# MODELLING A UNIVERSITY AS A MULTI-AGENT SYSTEM

by

ÖNDER ÖZKAN

BS, Information Technologies, FMV IŞIK UNIVERSITY, 2005

BS, Computer Sciences and Engineering, FMV IŞIK UNIVERSITY, 2006

Submitted to the Graduate School of
Science and Engineering in partial fulfillment of
the requirements for the degree of
Master of Science
in
Information Technologies

FMV Işık University

2008

IŞIK UNIVERSITY

GRADUATE SCHOOL OF SCIENCE AND ENGINEERING

# MODELLING A UNIVERSITY AS A MULTI-AGENT SYSTEM

ÖNDER ÖZKAN

APPROVED BY :

| | |
|---|---|
| Assist. Prof. Nazım Ziya PERDAHÇI (Thesis Supervisor) | .................... |
| Assoc. Prof. Seyhun ALTUNBAY | .................... |
| Assist. Prof. Vedat COŞKUN | .................... |

DATE OF APPROVAL: 09.09.2008

# MODELLING A UNIVERSITY AS A MULTI-AGENT SYSTEM

## Abstract

Multi-Agent systems are an approach for designing and implementing complex social organizations in the software domain. In multi-agent systems; actors that interact with the system are modeled as software entities called "agents". These software agents can show autonomous and intelligent behaviour. Also when a supporting runtime environment (or platform) is available, agents show mobility and can move between interconnected platforms that form the system. With these properties agents try to satisfy the expectations of their respective owner actors by using their knowledge and belief bases (derived from the actors). An implementation of a multi-agent system can both be used for simulation purposes (by initialising agents based on reference test data) and as an actual information system by attaching agents to real world actors.

This work aims at analysing a university as a complex social system and creating a corresponding multi-agent model. In the designed model academic and administrative units and students are represented by intelligent and mobile agents that can be implemented using JADE multi-agent system development framework. To provide a satisfactory mobile environment faculties and administrative units are defined to have their own (separate) agent platforms.

# BİR ÜNİVERSİTENİN ÇOKLU-AJANLI SİSTEM OLARAK MODELLENMESİ

## Özet

Çoklu ajanlı sistemler (Multi-Agent Systems) karmaşık sosyal organizasyonların bilgisayar ortamında modellenmesine ve uygulanmasına yönelik kolaylıklar sağlayan bir yazılım gelistirme yaklaşımıdır. Çoklu ajanlı sistemlerde gerçek sistemde etkin olan aktörlerin yazılımsal karşılıkları "ajan" olarak adlandırılan yazılımlar ile sağlanır. Bu yazılımsal ajanlar otonom ve (dereceli olarak) zeki davranışlar gösterebilirler. Buna ek olarak desteklenen ortamlarda ajanlar hareketlilik (mobility) özelliği ile birbirine bağlı platformlar arasında dolaşabilirler. Bu özellikleri ile temsil ettikleri aktörlerin amaçlarına uygun şekilde karşılanması gereken fayda beklentilerini ilgili aktörlerin bilgi birikimleri ve inançları doğrultusunda gerçekleştirmeye çalışırlar. Çoklu ajanlı modeli hazırlanmış bir sistemin uygulaması hem (test bilgi birikimleri yüklenmiş ajanlar ile) sisteme yönelik simulasyon amaçları için kullanılabilir, hem de reel bilgiler ile yüklenerek ve/veya ajanlar gerçek kullanıcıların erişimine açılarak aktif bir bilgi sistemi olarak kullanılabilirler.

Bu çalışma ile karmaşık sosyal bir sistem olarak bir üniversitenin incelenmesi ve çoklu ajanlı modelinin oluşturulması amaçlanmıştır. Oluşturulan modelde üniversite içinde varolan akademik ve idari birimler ve öğrenciler JADE çoklu ajan geliştirme ortamında uygulanacak zeki ve mobil ajanlar ile temsil edilmektedirler. Mobilite için gerekli ortamı yaratmak adına üniversite içinde bulunan fakülte ve idari birimler ayrı ajan platformları olarak tanımlanmışlardır.

## Acknowledgements

I would like to thank my advisor for his support, mentorship and friendship during past 7 years we have been working together.

To family, friends and Snow White...

