aspecto de distribución. Con el objetivo de ilustrar el esfuerzo de añadir una nueva plataforma a nuestra propuesta, y para hacer frente a **R2.2**, implementamos *Blue*, una plataforma de agentes basada en Bluetooth. Además, para abordar **R2.2**, hemos implementado *Sol*, una plataforma que soporta el despliegue de sistemas AmI compuestos por un conjunto de agentes auto-gestionados que son desplegados en dispositivos heterogéneos. Las principales características de *Sol* son el soporte a la comunicación entre agentes integrados en dispositivos heterogéneos (**R2.1**), a la vez que hacemos frente a protocolos de transporte diversos (WiFi, Bluetooth y ZigBee), y la comunicación grupal que suele ser necesaria en los sistemas AmI (**R2.2**).

Finalmente, los agentes generados por el proceso MDD pueden ser desplegados en plataformas que cumplen el estándar FIPA (etiqueta *Deployment* en la Figura 1). Debido a la capacidad de auto-gestión, es posible que un agente integrado en un dispositivo AmI (e.g. un teléfono inteligente) pueda auto-configurar la plataforma de agentes en la que está desplegado y/o el protocolo de transporte utilizado (este último caso solo se da con *Sol*) usado en cada momento, dependiendo de los recursos disponibles. Con el objetivo de ilustrar los beneficios de la capacidad de auto-gestión de los agentes MalacaTiny, hemos realizado una serie de experimentos que muestran las ventajas de cambiar el protocolo de transporte con el objetivo de ahorrar energía. Hemos implementado distintos objetivos de auto-gestión, como extender la vida del sistema o recuperar el dispositivo de algunos fallos. Además, hemos usado el proceso presentado aquí para implementar: un museo inteligente que puede encontrarse en la E.T.S.I. de Informática de Málaga, distintas versiones de sistemas de transporte inteligentes y aplicaciones de ambientes asistidos.

A.3 Contribuciones

En esta sección enumeraremos las que consideramos que son las contribuciones más relevantes de esta tesis.

1. Hemos desarrollado un metamodelo de agentes llamado Pineapple que incluye conceptos para modelar la propiedad de auto-gestión [Ayala et al., 2011b, 2013c]. Concretamente, nos hemos centrado en el modelado de políticas para describir cuando y cómo ajustar el comportamiento del agente y del MAS. Las políticas están integradas en los roles que desempeñará el agente en tiempo de ejecución. La base de Pineapple es el metamodelo PIM4Agents. Hemos integrado conceptos del lenguaje de descripción de políticas APPEL. La utilización de este lenguaje nos permite validar las políticas utilizando herramientas que ya están disponibles para APPEL, por ejemplo, el verificador de modelos UMC.

2. Hemos definido el metamodelo MalacaTiny [Ayala et al., 2013b]. Un metamodelo de plataforma neutral que usa orientación a aspectos para representar de manera separada la funcionalidad específica de la aplicación y los elementos concernientes a la comunicación del agente a través de un mecanismo concreto.

3. Hemos definido un proceso MDD que genera agentes MalacaTiny [Ayala et al., 2011a, 2013a,b]. Este proceso está compuesto por una transformación M2M entre Pineapple y MalacaTiny, y un conjunto de transformaciones M2T que permiten generar código de las distintas versiones de MalacaTiny. El uso de estos agentes (a nivel de modelado y despliegue) facilita la extensión del proceso MDD. Los agentes MalacaTiny son agentes de plataforma neutra, así que podemos utilizar el mismo proceso MDD para generar agentes que pueden ser desplegados en una variedad de plataformas de agentes.

4. Hemos implementado MalacaTiny [Ayala et al., 2011a, 2013a,b], una familia de agentes orientados a aspectos que pueden ser ejecutadas en dispositivos con pocos recursos computacionales. Los agentes MalacaTiny pueden ser ejecutados en dispositivos Android, motas sensoras Sun SPOT y teléfonos

móviles que tengan soporte para J2ME. Estos agentes son de plataforma neutra, lo que significa que pueden ser desplegados en distintas plataformas de agentes, tales como Jade-Leap, *Sol* y *Blue*.

5. Hemos definido una versión orientada a objetivos de MalacaTiny que puede ser ejecutada en dispositivos Android [Ayala et al., 2012d]. Esta arquitectura de agentes explota la reflexividad de la API de Java para dispositivos Android. El proceso de composición de aspectos de MalacaTiny es diferente al de MalacaTiny orientada a objetivos, esta última considera un modelo de puntos de unión (*join point model*) extensible.

6. Con el objetivo de superar las limitaciones actuales de las plataformas de agentes para dispositivos ligeros hemos desarrollado la plataforma *Sol* [Ayala et al., 2012a,b]. Las principales características de esta plataforma son: el soporte para múltiples tecnologías de red (ZigBee, WiFi y Bluetooth) y la comunicación grupal. Además, para comprobar las posibilidades de la neutralidad a nivel de plataforma, hemos desarrollado una plataforma de agentes basada en Bluetooth llamada *Blue* y un aspecto de distribución de mensajes para la plataforma Jade-Leap.

7. Hemos integrado la auto-gestión dentro de la arquitectura del agente [Ayala et al., 2011a, 2012b,c,d, 2013a]. Asimismo, es tenida en cuenta en cada una de las fases del desarrollo del sistema. Por lo tanto, puede ser especificado y validado en la fase de modelado. Además, es posible generar el código relacionado con la auto-gestión usando nuestro proceso MDD.

8. También pertenece al ámbito de esta tesis distintas aplicaciones y prototipos que hemos desarrollado para estudiar la aplicación de la tecnología de agentes para el desarrollo de sistemas AmI. Usando Jade-Leap, implementamos una aplicación de sistemas de transporte inteligente [Amor et al., 2009, 2010]. Con la tecnología de agentes MalacaTiny, y enfocándonos en cuestiones de comunicación entre dispositivos heterogéneos, desarrollamos un sistema de detección de caídas que usaba un dispositivo Android y una mota sensora Sun SPOT [Ayala et al., 2011a, 2013a]. Además, hemos desarrollado una aplicación para el "Museo de la Informática" situado en la "E.T.S.I. Informática"

de Málaga [Ayala et al., 2012a].

A.4 Estructura de la tesis

Continuando con esta introducción, el primer capítulo de esta tesis explica al lector los fundamentos de los principios utilizados en las soluciones propuestas. En este capítulo se explicarán los conceptos fundamentales, el estado del arte y el trabajo relacionado. A continuación, nuestra propuesta será presentada en los siguientes 4 capítulos. En la última parte de esta tesis, se presentarán resultados que validan las soluciones propuestas y se discutirán las contribuciones de nuestro trabajo junto con las conclusiones y el trabajo futuro.

Background

Este capítulo muestra los conceptos necesarios para comprender el resto de esta disertación. Las características de la Inteligencia Ambiental son descritas, así como las aplicaciones más relevantes en esta área. A continuación, se analiza como la tecnología de agentes es aplicada al entorno de la AmI. Tras esto, la auto-gestión, otra parte muy importante de esta tesis, es descrita. Hemos prestado una atención especial a la propuesta de IBM conocida como *Autonomic Computing*. La última parte de este capítulo está dedicada al MDD. Son descritas las bases de esta tecnología y se revisan sus contribuciones a la computación orientada a agentes. Finalmente, revisamos las tecnologías de MDD usadas para el desarrollo de esta tesis.

A metamodel for self-managed agents

En este capítulo, presentamos el caso de estudio que vamos a utilizar para ilustrar las distintas contribuciones de esta disertación. Este caso de estudio se corresponde a la aplicación de un museo inteligente situado en la "E.T.S.I. Informática" de Málaga. Además el metamodelo de Pineapple es descrito y además, se explica cómo modelar una aplicación de AmI utilizándolo. Concretamente, en este capítulo

nos centramos en cómo modelar la aplicación y las propiedades de auto-gestión. Asimismo, describimos el método para validar el comportamiento auto-gestionado de los agentes MalacaTiny usando el verificador de modelos UMC.

From Pineapple to MalacaTiny

Este capítulo está centrado en la transformación entre Pineapple y MalacaTiny. Comenzamos con la presentación del metamodelo MalacaTiny. A continuación, proporcionamos una descripción detallada del proceso de transformación M2M que utiliza Pineapple para generar modelos de MalacaTiny. Distinguimos 3 partes en esta transformación, la transformación de la arquitectura del agente, la generación de aspectos y la generación de la funcionalidad de auto-gestión.

Code generation of MalacaTiny agents

La implementación de las distintas versiones de MalacaTiny y el proceso de transformación M2T son descritos en este capítulo. Los agentes MalacaTiny pueden ser ejecutados en teléfonos móviles con perfil MIDP, dispositivos Android y motas sensoras Sun SPOT. Las arquitecturas de estos agentes presentan sólo pequeñas diferencias, así que nos enfocamos en uno de ellas, los agentes que se ejecutan en dispositivos con perfil MIDP. Además, se describe el diseño interno de MalacaTiny orientada a objetivos y sus diferencias con las otras versiones de MalacaTiny. Con respecto a la autogestión, este capítulo trata el tema de cómo integrarla en la arquitectura de los agente usando aspectos. Finalmente, son presentadas las transformaciones M2T para MalacaTiny y MalacaTiny orientada a objetivos.

The communication concern

En este capítulo describimos los dos aspectos de distribución que se han desarrollado específicamente para MalacaTiny y MalacaTiny orientada a objetivos, *Blue* y el correspondiente a la plataforma de agentes *Sol*. *Blue* es un aspecto de distribución basado en Bluetooth específico para teléfonos móviles con perfil MIDP

que fue desarrollado para estudiar el consumo de recursos de los agentes MalacaTiny. Por otro lado, la plataforma de agentes *Sol* es nuestra solución para comunicar agentes ejecutándose en dispositivos heterogéneos y utilizando tencologías de red diversas. El desarrollo de esta plataforma ha sido muy importante para el Museo Inteligente.

Validation

La validación de ciertos aspectos de esta disertación es presentada en este capítulo. Concretamente, discutimos y presentamos resultados sobre la generación de código, el rendimiento y el consumo de recursos de los agentes MalacaTiny. Además, mostramos resultados que ilustran la latencia de la funcionalidad de auto-gestión y del intercambio de mensajes, utilizando los distintos aspectos de distribución implementados para MalacaTiny, *Blue*, Jade-Leap y *Sol*.

Conclusions

Este capítulo resume el trabajo presentado a lo largo de esta disertación, resaltando las contribuciones de nuestro trabajo. Además, detallamos las publicaciones más importantes obtenidas de estas contribuciones. Para finalizar, se discuten las lecciones aprendidas durante el desarrollo de este trabajo y se esbozan algunas posibles líneas de trabajo futuro.

Appendix A: Resumen

Este apéndice presenta un resumen de esta tesis en Español.

Appendix B: Conclusiones

Este apéndice presenta las conclusiones de este trabajo, las publicaciones obtenidas, las lecciones aprendidas y el trabajo futuro en español.

APPENDIX A

Appendix B: Conclusiones

En este apéndice se resume la propuesta presentada a lo largo de esta disertación resaltando sus contribuciones más relevantes (Section B.1). Además, en la Sección B.2, se detallan las publicaciones más importantes obtenidas de estas contribuciones. La Sección B.3 trata las lecciones aprendidas durante el desarrollo de esta tesis. Finalmente, se explicaran nuestras líneas de trabajo futuro (Sección B.4) surgidas a partir de este trabajo.

B.1 Resumen

Después de los capítulos de Introducción (Capítulo 1) y Antecedentes (Capítulo 2), se ha presentado un proceso MDD (Capítulo 3) para generar automáticamente sistemas basados en agentes a partir de un diseño de MAS. Este proceso se apoya en MalacaTiny, un marco que incluye tecnologías de agentes para dispositivos ligeros y plataformas de agentes, y que resulta especialmente adecuando para el desarrollo de agentes en sistemas AmI. La solución dirigida por modelos que hemos propuesto en esta disertación cubre las fases de diseño e implementación. En nuestro proceso, transformamos modelos de Pineapple, un metamodelo para MAS de propósito general, que hemos adaptado para modelar sistemas sensibles al contexto con propiedades de auto-gestión. El resultado es un conjunto de agentes MalacaTiny auto-gestionados, que pueden ser ejecutados en distintos dispositivos ligeros típicos de la AmI.

Se ha definido el metamodelo Pineapple, cuyo propósito es el modelado de sistemas AmI basados en agentes auto-gestionados (Capítulo 3) y está basado en

el metamodelo PIM4Agents. Hemos extendido PIM4Agents con un nuevo punto de vista de modelado, llamado *Self-Management*, que permite modelar roles para la auto-gestión (incluyendo el conocimiento relacionado con esta propiedad) y las políticas que dirigen el comportamiento auto-gestionado del sistema AmI. Estas políticas son descritas utilizando un lenguaje de dominio específico que sigue la sintaxis de APPEL, un lenguaje de especificación de políticas. Esta descripción de las políticas nos permite detectar conflictos entre las políticas de auto-gestión usando el verificador de modelos UMC. Finalmente, este nuevo punto de vista en el modelado de MAS también incluye acciones específicas para modelar la funcionalidad que todo sistema auto-gestionado debe de poseer: auto-conciencia (*self-awareness*), auto-ubicación (*self-situation*), auto-monitorización (*self-monitoring*) y auto-ajuste (*self-adjusting*).

En el Capítulo 4, se define el metamodelo de MalacaTiny, un metamodelo específico de una plataforma, concretamente de la tecnología de agentes MalacaTiny. Este metamodelo traslada las ventajas de la arquitectura de agentes Malaca a nivel de metamodelo. La principal característica de la arquitectura interna de los agentes MalacaTiny es que mantiene separados a la funcionalidad específica de la aplicación y a las propiedades extra-funcionales del agente. Esta separación mejora la modularización de la arquitectura interna del agente. Esto contribuye a mejorar la adaptación, la reutilización y el mantenimiento del agente. Además, los aspectos encapsulan la funcionalidad relacionada con la plataforma de comunicaciones en la que el agente será desplegado, lo que convierte a los agentes MalacaTiny en neutrales respecto a las plataformas de agentes FIPA.

Hemos presentado el diseño interno de los agentes MalacaTiny y los agentes MalacaTiny orientados a objetivos (Capítulo 5). MalacaTiny es una familia de agentes para dispositivos ligeros que se inspira en la arquitectura de agentes Malaca. Estos pueden ser ejecutados en motas sensoras Sun SPOT, teléfonos móviles con perfil MIDP y dispositivos Android. Además, pueden utilizar distintos mecanismos para comunicarse, que incluyen la plataforma de agentes Jade-Leap, un mecanismo basado en Bluetooth llamado *Blue* y nuestra propia plataforma para agentes heterogéneos, *Sol*. MalacaTiny orientada a objetivos es una versión más avanzada de estos agentes específica para dispositivos Android. Sus principales diferencias son el comportamiento orientado a la consecución de objetivos y

su proceso de tejido de aspectos dinámico.

El Capítulo 6 se ha ocupado de presentar implementaciones que conciernen a la comunicación entre agentes (Capítulo 6). Hemos implementado un mecanismo de comunicación basado en Bluetooth para comunicar directamente dos dispositivos con perfil MIDP. Además, con el objetivo de proporcionar una solución factible para el desarrollo de sistemas AmI basados en agentes, hemos implementado la plataforma de agentes *Sol*. Esta plataforma, que está en consonancia con el estándar FIPA, permite la comunicación entre agentes desplegados en dispositivos heterogéneos usando tecnologías de red distintas. *Sol* actúa como una pasarela (*gateway*) que facilita la comunicación y el intercambio de servicios entre agentes que se ejecutan en dispositivos heterogéneos. Así mismo, proporciona servicios para la gestón y comunicación de grupos de nodos del sistema AmI. Esta característica es importante, ya que muchas aplicaciones de ésta área requieren este tipo de comunicación.

La auto-gestión es una propiedad de gran importancia para los entornos AmI. En esta disertación, hemos proporcionado un marco de trabajo que permite modelar políticas de auto-gestión utilizando la notación de APPEL, validarlas utilizando el verificador de modelos UMC y desplegarlas en los agentes MalacaTiny (Capítulos 3, 4 and 5).

Finalmente, en el Capítulo 7, hemos mostrado la evaluación de las distintas implementaciones que se han desarrollado en el marco de esta disertación. Se ha evaluado la conveniencia de aplicar un enfoque MDD utilizando la métrica Grado de Automatización (*Degree of Automation*). Los resultados muestran que puede generarse alrededor de un 40% del código en sistemas AmI complejos. Además, hemos presentado y discutido una evaluación de la implementación de MalacaTiny para dispositivos con perfil MIDP y Android usando Jade-Leap y la plataforma *Blue* evaluando parámetros como el rendimiento, el consumo de memoria y la eficiencia energética. También hemos realizado una comparativa con Jade-Leap que pone de manifiesto que el diseño interno de MalacaTiny es muy eficiente con respecto a la memoria, el consumo de batera y la latencia de los mensajes intercambiado entre los agentes. Los resultados de los experimentos mostraron que usando nuestro marco de trabajo, incluso sobre otra plataforma propietaria, tenemos una penalización muy leve con respecto a las variables mencionadas. En

particular, hemos obtenido muy buenos resultados con la comunicación basada en Bluetooth con respecto a la escabilidad y el consumo de batería. De esta manera, evidenciamos que es ventajoso disponer de agentes con mecanismos de comunicación alternativos y configurables en tiempo de ejecución. Además, hemos realizado otros experimentos mostrando que la auto-gestión de los agentes MalacaTiny es factible. Este capítulo concluye con una evaluación del rendimiento de la plataforma *Sol*, que muestra el rendimiento de esta plataforma en términos de latencia de las comunicaciones inalámbricas.