# Table of Contents

Abstract . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . ii

Özet . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . iii

Acknowledgements . . . . . . . . . . . . . . . . . . . . . . . . . . . . . iv

List of Tables . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . viii

List of Figures . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . ix

List of Symbols/Abbreviations . . . . . . . . . . . . . . . . . . . . . . xi

1. Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 1

    1.1. Motivation . . . . . . . . . . . . . . . . . . . . . . . . . . . . 2

    1.2. Objectives . . . . . . . . . . . . . . . . . . . . . . . . . . . . 3

2. Review on Agents and Agent Technology . . . . . . . . . . . . . . . . 4

    2.1. Agent Concept . . . . . . . . . . . . . . . . . . . . . . . . . . . 4

    2.2. Agent Architectures . . . . . . . . . . . . . . . . . . . . . . . . 6

    2.3. FIPA . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7

    2.4. JADE . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9

        2.4.1. Agent Model in JADE . . . . . . . . . . . . . . . . . . . . 9

        2.4.2. Agent Activities . . . . . . . . . . . . . . . . . . . . . . 10

        2.4.3. Communication Among Agents . . . . . . . . . . . . . . . . 12

        2.4.4. Content Languages and Ontologies . . . . . . . . . . . . . 16

        2.4.5. Agent Platforms and Containers . . . . . . . . . . . . . . 16

    2.5. Gaia Methodology for Developing Multi-Agent Systems . . . . . . . 18

2.5.1. Analysis Phase . . . . . . . . . . . . . . . . . . . . . . . . . 18
2.5.2. Design Phase . . . . . . . . . . . . . . . . . . . . . . . . . . 21
3. Modelling a University as a Multi-Agent System . . . . . . . . . . . . . . . . . 23
3.1. Model Definition . . . . . . . . . . . . . . . . . . . . . . . . . 23
3.1.1. Agent Platform and Container Definition . . . . . . . . . . . . 23
3.1.2. Agent Definitions . . . . . . . . . . . . . . . . . . . . . . . . 23
3.1.3. Model-Phase Agents . . . . . . . . . . . . . . . . . . . . . . . 24
3.1.3.1. Student-Agent . . . . . . . . . . . . . . . . . . . 24
3.1.3.2. Faculty-Agent . . . . . . . . . . . . . . . . . . . . 25
3.1.3.3. Administrative-Agent . . . . . . . . . . . . . . . . 25
3.1.3.4. Aggregate-Agent . . . . . . . . . . . . . . . . . . 26
3.1.3.5. Pseudo-Agent . . . . . . . . . . . . . . . . . . . . 26
3.1.4. Implementation-Phase Agents . . . . . . . . . . . . . . . . . . 26
3.1.4.1. MausAgent . . . . . . . . . . . . . . . . . . . . . 26
3.1.4.2. DataProviderAgent . . . . . . . . . . . . . . . . . 26
3.1.4.3. CoreServicesAgent . . . . . . . . . . . . . . . . . 26
3.1.4.4. JADECreatorAgent . . . . . . . . . . . . . . . . . 27
3.2. Implementation . . . . . . . . . . . . . . . . . . . . . . . . . . 27
3.2.1. Programming Implementation . . . . . . . . . . . . . . . . . . 27
3.2.2. Package Structure . . . . . . . . . . . . . . . . . . . . . . . . 27
3.2.2.1. edu.zion.mas.maus.core . . . . . . . . . . . . . . . 27
3.2.2.2. edu.zion.mas.maus.core.data.providers . . . . . . . . . 28
3.2.2.3. edu.zion.mas.maus.core.data.types . . . . . . . . . . 29
3.2.3. edu.zion.mas.maus.core.data.onto . . . . . . . . . . . . . . . . 29
3.2.4. edu.zion.mas.maus.agents . . . . . . . . . . . . . . . . . . . . 29
3.2.4.1. edu.zion.mas.maus.agents.actions . . . . . . . . . . . 31
3.2.4.2. edu.zion.mas.maus.behaviour . . . . . . . . . . . . . 31
3.2.4.3. edu.zion.mas.maus.kb.believes . . . . . . . . . . . . . 32
3.2.4.4. edu.zion.mas.maus.kb.rules . . . . . . . . . . . . . . 33

3.2.4.5. edu.zion.mas.maus.util . . . . . . . . . . . . . . . . . . . 34
3.2.4.6. edu.zion.mas.maus.util.management . . . . . . . . . . 34
3.2.4.7. edu.zion.mas.maus.util.ui . . . . . . . . . . . . . . . . . . 34
3.2.4.8. edu.zion.mas.maus.util.web . . . . . . . . . . . . . . . . 34
3.3. Data Storage . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 35
3.4. Deployment . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 35
4. Applications of this Model . . . . . . . . . . . . . . . . . . . . . . 37
4.1. Quota Forecasting . . . . . . . . . . . . . . . . . . . . . . . . . . 37
4.2. Online Course Registration . . . . . . . . . . . . . . . . . . . . . 38
4.3. Regulation Simulations . . . . . . . . . . . . . . . . . . . . . . . 38
4.4. Online Student-Faculty Collaboration System . . . . . . . . . . . . . 39
5. Sample Application . . . . . . . . . . . . . . . . . . . . . . . . . . 40
5.1. Description . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 40
5.2. Analysis . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 41
5.2.1. Roles Model . . . . . . . . . . . . . . . . . . . . . . . . . . . 41
5.2.2. Interactions Model . . . . . . . . . . . . . . . . . . . . . . . 46
5.3. Design . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 46
5.3.1. Agent Model . . . . . . . . . . . . . . . . . . . . . . . . . . . 46
5.3.2. Services Model . . . . . . . . . . . . . . . . . . . . . . . . . 47
5.3.2.1. StudentAgent Services . . . . . . . . . . . . . . . . . . 47
5.3.2.2. RegistrarAgent Services . . . . . . . . . . . . . . . . . 51
5.3.2.3. DataProviderAgent Services . . . . . . . . . . . . . . . 51
5.3.3. Acquaintance Model . . . . . . . . . . . . . . . . . . . . . . . 52
5.4. Implementation . . . . . . . . . . . . . . . . . . . . . . . . . . . . 53
6. Conclusion . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 54
References . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 56
References not Cited . . . . . . . . . . . . . . . . . . . . . . . . . . . 57
Curriculum Vitae . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 58

# List of Tables

Table 2.1. FIPA ACL MessageParameters [6] . . . . . . . . . . . . . . . . . . . . 14

Table 2.2. Gaia Operators for Liveness Formulas . . . . . . . . . . . . . . . . . 20

Table 2.3. Gaia Agent Instance Qualifiers . . . . . . . . . . . . . . . . . . . . . 22

Table 5.1. Gaia Interactions Model . . . . . . . . . . . . . . . . . . . . . . . . 48

Table 5.2. Gaia Services Model for Course Registration System (StudentAgent, Registrar Agent) . . . . . . . . . . . . . . . . . . . . . . . . 49

Table 5.3. Gaia Services Model for Course Registration System (DataProviderAgent) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 50

# List of Figures

Figure 2.1. An agent and its environment . . . . . . . . . . . . . . . . . . . . 5

Figure 2.2. Horizontal and Vertical Layered Architectures . . . . . . . . . . . 8

Figure 2.3. JADE - Agent thread path of execution[2] . . . . . . . . . . . . . 11

Figure 2.4. FIPA message structure . . . . . . . . . . . . . . . . . . . . . . . 13

Figure 2.5. JADE Agent Platform Architecture[2] . . . . . . . . . . . . . . . . 17

Figure 2.6. JADE Agent Platform Architecture Elements[2] . . . . . . . . . . 18

Figure 2.7. Gaia Framework Structure . . . . . . . . . . . . . . . . . . . . . . 19

Figure 2.8. Sample Gaia Liveness Property Definition . . . . . . . . . . . . . . 19

Figure 2.9. Gaia Analysis Process . . . . . . . . . . . . . . . . . . . . . . . . 21

Figure 2.10. Gaia Design Process . . . . . . . . . . . . . . . . . . . . . . . . 22

Figure 5.1. Gaia Agent Model for Course Registration System . . . . . . . . . 46

Figure 5.2. Gaia Acquaintance Model for Course Registration System . . . . . 52

## List of Symbols/Abbreviations

| | |
|---|---|
| AI | Artificial Intelligence |
| AMS | Agent Management System |
| AP | Agent Platform |
| DF | Directory Facilitator |
| FIPA | Foundation for Intelligent Physical Agents |
| FSM | Finite State Machine |
| IEEE | International Association of Electrical and Electronics Engineers |
| JADE | Java Agent Development Framework |
| JADE-WSIG | JADE Web Services Integration Gateway |
| MAM | Multi-Agent Model |
| MAS | Multi-Agent System |
| MAUS | Multi-Agent UniverSity |
| MTS | Message Transport Service |
| WSDL | Web Services Definition Language |

# 1. Introduction

Multi-Agent systems (MAS) are systems composed of multiple interacting computing elements, known as agents. Agents are computer systems with two important capabilities. First, they are at least to some extend capable of *autonomous action* - of deciding *for themselves* what they need to do in order to satisfy their design objectives. Second, they are capable of interacting with other agents - not simply by exchanging data, but by engaging in analogues of the kind of social activity that we all engage in every day of our lives: cooperation, coordination, negotiation and the like.[1]

MAS's can be used in modelling, designing and implementing complex and dynamic systems. Due to the intelligent and autonomous nature of the agents the resulting system can also be used for experimental simulation purposes. Multi-Agent systems can also be used for Multi-Agent modelling for observing social behaviour and effects of outside factors in artificial societies.

A university presents an interesting social organization for studying as a multi-agent system. The various actors in the system; administrative departments, academic departments, students, etc., can be represented by carefully designed intelligent agents. The interactions that happen between these actors can then be modeled as agent behaviours and the resulting system can be used for simulating the behaviour of the university during special periods (consider course registration period as an example) and when faced with certain scenarios (such as curriculum changes).