En resumen, las principales contribuciones de nuestro trabajo son las siguientes:

1. El metamodelo Pineapple, para el modelado de sistemas AmI basados en agentes con capacidades de auto-gestión y sensibles al contexto.
2. El metamodelo MalacaTiny, un metamodelo de agentes de plataforma neutra basados en aspectos.
3. Un proceso de desarrollo dirigido por modelos que partiendo de un modelo Pineapple y genera un conjunto de agentes MalacaTiny.
4. La implementación y la evaluación de los agentes MalacaTiny para teléfonos móviles con perfil MIDP, motas sensoras Sun SPOT y dispostivos Android.
5. La versión orientada a objetivos de MalacaTiny para dispositivos Android.
6. La plataforma de agentes *Sol* para comunicar agentes heterogéneos que usan mecanismos de comunicación distintos.
7. La implementación y la evaluación del aspecto de distribución para JadeLeap, *Blue* y la plataforma *Sol*.
8. El modelado de la auto-gestión y la integración de esta funcionalidad dentro de la arquitectura del agente utilizando orientación a aspectos.

B.2 Publicaciones

Los resultados de este trabajo de investigación han sido publicados en las siguientes revistas internacionales, conferencias y talleres de trabajo, con revisión por pares,

de la siguiente manera:

Revistas

- Inmaculada Ayala, Mercedes Amor, and Lidia Fuentes. A model driven engineering process of platform neutral agents for ambient intelligence devices. *Autonomous Agents and Multi-Agent Systems*, pages 1-42, 2013.

- Inmaculada Ayala, Mercedes Amor, and Lidia Fuentes. Self-configuring agents for ambient assisted living applications. *Personal and Ubiquitous Computing*, pages 1-11, 2012.

Conferencias

- Inmaculada Ayala, MercedesAmor Pinilla, and Lidia Fuentes. Enhancing agent metamodels with self-management for ami environments. In *Progress in Artificial Intelligence.* Springer Berlin Heidelberg, 2013. To appear.

- Inmaculada Ayala, Mercedes Amor and Lidia Fuentes. Exploiting dynamic weaving for self-managed agents in the IoT. In Ingo J. Timm and Christian Guttmann (editors) *Multiagent System Technologies, volume 7598 of Lecture Notes in Computer Science*, pages 5-14. Springer Berlin Heidelberg, 2012.

- Inmaculada Ayala, Mercedes Amor, and Lidia Fuentes. Self-starmas: A multi-agent system for the self-management of AAL applications. In *Proceedings of the 2012 Sixth International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing, IMIS'12*, pages 901-906, Washington, DC, USA, 2012. IEEE Computer Society.

- Inmaculada Ayala, Mercedes Amor, and Lidia Fuentes. Self-management of ambient intelligence systems: a pure agent-based approach. In *Proceedings of the 11th International Conference on Autonomous Agents and Multiagent Systems-Volume 3, AAMAS'12*, pages 1427-1428, Richland, SC, 2012. International Foundation for Autonomous Agents and Multiagent Systems.

- Inmaculada Ayala, Mercedes Amor, and Lidia Fuentes. Autonomic agents for mobile ambient assisted living applications. In José Bravo; Diego López

de Piña; Sergio Ochoa y Jesús Favela (editors). In *Proceedings of the 5th International symposium on ubiquitous computing and ambient intelligence. UCAmI 2011.* Riviera Maya, Mexico, december 5-9, December 2011.

- Inmaculada Ayala, MercedesAmor Pinilla, and Lidia Fuentes. Modeling context-awareness in agents for ambient intelligence: An aspect-oriented approach. In Luis Antunes and H. Sofia Pinto (editors). *Progress in Artificial Intelligence, volume 7026 of Lecture Notes in Computer Science*, pages 29-43. Springer Berlin Heidelberg, 2011.

- Inmaculada Ayala, Mercedes Amor, and Lidia Fuentes. A model driven development of platform-neutral agents. In Jürgen Dix and Cees Witteveen (editors). *Multiagent System Technologies, volume 6251 of Lecture Notes in Computer Science*, pages 3-14. Springer Berlin Heidelberg, 2010.

- Mercedes Amor, Inmaculada Ayala, and Lidia Fuentes. A4VANET: Context aware Jade-Leap agents for vanets. In Yves Demazeau, Frank Dignum, Juan M. Corchado, and Javier Bajo Pérez (editors). *Advances in Practical Applications of Agents and Multiagent Systems, volume 70 of Advances in Intelligent and Soft Computing*, pages 279-284. Springer Berlin Heidelberg, 2010.

- Mercedes Amor, Inmaculada Ayala, and Lidia Fuentes. A4VANET: una aplicación basada en agentes Jade-Leap para redes VANET. *In XIII Conferencia de la Asociación Española para la Inteligencia Artificial. CAEPIA TTIA 2009.* Sevilla, 9-13 de Noviembre de 2009. Actas, pages 561-570. Asociación Española para la Inteligencia Artificial, February 2009.

Talleres de trabajo

- Inmaculada Ayala, Mercedes Amor, and Lidia Fuentes. An agent platform for self-configuring agents in the Internet of Things. In *Proceedings of the Thirds International Workshop on Infrastructures and Tools for Multiagent Systems. ITMAS 2012.* June 5, 2012 Valencia, Spain, pages 65-78.

- Inmaculada Ayala, Mercedes Amor, and Lidia Fuentes. Towards the automatic derivation of Malaca agents using MDE. In Wiebe Van der Haek, et. al. (editors). *The Eleventh International Workshop on agent oriented software engineering. AOSE 2010.* Toronto, Canada, 10 of May 2010, pages 61-72, May 2010.

B.3 Lecciones aprendidas

Nuestra propuesta está ideada para el desarrollo de sistemas AmI en los que los agentes son las entidades más significativas de la aplicación. El trabajo relacionado ha mostrado que el uso de la tecnología de agentes no es necesario en todos los sistemas AmI. Sin embargo, una vez que el uso de la tecnología de agentes está justificado por los requisitos de la aplicación, queremos discutir las lecciones aprendidas de este trabajo, que incluyen las principales limitaciones de nuestra propuesta. En esta discusión, vamos a considerar 2 puntos de vista: por un lado, el del proceso de ingeniería dirigido por modelos; y por otro, la integración y la interoperabilidad.

B.3.1 Ingeniería Dirigida por Modelos

Somos conscientes de que nuestra propuesta puede no resultar atractiva para los desarrolladores de entornos AmI que no están familiarizados con la tecnología de agentes. Además, la elección de Pineapple como PIM de nuestro proceso MDD puede reducir el número de usuarios potenciales, incluso en el caso de que sean usuarios de la tecnología de agentes. Estos últimos, suelen estar familiarizados con herramientas para el desarrollo de agentes (e.g. Jade), pero en caso de que quisieran usar nuestra propuesta, deberían de aprender cómo diseñar MAS usando Pineapple. En nuestra propuesta este inconveniente se ve aliviado debido a que (1) Pineapple ofrece una sintaxis y unos conceptos que son familiares para los desarrolladores de agentes, (2) la propuesta puede ser extendida a otros PIM simplemente proporcionando un transformación M2M para MalacaTiny, y (3) nuestro proceso MDD comienza en la fase de diseño, utilizando Pineapple.

Por otra parte, nuestro proceso no cuenta con una fase de especificación de

requisitos generales para sistemas AmI. Lo que hemos hecho es extender el metamodelo Pineapple para incorporar algunas propiedades específicas de sistemas AmI. Una solución más completa incluiría un metamodelo independiente de computación (CIM) a nuestro proceso MDD. Una transformación M2M convertiría este CIM a su corresponiente (y extendido) modelo de Pineapple, que incluiría requisitos específicos de AmI. Además, este CIM sería útil para identificar las debilidades de otros modelos de agentes para diseñar sistemas AmI.

Finalmente, las herramientas de desarrollo que nuestra propuesta usa (principalmente el marco de trabajo para modelado de Eclipse) no lleva a cabo una validación del modelo del sistema. Tampoco tiene en cuenta un proceso de evaluación u optimización. Por lo tanto, debemos asumir que el diseñador nos proporcionará en todos los casos el mejor de los diseños posibles del MAS.

B.3.2 Integración e Interoperabilidad

Otra importante cuestión que no podemos dejar de tener en cuenta, es que nuestra propuesta no puede competir en rendimiento con tecnologías de agentes especialmente ideadas para un sistema en concreto. Nuestra propuesta es una alternativa a las tecnologías de agentes de propósito general tales como Jade-Leap o AFME. Nuestro valor añadido es ofrecer la posibilidad de modificar al agente para cumplir un requisito impuesto por la plataforma de agentes, como el mecanismo de comunicación utilizado por el sistema.

Por otro lado, la buena modularización de los agentes MalacaTiny hace más sencilla la adaptación y la evolución de la arquitectura del agente. Si consideramos la velocidad a la que aparecen y desaparecen nuevas tecnologías de comunicación y dispositivos hoy en día, poder contar con una tecnología que facilite la adopción de nuevos avances, es una gran ventaja.