## 1.1. Motivation

The ability to forecast the outcomes of changes made to the dynamics of a university is a valuable tool for administrative bodies. A MAS representing a university can be used to provide such a forecasting tool. In such a system agents representing students will have internal rule definitions representing the beliefs (impressions about a certain instructor, the students ability in course subjects - mathematics, programming, etc. -, expectations - leaving a day of the week free, taking the same course with a friend - and constraints - satisfying a minimum course credit, taking outstanding failed courses - of students. A similar specialized agent representation will exist for instructors and administrative staff. Using this MAS, simulations and forecasts on system reflexes to situations such as;

1. demand on individual courses during registration periods
2. effects of changes on course curriculum
3. effects of changes on administrative workflows

On the other hand a MAS representing a university can be used to deploy information systems for applications such as;

1. online course registration system
2. online information gathering and communication system
3. online petition registration, processing and tracking system

## 1.2. Objectives

This work aims at providing a basis for a complete Multi-Agent model (as a Multi-Agent system) of a university. In the following chapters conceptual definition and actual implementation details of this model will be given along with sample applications. The objectives of this work can be summarized as;

1. provide a basis conceptual model of a university
2. map the conceptual model to a MAS
3. provide an actual implementation design of the MAS
4. provide a sample application for the implemented MAS

# 2. Review on Agents and Agent Technology

## 2.1. Agent Concept

An *agent* is a computer system that is *situated* in some *environment*, and that is capable of *autonomous action* in this environment in order to meet its design objectives. [1] An agent is in constant interaction with its environment. The environment affects the agent causing it to take actions and change its internal state. Likewise actions of an agent will affect the environment. When multiple agents exist in the same environment actions of one agent may have effects on all the agents. A very simple graphic representation of an agent an its environment is given in figure 2.1. Having said that agents can affect their environments, agents generally do not have total control over their environments, other entities (agents, external input sources, etc.) may be affecting the system. Thus, when an agent executes an action $A_1$ at time $t_0$, it can not expect the environment to respond in the same way when the same action is executed at time $t_1$. So agents must be prepared for failures (as in action results not meeting expectations). Consider a room temperature control system as an example. In this system the thermostat can be modeled as an agent. If the agent had total control over the environment (the environment was deterministic) running the heater units for a given period of time would have the same effect on the ambient room temperature regardless of the time this action was triggered. However in a real scenario due to

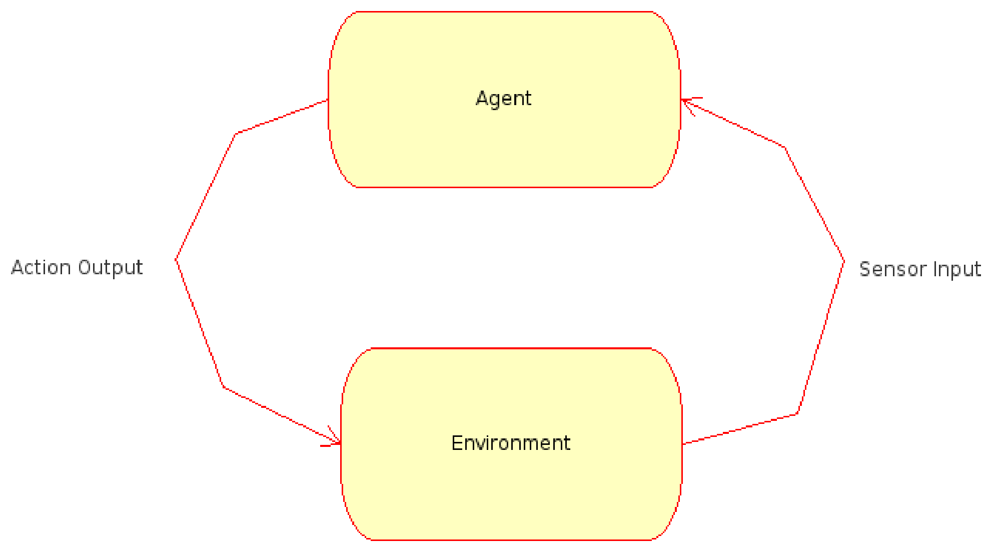

Figure 2.1. An agent and its environment

external factors like; the number of people in the room, having the windows and/or doors open, the time of the day, the same action will result in a different temperature change ( in other words the effort for keeping the temperature at a fixed point will vary from time to time). When designing and analyzing agents it is important to identify and characterize the environments. Agent environments can be characterized as;

1. Accessible vs. Inaccessible
2. Deterministic vs. Non-deterministic
3. Static vs. Dynamic
4. Discrete vs. Continuous

Accessible environments are environments where the agent has complete,accurate and upto date access to the state of the environment. Whereas in inaccessible environments the agents vision (or perception) of the environment is limited. In deterministic environments results of an action are the same regardless of the conditions (time, environment state, agent state, etc.) the action is executed. In non-deterministic environments the results of an action may vary depending on the conditions it is executed.