Actualmente, el número de aplicaciones y servicios para dispositivos ligeros, como teléfonos inteligentes y tabletas está ganando cuota de mercado. En este escenario, las tecnologías de comunicación inalámbricas tienen un papel muy importante proporcionando conectividad a estos dispositivos. Hace algunos años, el coste y la inmadurez de estas tecnologías limitaba su uso para este tipo de dispositivos. En la actualidad, la situación es muy distinta, ya que la mayoría de los

dispositivos personales e incluso los sensores, pueden disponer de varias interfaces de red que pueden ser explotadas con éxito por los sistemas AmI. Sin embargo, un problema adicional que se presenta es que en ocasiones no es posible que estos dispositivos se comuniquen entre ellos porque utilizan distintas tecnologías de red, por ejemplo Bluetooth y WiFi. O que implementen de manera distinta el mismo estándar de red y eso haga imposible la comunicación, como es el caso de las motas sensoras Sun SPOT y MicaZ. En estas situaciones, necesitamos soluciones adicionales, como es la utilización de pasarelas (*gateways*). En este punto, nuestra propuesta puede habilitar la comunicación, desarrollando nuevos aspectos de distribución que permitan al agente actuar como pasarela. Esto ofrece la posibilidad de desarrollar sistemas en los que convivan e interactúen agentes de propósito específico, optimizados para cumplir unos requisitos concretos, con agentes de propósito general.

Con respecto al uso de la orientación a aspectos en el diseño interno de MalacaTiny, nos proporciona una modularización mejorada que proporciona una gran flexibilidad. Este hecho es explotado con éxito por estos agentes, a la hora de separar los componentes relacionados con la comunicación. Como se mostró en el Capítulo 7 (Validación), las diferencias en el rendimiento no son notables, esto se debe principalmente a que nuestro proceso de tejido de aspectos es estático para los dispositivos con perfil MIDP. En el caso de MalacaTiny orientada a objetivos, donde fue posible la implementación de un proceso de tejido dinámico, la sobrecarga introducida es mayor, pero la flexibilidad y la extensibilidad de la que hemos dotado al agente, compensan la penalización en el consumo de recursos. Lo que concluímos del Capítulo 7, es que la sobrecarga introducida por la orientación a aspectos es aceptable y no debería de ser una razón para dejar de beneficiarse de la tecnología de aspectos en aplicaciones de AmI reales.

B.4 Trabajo futuro

Si bien podemos considerar que las contribuciones de esta disertación forman un trabajo completo, siempre es difícil dar por cerrada una tesis. En consecuencia, en esta sección trataremos varias partes de nuestra propuesta que merecerían una

investigación más a fondo o podrían ser mejoradas.

En esta tesis hemos presentado un proceso MDD, sin embargo no hemos llevado a cabo alguno de los objetivos de esta disciplina. Los modelos son usados como entidades primordiales de nuestro proceso de desarrollo, pero nuestro proceso es en una sola dirección. Generamos código a partir de un modelo de PSM, y este modelo es generado a partir de un modelo de PIM, pero no hemos desarrollado la transformación contraria. En las diferentes fases de desarrollo de un proyecto software es usual que el conocimiento sobre el problema cambie. Por lo tanto, tener un proceso biyectivo nos permitiría mantener una consistencia entre los distintos modelos del sistema.

En relación con el proceso MDD, planificamos desarrollar un lenguaje gráfico de dominio específico (*Graphical Domain Specific Language*) para Pineapple y MalacaTiny. Actualmente, modelamos el sistema utilizando las herramientas proporcionadas por el marco de trabajo para modelado de Eclipse (EMF por sus siglas en inglés, *Eclipse Modeling Framework*), pero esto puede ser complicado para desarrolladores que no están familiarizados con este. La gran ventaja del EMF es que proporciona herramientas listas para ser usadas, que nos permiten manipular e instanciar nuestro metamodelo. Los lenguajes gráficos basados en metamodelos Ecore (como es nuestro caso), son desarrollados utilizando el marco de trabajo para gráficos de Eclipse (GMF por *Graphical Modeling Framework*). Hay una relación muy fuerte entre el metamodelo Ecore y su sintaxis gráfica, por lo tanto cualquier modificación en el metamodelo hará inservible la notación gráfica que hayamos proporcionado. Si usamos simplemente las herramientas del EMF, este problema no existe. Por lo tanto, antes de proporcionar un lenguaje gráfico, hemos esperado a tener una versión más estable de nuestro metamodelo. De hecho, es en este momento cuando nos planteamos acometer esta tarea. Otra ampliación para nuestro proceso MDD es agregar un nivel CIM.

El desarrollo del "Museo de la Informática" ha sido un gran reto que ha inspirado algunas ideas para extender la plataforma de agentes *Sol*, y hacer de ella una solución factible para desarrollar aplicaciones AmI reales. Algunas ideas están actualmente en desarrollo, mientras que otras están planeadas como trabajo futuro.

El rendimiento es una preocupación constante cuando se está trabajando con tecnologías de dispositivos ligeros. Como hemos mencionado anteriormente, esta-

mos mejorando tanto la implementación de *Sol* como la de los agentes MalacaTiny para hacer el proceso de codificación de mensajes (el aspecto de *Representación*) más eficiente. Somos conscientes de que nuestra propuesta incluye una sobrecarga que no se da en otras tecnologías de agentes, así que estamos trabajando para hacerla lo más ligera posible. Para llevar a cabo esta tarea, estamos usando herramientas de perfilado de consumo de recursos y de realización pruebas, tales como las proporcionadas por el entorno de desarrollo de Android.

A lo largo de esta tesis, hemos resaltado que las aplicaciones AmI deben de poder ejecutarse en una gran diversidad de dispositivos, con un conjunto variable de recursos computacionales y servicios disponibles. Para poder dar soporte a un número mayor de tecnologías, estamos extendiendo nuestra propuesta para motas sensoras Libellium Waspmotes. Esto ha sido necesario debido a que las motas Sun SPOT ofrecen un número limitado de capacidades sensoras (luz, ruido, sonido, aceleración y temperatura). Estas nuevas motas tienen menos recursos computacionales que Sun SPOT y a pesar de que usan ZigBee, no pueden utilizar la interfaz de red utilizada por las Sun SPOT. Por lo tanto, estamos dotando a la plataforma *Sol* con los medios necesarios para interactuar con estas motas, considerando sus reducidos recursos computacionales. Concretamente, estamos adaptando un agente MalacaTiny para motas MicaZ.

La distribución espacial del "Museo de la Informática" ha supuesto otro reto para nosotros. Como se menciona en el Capítulo 3, las diferentes salas del museo están distribuidas en pasillos y estancias a lo largo del edificio de la "E.T.S.I. de Informática". El problema surge cuando intentamos comunicar dispositivos que usan comunicaciones de corto alcance y necesitan comunicarse con el ordenador en el que *Sol* se está ejecutando. Nuestra solución inicial ha sido introducir encaminadores Meshlium[1] de Libellium, que permiten conectar dispositivos personales que usan WiFi y Bluetooth, y Libellium Waspmotes. Sin embargo, esta es una solución cara, por lo tanto estamos trabajando en una versión distribuida de la plataforma que tenga la capacidad de mantener nodos remotos. Estos nodos están siendo implementados como un nuevo tipo de conexión para el IMTP de *Sol*.

Finalmente, planeamos integrar en *Sol* un mecanismo más fléxible para la definición de grupos y su gestión. Capacidades como la auto-gestión y la sensibili-

[1] `http://www.libelium.com/products/meshlium`

dad al contexto tienen una gran prioridad para la AmI. Así que nuestro objetivo es integrar estas propiedades en los grupos. En la implementación actual de *Sol*, los agentes se unen a los grupos utilizando un identificador que conocen previamente, pero en algunas situaciones este mecanismo puede ser muy rígido. Por ejemplo, un agente puede necesitar unirse a un grupo con una característica en concreto (e.g. un grupo cuyo número de miembros sea inferior a 10, un grupo compuesto sólo por mujeres, un grupo de anglo-parlantes,...) pero no conoce su identificador y por lo tanto para este agente sería imposible formar parte de un grupo así.

Por otro lado, queremos facilitar que los agentes puedan unirse a ciertos grupos una vez han sido creados. Por ejemplo, en el Museo, puede ser creado un grupo de estudiantes extranjeros para informarles de que acaban de recibir un regalo. Así que los *VisitorAgent* que están proporcionando sus servicios a estos estudiantes, pasarían a formar parte automáticamente de este grupo tan pronto como fuera creado. Además, tenemos planeado extender el GMS para ofrecer la posibilidad de hacer consultas sobre los grupos que están registrados en la plataforma. Sin embargo, extender este servicio no es una tarea trivial debido a que debemos extender las descripciones de grupos (i.e. incluir las propiedades que caracterizan a un grupo) y definir una ontología con los términos necesarios para la AmI y aplicaciones concretas.