A static environment is an environment that can only be changed (or affected) by the agent operating in it. In contrast, dynamic environments have other processes operating on them and are open to changes that can result from sources other than the agent. In a discrete environment the set of actions that an agent can do and the set of responses that can be received from the environment are fixed and finite.

## 2.2. Agent Architectures

There are several approaches and architectures that can be used when designing or defining agents. Generally there is no single architecture that can be selected as the best. Instead the choice of architecture should be based on the requirements of the agent system to be developed. For example; consider MAS that acts as a flight traffic controller system. An architecture for agents that can learn and adapt their behaviour is unnecessary and possibly dangerous for such a system. Such control systems are designed (and are **expected**) to give the same responses to same events. What follows is a summary of available agent architectures. *Logic-based (symbolic)* architectures draw their foundation from traditional knowledge-based systems techniques in which an environment is symbolically represented and manipulated using reasoning mechanisms. The advantage of this approach is that human knowledge is symbolic so encoding is easier, and they can be constructed to be computationally complete, which makes it easier for humans to understand the logic. The disadvantages are that it is difficult to translate the real world into an accurate, adequate symbolic description, and that symbolic representation and manipulation can take considerable time to execute with results that are often available too late to be useful.[2] *Reactive* architectures implement decision-making as a direct mapping of situation to action and are based on a stimulus-response mechanism triggered by sensor data. Unlike logic-based architectures, they do not have any central symbolic model and therefore do not utilize any complex symbolic reasoning.[2] Instead they generally include predefined finite-state machines that define their behaviour. Sensory input data is run through these finite-state machines and responses are determined accordingly. These finite-state machines can also be organized into a decision-making hierarchy that creates a layered architecture where each layer can be specialized for handling a specific response types (much like the layered structure of sensory parts of the human brain) *BDI (Belief, Desire, Intention)* architectures are probably the most popular agent architectures. They have their roots in philosophy and offer a logical theory which defines the mental attitudes of belief, desire and intention using a modal logic.[2] *Layered (hybrid)* architectures allow both reactive and deliberative agent behaviour. To enable this flexibility, subsystems arranged as the layers of a hierarchy are utilized to accommodate both types of agent behaviour. There are two types of control flow in layered architectures; horizontal and vertical.[2] In horizontal layering each layer is directly connected to sensory inputs and action outputs, having each layer act as a separate agent. Although this approach is simple to design, synchronization, coordination and consistency issues can occur in such an architecture. Also as the number of layers increases the complexity of the actions that can take place increases. Vertical layering eliminates these problems to some extent by dealing with sensory input and action output with by at most one layer (thus having no consistency problems). Data and control flow in layered architectures is shown in figure 2.2.

## 2.3. FIPA

Foundation for Physical Agents is a non-profit organization founded in 1996 for creating a set of standards for developing multi-agent platforms. At the core of FIPA is the following set of principles:

1. Agent technologies provide a new paradigm for solving old and new problems;
2. Some agent technologies have reached a considerable degree of maturity;
3. To be of use some agent technologies require standardization;
4. Standardization of generic technologies has been shown to be possible and to

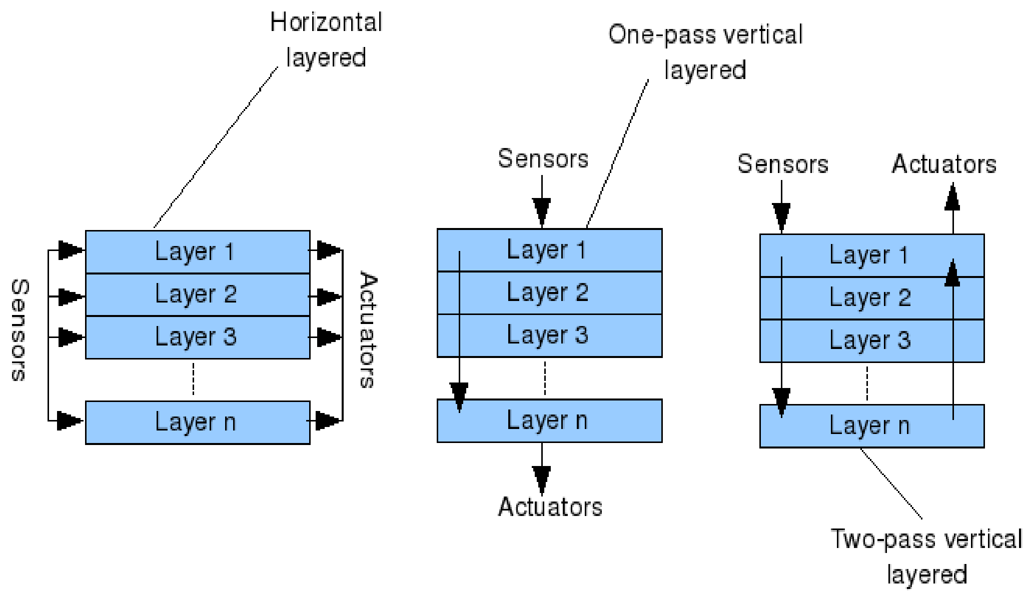

Figure 2.2. Horizontal and Vertical Layered Architectures

provide effective results by other standardization fora;

5. The standardization of the internal mechanics of agents themselves is not the primary concern, but rather the infrastructure and language required for open inter-operation. [2]

Since its inception FIPA has published three versions of open standards for multi-agent systems; FIPA '97, FIPA '98, FIPA2000. With the last standard release FIPA also published the FIPA Abstract Architecture; a document that defines all core architectural elements of a FIPA compliant Multi-Agent platform and the relationships between these elements, guidelines for multi-agent system definitions in terms of software technologies and communication protocols.

After a brief period of inactivity (between 2004 and 2005) FIPA was reformed (and renamed as FIPA-IEEE) as a standards activity under the IEEE. Since then work has been concentrated on integration with web services, human-agent communication, mobile agents and peer-to-peer nomadic agents.

## 2.4. JADE

Java Agent Development Framework is a FIPA compliant multi-agent systems development framework and run-time platform that was developed by Telecom Italia. The initial goal behind the development of JADE was to test early FIPA specifications. JADE provides standard agent technologies and offers to the developer a number of features in order to simplify the development process:

- Distributed agent platform. The agent platform can be distributed on several hosts, each one of them executes one Java Virtual Machine.
- FIPA-Compliant agent platform, which includes the Agent Management System the Directory Facilitator and the Agent Communication Channel.
- Efficient transport of ACL messages between agents. [5]

In the following sections the major components of the JADE framework are discussed.

### 2.4.1. Agent Model in JADE

Agents in JADE are developed as Java classes that extend the jade.core.Agent class. Agents in JADE have a lifecycle similar to Java Applets. They are created, have a run-time, then they are terminated. Similar to Applets JADE agents have predefined methods that correspond to these life cycle periods. Agents can override the **setup** method defined in jade.core.Agent class to have custom initialization steps. Likewise agents can override the **takeDown** method to have custom finalization code. The basic structure for the lifecycle of JADE agents is given in figure 2.3. Each agent in a JADE platform must have a name that is globally unique. In practice global uniqueness is achieved by (a process managed by the AMS) adding the platform name to the original name of the agent. The identification of each agent is stored in an instance of the jade.core.AID class.

### 2.4.2. Agent Activities

Actions an agent can do in a JADE platform are defined through **behaviors**. Behaviors an agent can have during its run-time are defined by extending one of the behavior classes defined in the jade.core.behaviours package and adding instances of these classes to the task queue of the agent. The jade.core.behaviours.Behaviour base class defines two public abstract methods that must be overridden by child classes;

1. an action method (method signature: public void action();) that defines the code that constitute the task carried out by this behaviour.
2. a done method (method signature: public boolean done() ;) that defines a control structure for checking the completion status of this behaviour. As it can be seen from figure 2.3 the done method is called after each execution of a behaviour.

In addition to the above methods **jade.core.behaviours.Behaviour** class also defines two methods that can be overridden for further behaviour customization;

1. the onStart() method which is executed once before the first call to the action method (similar to the setup() method of jade.core.Agent class)
2. the onEnd() method which is executed right after the done() method returns true (similar to the takeDown() method of jade.core.Agent class)

There are two direct extensions to the **Behaviour** class that constitute the two primary types of agent behaviours;