References

Jorge Agüero, Miguel Rebollo, Carlos Carrascosa, and Vicente Julián. Agent design using model driven development. In Yves Demazeau, Juan Pavón, Juan M. Corchado, and Javier Bajo, editors, *7th International Conference on Practical Applications of Agents and Multi-Agent Systems (PAAMS 2009)*, volume 55 of *Advances in Intelligent and Soft Computing*, pages 60–69. Springer Berlin Heidelberg, 2009. ISBN 978-3-642-00486-5. doi: 10.1007/978-3-642-00487-2_7. 2, 4, 25, 32, 34, 35, 180, 182

Francesco Aiello, Giancarlo Fortino, Antonio Guerrieri, and Raffaele Gravina. Maps: a mobile agent platform for wsns based on java sun spots. In *Proceedings of the third international workshop on Agent Technology for Sensor Networks (ATSN)*, 2009. 2, 180

Francesco Aiello, Giancarlo Fortino, Raffaele Gravina, and Antonio Guerrieri. A java-based agent platform for programming wireless sensor networks. *The Computer Journal*, 54(3):439–454, 2011. 26

Mercedes Amor and Lidia Fuentes. Malaca: A component and aspect-oriented agent architecture. *Information and Software Technology*, 51(6):1052 – 1065, 2009. ISSN 0950-5849. doi: 10.1016/j.infsof.2008.12.004. 67, 68, 91, 112, 141

Mercedes Amor, Lidia Fuentes, and Antonio Vallecillo. Bridging the gap between agent-oriented design and implementation using mda. In James Odell, Paolo Giorgini, and Jörg Müller, editors, *Agent-Oriented Software Engineering V*, volume 3382 of *Lecture Notes in Computer Science*, pages 93–108. Springer Berlin

Heidelberg, 2005. ISBN 978-3-540-24286-4. doi: 10.1007/978-3-540-30578-1_7. 30

Mercedes Amor, Inmaculada Ayala, and Lidia Fuentes. A4vanet: una aplicación basada en agentes jade-leap para redes vanet. In *XIII Conferencia de la Asociación Española para la Inteligencia Artificial. CAEPIA TTIA 2009. Sevilla, 9-13 de Noviembre de 2009. Actas*, pages 561–570. Asociación Española para la Inteligencia Artificial, February 2009. ISBN 978-84-692-6424-9. 13, 192

Mercedes Amor, Inmaculada Ayala, and Lidia Fuentes. A4vanet: Context-aware jade-leap agents for vanets. In Yves Demazeau, Frank Dignum, Juan M. Corchado, and Javier Bajo Pérez, editors, *Advances in Practical Applications of Agents and Multiagent Systems*, volume 70 of *Advances in Intelligent and Soft Computing*, pages 279–284. Springer Berlin Heidelberg, 2010. ISBN 978-3-642-12383-2. doi: 10.1007/978-3-642-12384-9_33. 13, 192

Juan Carlos Augusto and Chris D. Nugent. The use of temporal reasoning and management of complex events in smart homes. In *Proceedings of the European Conference on Artificial Intelligence, 2004*, pages 778–782, 2004. 22

Inmaculada Ayala, Mercedes Amor, and Lidia Fuentes. A model driven development of platform-neutral agents. In Jürgen Dix and Cees Witteveen, editors, *Multiagent System Technologies*, volume 6251 of *Lecture Notes in Computer Science*, pages 3–14. Springer Berlin Heidelberg, 2010a. ISBN 978-3-642-16177-3. doi: 10.1007/978-3-642-16178-0_3. 12

Inmaculada Ayala, Mercedes Amor, and Lidia Fuentes. Towards the automatic derivation of malaca agents using mde. In *Wiebe van der Haek, et. al. (eds.). The Eleventh International Workshop on agent oriented software engineering. AOSE 2010. Toronto, Canada, 10 of May 2010*, pages 61–72, May 2010b. 12

Inmaculada Ayala, Mercedes Amor, and Lidia Fuentes. Autonomic agents for mobile ambient assisted living applications. In *José Bravo; Diego López-de-Piña; Sergio Ochoa y Jesús Favela (Eds.). 5th International symposium on ubiquitous computing and ambient intelligence. UCAmI 2011. Conference proceedings. Riv-*

iera Maya, Mexico, december 5-9, December 2011a. ISBN 978-84-694-9677-0. 12, 13, 191, 192

Inmaculada Ayala, MercedesAmor Pinilla, and Lidia Fuentes. Modeling context-awareness in agents for ambient intelligence: An aspect-oriented approach. In Luis Antunes and H.Sofia Pinto, editors, *Progress in Artificial Intelligence*, volume 7026 of *Lecture Notes in Computer Science*, pages 29–43. Springer Berlin Heidelberg, 2011b. ISBN 978-3-642-24768-2. doi: 10.1007/978-3-642-24769-9_3. 12, 191

Inmaculada Ayala, Mercedes Amor, and Lidia Fuentes. An agent platform for self-configuring agents in the internet of things. In *Proceedings of the Thirds International Workshop on Infrastructures and Tools for Multiagent Systems. IT-MAS 2012. June 5, 2012 Valencia, Spain*, pages 65–78. Universidad Politècnica de València, June 2012a. ISBN 978-84-8363-850-7. 13, 192, 193

Inmaculada Ayala, Mercedes Amor, and Lidia Fuentes. Self-management of ambient intelligence systems: a pure agent-based approach. In *Proceedings of the 11th International Conference on Autonomous Agents and Multiagent Systems - Volume 3*, AAMAS '12, pages 1427–1428, Richland, SC, 2012b. International Foundation for Autonomous Agents and Multiagent Systems. ISBN 0-9817381-3-3, 978-0-9817381-3-0. 13, 23, 192

Inmaculada Ayala, Mercedes Amor, and Lidia Fuentes. Self-starmas: A multi-agent system for the self-management of aal applications. In *Proceedings of the 2012 Sixth International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing*, IMIS '12, pages 901–906, Washington, DC, USA, 2012c. IEEE Computer Society. ISBN 978-0-7695-4684-1. doi: 10.1109/IMIS.2012.28. 13, 192

Inmaculada Ayala, MercedesAmor Pinilla, and Lidia Fuentes. Exploiting dynamic weaving for self-managed agents in the iot. In Ingo J. Timm and Christian Guttmann, editors, *Multiagent System Technologies*, volume 7598 of *Lecture Notes in Computer Science*, pages 5–14. Springer Berlin Heidelberg, 2012d. ISBN 978-3-642-33689-8. doi: 10.1007/978-3-642-33690-4_3. 13, 192

Inmaculada Ayala, Mercedes Amor, and Lidia Fuentes. Self-configuring agents for ambient assisted living applications. *Personal and Ubiquitous Computing*, 17 (6):1159–1169, 2013a. ISSN 1617-4909. doi: 10.1007/s00779-012-0555-9. 12, 13, 191, 192

Inmaculada Ayala, Mercedes Amor, and Lidia Fuentes. A model driven engineering process of platform neutral agents for ambient intelligence devices. *Autonomous Agents and Multi-Agent Systems*, pages 1–42, 2013b. ISSN 1387-2532. doi: 10.1007/s10458-013-9223-3. 5, 12, 184, 191

Inmaculada Ayala, Mercedes Amor Pinilla, and Lidia Fuentes. Enhancing agent metamodels with self-management for ami environments. In Luís Correia, Luís Paulo Reis, and José Cascalho, editors, *Progress in Artificial Intelligence*, volume 8154 of *Lecture Notes in Computer Science*, pages 420–431. Springer Berlin Heidelberg, 2013c. ISBN 978-3-642-40668-3. doi: 10.1007/978-3-642-40669-0_36. 12, 191

Fabio Bellifemine, Agostino Poggi, and Giovanni Rimassa. Jade: a fipa 2000 compliant agent development environment. In *Proceedings of the fifth international conference on Autonomous agents*, AGENTS '01, pages 216–217, New York, NY, USA, 2001. ACM. ISBN 1-58113-326-X. doi: 10.1145/375735.376120. 2, 6, 136, 180, 184

Fabio Bellifemine, Giovanni Caire, Agostino Poggi, and Giovanni Rimassa. Jade: A software framework for developing multi-agent applications. lessons learned. *Information and Software Technology*, 50(1):10 – 21, 2008. ISSN 0950-5849. doi: 10.1016/j.infsof.2007.10.008. 24

Federico Bergenti and Agostino Poggi. Leap: A fipa platform for handheld and mobile devices. In John-JulesCh. Meyer and Milind Tambe, editors, *Intelligent Agents VIII*, volume 2333 of *Lecture Notes in Computer Science*, pages 436–446. Springer Berlin Heidelberg, 2002. ISBN 978-3-540-43858-8. doi: 10.1007/3-540-45448-9_33. 2, 180

Carole Bernon, Massimo Cossentino, Marie-Pierre Gleizes, Paola Turci, and Franco Zambonelli. A study of some multi-agent meta-models. In James Odell, Paolo

Giorgini, and Jörg P. Müller, editors, *Agent-Oriented Software Engineering V*, volume 3382 of *Lecture Notes in Computer Science*, pages 62–77. Springer Berlin Heidelberg, 2005. ISBN 978-3-540-24286-4. doi: 10.1007/978-3-540-30578-1_5. 7, 32, 56, 57, 185