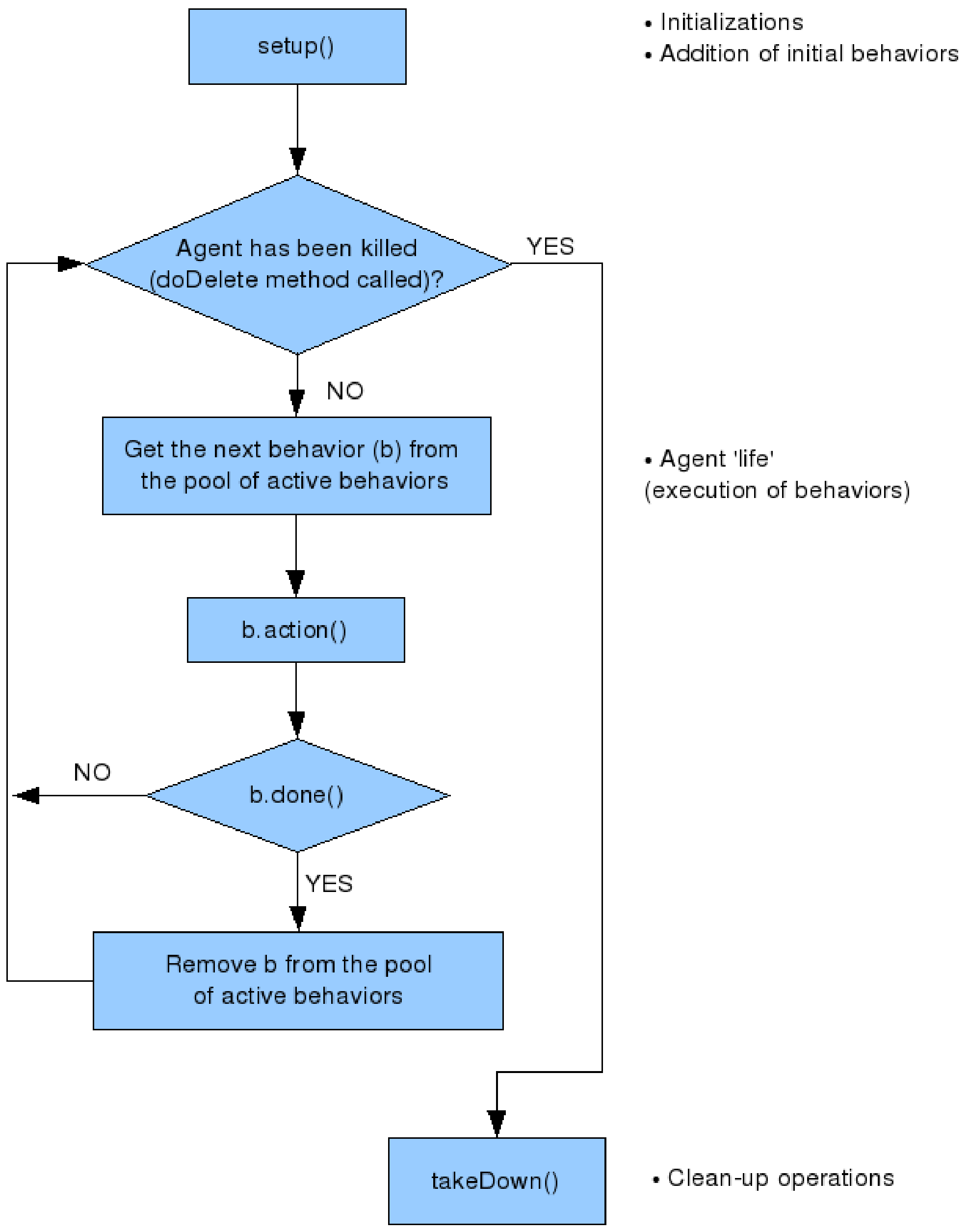
setup()
• Initializations
• Addition of initial behaviors
Agent has been killed (doDelete method called)?
YES
NO
Get the next behavior (b) from the pool of active behaviors
• Agent 'life' (execution of behaviors)
b.action()
b.done()
NO
YES
Remove b from the pool of active behaviors
takeDown()
• Clean-up operations

Figure 2.3. JADE - Agent thread path of execution[2]

- **SimpleBehaviour**; (and extending classes) represent atomic simple tasks that can be executed a number of times specified by the developer. [5]
- **CompositeBehaviour**; (and extending classes) support the handling of multiple behaviours according to a policy. [5]

There are two primary types of simple behaviours in JADE; [2]

1. One-shot behaviours are designed to complete in one execution phase; their action() method is thus executed only once.
2. Cyclic behaviours are designed to never complete; their action() method executes the same operations each time it is called.

There are three implementations of **CompositeBehaviour** provided with the JADE distribution;

- **SequentialBehaviour**; a composite behaviour that schedules its children according to a very simple sequential policy [2]
- **FSMBehaviour**; which executes its children behaviours according to a Finite State Machine (FSM) of behaviours [5]
- **ParallelBehaviour**; a composite behaviour that schedules its children in parallel [2]

The Agent-core of each agent (provided by the superclass **jade.core.Agent**) keeps a tasklist for active behaviours.

### 2.4.3. Communication Among Agents

Agents in JADE talk to each other by creating messages that adhere to the Agent Communication Language (ACL) standard. Communication between agents in JADE is realized by the Message Transport Service (as defined by the FIPA specifications). The MTS is a service provided by an AP to transport FIPA-ACL messages between agents on any given AP and between agents on different APs. Messages are providing a transport envelope that comprises the set of parameters detailing, for example; to whom the message is to be sent. The general structure of a FIPA compliant message is depicted in Figure 2.4.