Ghassan Beydoun, Graham Low, Brian Henderson-Sellers, Haralambos Mouratidis, Jorge Jesús Gómez Sanz, Juan Pavón, and Cesar Gonzalez-Perez. Faml: A generic metamodel for mas development. *IEEE Transactions on Software Engineering*, 35(6):841–863, 2009. ISSN 0098-5589. doi: 10.1109/TSE.2009.34. 4, 7, 32, 182, 185

Lars Braubach, Alexander Pokahr, and Winfried Lamersdorf. Jadex: A bdi-agent system combining middleware and reasoning. In Rainer Unland, Monique Calisti, and Matthias Klusch, editors, *Software Agent-Based Applications, Platforms and Development Kits*, Whitestein Series in Software Agent Technologies, pages 143–168. Birkhäuser Basel, 2005. ISBN 978-3-7643-7347-4. doi: 10.1007/3-7643-7348-2_7. 25, 32

Stefano Bromuri, Michael Schumacher, and Kostas Stathis. Towards distributed agent environments for pervasive healthcare. In Jürgen Dix and Cees Witteveen, editors, *Multiagent System Technologies*, volume 6251 of *Lecture Notes in Computer Science*, pages 125–137. Springer Berlin / Heidelberg, 2010. ISBN 978-3-642-16177-3. 150

Arnaud Brossard, Mourad Abed, and Christophe Kolski. Taking context into account in conceptual models using a model driven engineering approach. *Inf. Softw. Technol.*, 53(12):1349–1369, December 2011. ISSN 0950-5849. doi: 10.1016/j.infsof.2011.06.011. 34

Giovanni Caire, Nicolas Lhuillier, and Giovanni Rimassa. A communication protocol for agents on handheld devices. In *Proceedings of the First International Workshop on Ubiquitous Agents on Embedded, Wearable, and Mobile Devices*, 2002. 23

Bo Chen and Harry H. Cheng. A review of the applications of agent technology in traffic and transportation systems. *IEEE Transactions on Intel-*

ligent Transportation Systems, 11(2):485–497, 2010. ISSN 1524-9050. doi: 10.1109/TITS.2010.2048313. 22

Beo Model Driven Company. Acceleo: transforming models into code. `http://www.eclipse.org/acceleo/`, 2013. 36, 40

Diane Cook and Sajal Das. *Smart Environments: Technology, Protocols and Applications (Wiley Series on Parallel and Distributed Computing)*. Wiley-Interscience, 2004. ISBN 0471544485. 20

Diane Cook, Michael Youngblood, and Sajal Das. A multi-agent approach to controlling a smart environment. In Juan Augusto and Chris Nugent, editors, *Designing Smart Homes*, volume 4008 of *Lecture Notes in Computer Science*, pages 165–182. Springer Berlin / Heidelberg, 2006. ISBN 978-3-540-35994-4. 5, 23, 183

Diane J. Cook, Juan C. Augusto, and Vikramaditya R. Jakkula. Ambient intelligence: Technologies, applications, and opportunities. *Pervasive and Mobile Computing*, 5(4):277 – 298, 2009. ISSN 1574-1192. doi: 10.1016/j.pmcj.2009.04.001. 4, 18, 182

Juan M. Corchado, Javier Bajo, and Ajith Abraham. Gerami: Improving healthcare delivery in geriatric residences. *IEEE Intelligent Systems*, 23(2):19–25, 2008. ISSN 1541-1672. doi: 10.1109/MIS.2008.27. 19

Nokia Developer. Nokia n96. `http://developer.nokia.com/Devices/Device_specifications/N96/`, 2013a. 150

Nokia Developer. Nokia 5630 xpressmusic. `http://developer.nokia.com/Devices/Device_specifications/5630_XpressMusic/`, 2013b. 150

Simon Dobson, Roy Sterritt, P. Nixon, and M. Hinchey. Fulfilling the vision of autonomic computing. *Computer*, 43(1):35–41, 2010. ISSN 0018-9162. doi: 10.1109/MC.2010.14. 28, 103

Ken Ducatel, Marc Bogdanowicz, Fabiana Scapolo, Jos Leijten, and Jean-Claude Burgelman. Scenarios for ambient intelligence in 2010. Technical Report IPTS-Seville, IST Advisory Group, February 2001. 3, 17, 181

Ken Ducatel, Marc Bogdanowicz, Fabiana Scapolo, Jos Leijten, and Jean-Claude Burgelman. Ambient intelligence: From vision to reality. Technical report, IST Advisory Group, 2003. 3, 181

Wikipedia The Free Encyclopedia. Htc desire. `http://en.wikipedia.org/wiki/HTC_Desire`, 2013. 150

Robert Filman, Tzilla Elrad, Siobhán Clarke, and Mehmet Aksit. *Aspect-oriented software development*. Addison-Wesley Professional, first edition, 2004. ISBN 0321219767. 2, 41

Chien-Liang Fok, G. Roman, and Chenyang Lu. Mobile agent middleware for sensor networks: an application case study. In *Information Processing in Sensor Networks, 2005. IPSN 2005. Fourth International Symposium on*, pages 382–387, 2005. doi: 10.1109/IPSN.2005.1440953. 26

Foundation for Intelligent Physical Agents. Fipa abstract architecture specification. Technical Report SC00001L, Foundation for Intelligent Physical Agents, Geneva, Switzerland, December 2002. 128

The Eclipse Foundation. Ecore Tools. `http://wiki.eclipse.org/index.php/Ecore_Tools`, 2013a. 36

The Eclipse Foundation. Jet: Model to text transformations. `www.eclipse.org/emft/projects/jet/`, 2013b. 40

David Franklin. Cooperating with people: the intelligent classroom. In *Proceedings of the fifteenth national/tenth conference on Artificial intelligence/Innovative applications of artificial intelligence*, AAAI '98/IAAI '98, pages 555–560, Menlo Park, CA, USA, 1998. American Association for Artificial Intelligence. ISBN 0-262-51098-7. 20

Christopher Frantz, Mariusz Nowostawski, and Martin K. Purvis. Augmenting android with aose principles for enhanced functionality reuse in mobile applications. In Francien Dechesne, Hiromitsu Hattori, Adriaan Mors, Jose Miguel Such, Danny Weyns, and Frank Dignum, editors, *Advanced Agent Technology*, volume 7068 of *Lecture Notes in Computer Science*, pages 187–211.

Springer Berlin Heidelberg, 2012. ISBN 978-3-642-27215-8. doi: 10.1007/978-3-642-27216-5_13. 25

Peter Friese. Getting started with Code Generation with Xpand. http://www.peterfriese.de/getting-started-with-code-generation-with-xpand/, March 2010. 40

José M. Gascueña, Elena Navarro, and Antonio Fernández-Caballero. Model-driven engineering techniques for the development of multi-agent systems. *Engineering Applications of Artificial Intelligence*, 25(1):159 – 173, 2012. ISSN 0952-1976. doi: 10.1016/j.engappai.2011.08.008. 9, 34, 188

Paolo Giorgini, John Mylopoulos, Anna Perini, and Angelo Susi. The tropos metamodel and its use. *Informatica, An International Journal of Computing and Informatics*, 29(4):251–273, Noviembre 2005. ISSN 0350–5596. 30, 32, 34

F. Gouaux, L. Simon-Chautemps, J. Fayn, S. Adami, M. Arzi, D. Assanelli, M.C. Forlini, C. Malossi, A. Martinez, J. Placide, G.L. Ziliani, and P. Rubel. Ambient intelligence and pervasive systems for the monitoring of citizens at cardiac risk: New solutions from the epi-medics project. In *Computers in Cardiology, 2002*, pages 289–292, 2002. doi: 10.1109/CIC.2002.1166765. 20

Dominic Greenwood and Giovanni Rimassa. Autonomic goal-oriented business process management. In *ICAS '07*, pages 43–48. IEEE Computer Society, 2007. ISBN 0-7695-2859-5. 33

Object Management Group. Object Constraint Language (OCL). http://www.omg.org/spec/OCL/, 2013a. 38

Object Management Group. OMG Formal Versions of QVT. http://www.omg.org/spec/QVT/, 2013b. 38

Erico Guizzo. How google's self-driving car works. *IEEE Spectrum Online, October*, 18, 2011. 3, 181

Arno Haase, Markus Völter, Sven Efftinge, and Bernd Kolb. Introduction to openarchitectureware 4.1.2. In *MDD Tool Implementers Forum*, 2007. 36

Dianne Hackborn. Service api changes starting with android 2.0. http://android-developers.blogspot.com/2010/02/service-api-changes-starting-with.html, 2010. 150

Hani Hagras, Victor Callaghan, Martin Colley, Graham Clarke, Anthony Pounds-Cornish, and Hakan Duman. Creating an ambient-intelligence environment using embedded agents. *IEEE Intelligent Systems*, 19(6):12–20, 2004. ISSN 1541-1672. doi: 10.1109/MIS.2004.61. 3, 19, 181

Christian Hahn, Cristian Madrigal-Mora, and Klaus Fischer. A platform-independent metamodel for multiagent systems. *Autonomous Agents and Multi-Agent Systems*, 18(2):239–266, 2009. ISSN 1387-2532. doi: 10.1007/s10458-008-9042-0. 4, 7, 31, 34, 182, 186

Karen Zita Haigh, Liana M. Kiff, and Geoffrey Ho. The independent lifestyle assistant: Lessons learned. *Assistive Technology: The Official Journal of RESNA*, 18(1):87–106, 2006. doi: 10.1080/10400435.2006.10131909. 22

Anthony Harrington and Vinny Cahill. Model-driven engineering of planning and optimisation algorithms for pervasive computing environments. *Pervasive and Mobile Computing*, 7(6):705 – 726, 2011. ISSN 1574-1192. doi: 10.1016/j.pmcj.2011.09.005. 9, 145, 188

Paul Horn. Autonomic computing: IBM's Perspective on the State of Information Technology, 2001. 29

Nick Howden, Ralph Rönnquist, Andrew Hodgson, and Andrew Lucas. Jack intelligent agents-summary of an agent infrastructure. In *5th International conference on autonomous agents*, 2001. 24

Markus C. Huebscher and Julie A. McCann. A survey of autonomic computing-degrees, models, and applications. *ACM Comput. Surv.*, 40(3):7:1–7:28, August 2008. ISSN 0360-0300. doi: 10.1145/1380584.1380585. URL http://doi.acm.org/10.1145/1380584.1380585. 28

Google Inc. Galaxy nexus. http://www.android.com/devices/detail/galaxy-nexus, 2013. 160

A. Janik and K. Zielinski. Aaop-based dynamically reconfigurable monitoring system. *Information and Software Technology*, 52(4):380–396, 2010. 108

Frèdèric Jouault, Freddy Allilaire, Jean Bèzivin, and Ivan Kurtev. Atl: A model transformation tool. *Science of Computer Programming*, 72(1 - 2):31 – 39, 2008. ISSN 0167-6423. doi: 10.1016/j.scico.2007.08.002. 36

Stephen Keegan, Gregory M. P. O'Hare, and Michael J. O'Grady. Easishop: Ambient intelligence assists everyday shopping. *Information Sciences*, 178(3):588 – 611, 2008. ISSN 0020-0255. 20, 23

Jeffrey O. Kephart. Research challenges of autonomic computing. In *Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on*, pages 15–22, 2005. doi: 10.1109/ICSE.2005.1553533. 28

Jeffrey O. Kephart and David M. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, 2003. ISSN 0018-9162. doi: 10.1109/MC.2003.1160055. 2, 103, 180

Jeffrey O. Kephart and William E. Walsh. An artificial intelligence perspective on autonomic computing policies. In *Fifth IEEE International Workshop on Policies for Distributed Systems and Networks, 2004. POLICY 2004.*, pages 3–12, 2004. doi: 10.1109/POLICY.2004.1309145. 30, 46, 107

Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *ECOOP'97 - Object-Oriented Programming*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer Berlin Heidelberg, 1997. ISBN 978-3-540-63089-0. doi: 10.1007/BFb0053381. URL `http://dx.doi.org/10.1007/BFb0053381`. 41

Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of aspectj. In Jürgen Lindskov Knudsen, editor, *ECOOP 2001 - Object-Oriented Programming*, volume 2072 of *Lecture Notes in Computer Science*, pages 327–354. Springer Berlin Heidelberg, 2001. ISBN 978-3-540-42206-8. doi: 10.1007/3-540-45337-7_18. URL `http://dx.doi.org/10.1007/3-540-45337-7_18`. 41

Benjamin Klatt. Xpand: A closer look at the model2text transformation language. In *12th European Conference on Software Maintenance and Reengineering*, 2008. 36

Fernando Koch, John-Jules C. Meyer, Frank Dignum, and Iyad Rahwan. Programming deliberative agents for mobile services: The 3apl-m platform. In RafaelH. Bordini, MehdiM. Dastani, Jrgen Dix, and Amal Fallah Seghrouchni, editors, *Programming Multi-Agent Systems*, volume 3862 of *Lecture Notes in Computer Science*, pages 222–235. Springer Berlin Heidelberg, 2006. ISBN 978-3-540-32616-8. 25

Young Min Kwon, Sameer Sundresh, Kirill Mechitov, and Gul Agha. Actornet: an actor platform for wireless sensor networks. In *Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems*, AAMAS '06, pages 1297–1300, New York, NY, USA, 2006. ACM. ISBN 1-59593-303-4. doi: 10.1145/1160633.1160871. 26

Oracle Labs. Sun SPOT World. `http://www.sunspotworld.com/`, 2013. 10, 189

Edward Lank, Amy Ichnowski, and Shalid Khatri. Zero knowledge access to a smart classroom environment. In *Proceedings of the Workshop on Ubiquitous Display Environments*, 2004. 20

Mikko Laukkanen, Sasu Tarkoma, and Jani Leinonen. Fipa-os agent platform for small-footprint devices. In John-JulesCh. Meyer and Milind Tambe, editors, *Intelligent Agents VIII*, volume 2333 of *Lecture Notes in Computer Science*, pages 447–460. Springer Berlin Heidelberg, 2002. ISBN 978-3-540-43858-8. doi: 10.1007/3-540-45448-9_34. 2, 25, 180

Till C. Lech and Leendert W. M. Wienhofen. Ambieagents: a scalable infrastructure for mobile and context-aware information services. In *Proceedings of the fourth international joint conference on Autonomous agents and multiagent systems*, AAMAS '05, pages 625–631, New York, NY, USA, 2005. ACM. ISBN 1-59593-093-0. 150

Ramon Lopes, Flávio Assis, and Carlos Montez. Maspot: A mobile agent system for sun spot. In *10th International Symposium on Autonomous Decentralized Systems (ISADS 2011)*, pages 25 –31, march 2011. 26

Goreti Marreiros, Ricardo Santos, Carlos Ramos, José Neves, Paulo Novais, José Machado, and José Bulas-Cruz. Ambient intelligence in emotion based ubiquitous decision making. In Juan Carlos Augusto and D. Shapiro, editors, *of the Second Workshop on Artificial Intelligence Techniques for Ambient Intelligence, 2007*, pages 86–91, 2007. URL `http://repositorium.sdum.uminho.pt/handle/1822/18984`. 20

Samsung Mobile. Samsung galaxy s. `http://www.samsung.com/global/microsite/galaxys/specification/spec.html?ver=high`, 2013. 150

Ambra Molesini, Enrico Denti, and Andrea Omicini. Homemanager: Testing agent-oriented software engineering in home intelligence. In *Agents and Artificial Intelligence*, volume 67 of *Communications in Computer and Information Science*, pages 205–218. Springer Berlin Heidelberg, 2010. ISBN 978-3-642-11818-0. 32

Bill Moore, David Dean, Anna Gerber, Gunnar Wagenknecht, and Philippe Vanderheyden. *Eclipse Development*. IBM Corporation, International Technical Support Organization, 2004. 36

Gianluca Morganti, Anna-Maria Perdon, Giuseppe Conte, and David Scaradozzi. Multi-agent system theory for modelling a home automation system. In *Bio-Inspired Systems: Computational and Ambient Intelligence*, volume 5517 of *LNCS*, pages 585–593. Springer Berlin Heidelberg, 2009. ISBN 978-3-642-02477-1. 32

Miguel A. Muñoz, Marcela Rodríguez, Jesus Favela, Ana I. Martinez-Garcia, and Victor M. González. Context-aware mobile communication in hospitals. *Computer*, 36(9):38–46, September 2003. ISSN 0018-9162. 5, 23

Conor Muldoon, Gregory O'Hare, Rem Collier, and Michael O'Grady. Agent factory micro edition: A framework for ambient applications. In Vassil Alexandrov,

Geert van Albada, Peter Sloot, and Jack Dongarra, editors, *Computational Science ICCS 2006*, volume 3993 of *Lecture Notes in Computer Science*, pages 727–734. Springer Berlin / Heidelberg, 2006. 6, 25, 150, 184

Conor Muldoon, Gregory O'Hare, Michael J. O'Grady, and Richard Tynan. Agent migration and communication in wsns. In *Ninth International Conference on Parallel and Distributed Computing, Applications and Technologies, 2008. PDCAT 2008.*, pages 425 –430, dec. 2008. 25

Jürgen Nehmer, Martin Becker, Arthur Karshmer, and Rosemarie Lamm. Living assistance systems: an ambient intelligence approach. In *Proceedings of the 28th international conference on Software engineering*, ICSE '06, pages 43–50, New York, NY, USA, 2006. ACM. ISBN 1-59593-375-1. doi: 10.1145/1134285.1134293. 19

Oracle. Java ME and Java Card Technology. `http://www.oracle.com/technetwork/java/javame/index.html`, 2013. 10, 189

Oscar Pastor, Sergio España, José Ignacio Panach, and Nathalie Aquino. Model-driven development. *Informatik-Spektrum*, 31(5):394–407, 2008. ISSN 0170-6012. doi: 10.1007/s00287-008-0275-8. 31

Juan Pavón, Jorge Gómez-Sanz, and Rubén Fuentes. Model driven development of multi-agent systems. In *Proceedings of the Second European conference on Model Driven Architecture: foundations and Applications*, ECMDA-FA'06, pages 284–298, Berlin, Heidelberg, 2006. Springer-Verlag. ISBN 3-540-35909-5, 978-3-540-35909-8. doi: 10.1007/11787044_22. URL `http://dx.doi.org/10.1007/11787044_22`. 30, 34

Juan Pavón, Jorge Gómez-Sanz, and Rubén Fuentes. Model driven development of multi-agent systems. In Arend Rensink and Jos Warmer, editors, *Model Driven Architecture, Foundations and Applications*, volume 4066 of *Lecture Notes in Computer Science*, pages 284–298. Springer Berlin Heidelberg, 2006. ISBN 978-3-540-35909-8. doi: 10.1007/11787044_22. 4, 182

Joaquin Peña, Michael G. Hinchey, Manuel Resinas, Roy Sterritt, and James L. Rash. Designing and managing evolving systems using a mas product line approach. *Sci. Comput. Program.*, 66(1):71–86, April 2007. ISSN 0167-6423. 33

Loris Penserini, Paolo Bresciani, Tsvi Kuflik, and Paolo Busetta. Using tropos to model agent based architectures for adaptive systems: a case study in ambient intelligence. In *Proceedings of IEEE International Conference onSoftware - Science, Technology and Engineering, 2005*, pages 37–46, 2005. doi: 10.1109/SWSTE.2005.23. 22, 23, 32

Alexander Pokhar. Jadex Android user guide. `http://jadex-agents.informatik.uni-hamburg.de/xwiki/bin/view/Android+User+Guide/01+Introduction`, 2013. 25

Martha E. Pollack, Laura Brown, Dirk Colbry, Cheryl Orosz, Bart Peintner, Sailesh Ramakrishnan, Sandra Engberg, Judith T. Matthews, Jacqueline Dunbar-Jacob, Colleen E. McCarthy, et al. Pearl: A mobile robotic assistant for the elderly. In *AAAI workshop on automation as eldercare*, volume 2002, pages 85–91, 2002. 3, 181

Andry Rakotonirainy and Richard Tay. In-vehicle ambient intelligent transport systems (i-vaits): towards an integrated research. In *Proceedings of the 7th International IEEE Conference on Intelligent Transportation Systems, 2004*, pages 648–651, 2004. doi: 10.1109/ITSC.2004.1398977. 20

Giuseppe Riva. Ambient intelligence in health care. *CyberPsychology & Behavior*, 6:295–300, July 2004. doi: 10.1089/109493103322011597. 20

Cesare Rocchi, Oliviero Stock, Massimo Zancanaro, Michael Kruppa, and Antonio Krüger. The museum visit: generating seamless personalized presentations on multiple devices. In *Proceedings of the 9th international conference on Intelligent user interfaces*, IUI '04, pages 316–318, New York, NY, USA, 2004. ACM. ISBN 1-58113-815-6. doi: 10.1145/964442.964517. 20

Stuart Jonathan Russell, Peter Norvig, John F. Canny, Jitendra M. Malik, and Douglas D. Edwards. *Artificial intelligence: a modern approach*, volume 74. Prentice hall, 1995. ISBN 0137903952. 29

Fariba Sadri. Ambient intelligence: A survey. *ACM Comput. Surv.*, 43(4):36:1–36:66, October 2011. ISSN 0360-0300. doi: 10.1145/1978802.1978815. 1, 4, 18, 21, 32, 179, 182

Nayat Sánchez-Pi, Javier Carbó, and José Molina. Jade/leap agents in an aml domain. In Emilio Corchado, Ajith Abraham, and Witold Pedrycz, editors, *Hybrid Artificial Intelligence Systems*, volume 5271 of *Lecture Notes in Computer Science*, pages 62–69. Springer Berlin / Heidelberg, 2008. ISBN 978-3-540-87655-7. 150

Andrea Santi, Marco Guidi, and Alessandro Ricci. Jaca-android: An agent-based platform for building smart mobile applications. In Mehdi Dastani, Amal Fallah Seghrouchni, Jomi Hbner, and Joo Leite, editors, *Languages, Methodologies, and Development Tools for Multi-Agent Systems*, volume 6822 of *Lecture Notes in Computer Science*, pages 95–114. Springer Berlin Heidelberg, 2011. ISBN 978-3-642-22722-6. doi: 10.1007/978-3-642-22723-3_6. 25

Bran Selic. The pragmatics of model-driven development. *IEEE Software*, 20(5): 19–25, 2003. ISSN 0740-7459. doi: 10.1109/MS.2003.1231146. 30

Weiming Shen, Sherman Y. T. Lang, and Lihui Wang. ishopfloor: an internet-enabled agent-based intelligent shop floor. *Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on*, 35(3):371–381, 2005. ISSN 1094-6977. doi: 10.1109/TSMCC.2004.843224. 20

Yuanchun Shi, Weikai Xie, Guangyou Xu, Runting Shi, Enyi Chen, Yanhua Mao, and Fang Liu. The smart classroom: merging technologies for seamless tele-education. *Pervasive Computing, IEEE*, 2(2):47–55, 2003. ISSN 1536-1268. doi: 10.1109/MPRV.2003.1203753. 20

Nikolaos Spanoudakis and Pavlos Moraitis. Modular jade agents design and implementation using aseme. In *2010 IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology (WI-IAT)*, volume 2, pages 221–228, 2010. doi: 10.1109/WI-IAT.2010.136. 34

Frank Sposaro and Gary Tyson. ifall: An android application for fall monitoring and response. In *Annual International Conference of the IEEE Engineering in*

Medicine and Biology Society, 2009. EMBC 2009, pages 6119–6122, 2009. doi: 10.1109/IEMBS.2009.5334912. 19

Thomas Stahl and Markus Völter. *Model-driven software development*. Wiley, first edition, May 2006. ISBN 978-0470025703. 1, 179

David Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework 2.0*. Addison-Wesley Professional, 2nd edition, 2009. ISBN 0321331885. 36

Oliviero Stock, Massimo Zancanaro, Paolo Busetta, Charles Callaway, Antonio Krüger, Michael Kruppa, Tsvi Kuflik, Elena Not, and Cesare Rocchi. Adaptive, intelligent presentation of information for the museum visitor in peach. *User Modeling and User-Adapted Interaction*, 17:257–304, 2007. ISSN 0924-1868. 5, 23, 183

Dante I. Tapia, Ajith Abraham, Juan M. Corchado, and Ricardo S. Alonso. Agents and ambient intelligence: case studies. *Journal of Ambient Intelligence and Humanized Computing*, 1(2):85–93, 2010. ISSN 1868-5137. doi: 10.1007/s12652-009-0006-2. 22

Maurice H. ter Beek, Steffania Gnesi, Carlo Montangero, and Laura Semini. Detecting policy conflicts by model checking UML state machines. In *ICFI X*, pages 59–74. IOS Press, 2009. 58, 59

Maurice H. ter Beek, Alessandro Fantechi, Stefania Gnesi, and Franco Mazzanti. A state/event-based model-checking approach for the analysis of abstract system properties. *Science of Computer Programming*, 76(2):119 – 135, 2011. ISSN 0167-6423. doi: 10.1016/j.scico.2010.07.002. 58, 61

Ivan Trencansky, Radovan Cervenka, and Dominic Greenwood. Applying a uml-based agent modeling language to the autonomic computing domain. In *OOPSLA'06*, pages 521–529. ACM, 2006. ISBN 1-59593-491-X. 33

Kenneth J. Turner, Stephan Reiff-Marganiec, Lynne Blair, Gavin A. Campbell, and Feng Wang. Appel: An adaptable and programmable policy environment

and language. Technical Report CSM-161, Computing Science and Mathematics, University of Stirling, April 2009. 9, 186

Radovan Červenka, Ivan Trenčanský, Monique Calisti, and Dominic Greenwood. Aml: Agent modeling language toward industry-grade agent-based modeling. In James Odell, Paolo Giorgini, and Jörg P. MÏler, editors, *Agent-Oriented Software Engineering V*, volume 3382 of *Lecture Notes in Computer Science*, pages 31–46. Springer Berlin Heidelberg, 2005. ISBN 978-3-540-24286-4. doi: 10.1007/978-3-540-30578-1_3. 33

Emil Vassev and Mike Hinchey. Assl: A software engineering approach to autonomic computing. *Computer*, 42(6):90–93, 2009. ISSN 0018-9162. 33

Meritxell Vinyals, Juan A. Rodriguez-Aguilar, and Jesus Cerquides. A survey on sensor networks from a multiagent perspective. *Computer Journal*, 54(3): 455–470, 2011. 26

Craig Walls and Ryan Breidenbach. *Spring in Action*. Manning Publications Co., 5th edition, 2005. ISBN 1932394354. xv, 41

Danny Weyns, Sam Malek, and Jesper Andersson. Forms: Unifying reference model for formal specification of distributed self-adaptive systems. *ACM Trans. Auton. Adapt. Syst.*, 7(1):8:1–8:61, May 2012. ISSN 1556-4665. 33