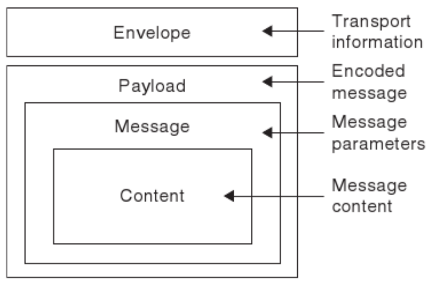

Figure 2.4. FIPA message structure

In more detail a FIPA ACL message is composed of the following fields;

- Type of Communicative Act
  - `performative`; is a required parameter that defines the communicative act (or purpose) of the message. Some performative types are; `INFORM, PROPOSE, CFP` (Call for Proposals).
- Participants in Communication
  - `sender`; gives the identity of the sender of the message (for JADE the AID) and can be omitted if the sender desires to remain anonymous.
  - `receiver`; gives the identities of the recipients of the message
  - `reply-to`; indicates that subsequent messages in this conversation thread are to be directed to the agent identified by the reply-to parameter.
- Content of Message
  - `content`; the content of the message encoded in a way (defined by the de-

| **Parameter** | **Category of Parameters** |
|---|---|
| `performative` | Type of communicative acts |
| `sender` | Participant in communication |
| `receiver` | Participant in communication |
| `reply-to` | Participant in communication |
| `content` | Content of message |
| `language` | Description of content |
| `encoding` | Description of content |
| `ontology` | Description of content |
| `protocol` | Control of conversation |
| `conversation-id` | Control of conversation |
| `reply-with` | Control of conversation |
| `in-reply-to` | Control of conversation |
| `reply-by` | Control of conversation |

Table 2.1. FIPA ACL MessageParameters [6]

scription of content parameters) that the receiver can understand.

- Description of Content
  - `language`; the language in which the content parameter is expressed
  - `encoding`; the specific encoding of the language expression
  - `ontology`; the vocabulary(ies) used in the content expression
- Control of Conversation
  - `protocol`; the interaction protocol used for the message; FIPA-request, FIPA-query, etc.
  - `conversation-id`; an expression used for identifying ongoing sequence of communication acts (a conversation) between agents
  - `reply-with`; used to define a key for identifying this message
  - `in-reply-to`; used for linking a reply message to a previous message with reply-with parameter
  - `reply-by`; specifies the latest time a reply is expected for this message, used for timeout definitions.

The content slot of an ACL message can contain either a string value or a sequence of bytes. In its most basic use an ACL compliant message, the content is a simple string that includes name-value pairs separated by semicolons. More advanced content representations are available through content languages and ontologies. Each agent is equipped with an incoming message box and message polling can be blocking or non-blocking with an optional timeout. Moreover, JADE provides methods for filtering. The developer can apply advanced filters on the various fields of the incoming message such as sender, performative or ontology. [5]

A typical agent-to-agent communication occurs as follows; An agent (initiator) starts communication by sending a message to another agent(s) (responder). If the message is not one that the responder can understand or is expecting the responder can reply with not-understood or refuse message indicating its inability to join the communication. Otherwise the responder replies with an agree message. Upon completion of the action requested with the message the responder must send an inform message to the initiator about the result of the action.

### 2.4.4. Content Languages and Ontologies

In realistic scenarios agents in a MAS often need to communicate complex information (such as database records, actual java objects, etc.) Content languages define syntaxes for representing complex data in ACL message slots. An example content language is the FIPA defined SL language which represents data as a string sequence. The actual conversion from complex data representation to a content language syntax is done via special codecs that handle the encoding and decoding processes.

An ontology defines mappings (name, type and validation) between named items in a ACL message content and actual fields and methods inside a Java class. Thus using an ontology processing decoded ACL message content into a usable Java class is automatized. When used together, content languages and ontologies provide a simple, effective and truly platform independent framework for serializing and deserializing agent communication content.

### 2.4.5. Agent Platforms and Containers

In a JADE environment the top level agent run-time environment is the Agent Platform. Physically an agent platform may span multiple computing environments (servers, workstations, mobile devices). Different agent platforms can be interconnected to form a platform forest for massively complex multi-agent systems. Inside agent platforms, agents exists inside special run-time elements called agent containers. When started (bootstrapping in JADE terms) each agent platform creates an initial (and singular) agent container called the main container (additional containers may also be created via bootstrap configurations). This container is responsible for keeping track of all the agents that exist in that specific platform and for providing synchronization services of this agent registry when there are more than one containers active in that specific platform. This responsibility (or role) can be transferred to another container in case the initial main container has to terminate for some reason. The main container is also home to two special agents (created during the bootstrap process); the Directory Facilitator (DF) agent and the Agent Management System (AMS). The DF is an agent that provides yellow pages services for the agent platform. Agents that want to announce their services can register themselves with the DF and requesting agents can run queries against the DF. The AMS provides (as its name implies) agent management services (agent creation/termination, etc.) and white pages services for the agent platform. [2] Figures 2.5 and 2.6 show main JADE architecture elements and their relations.

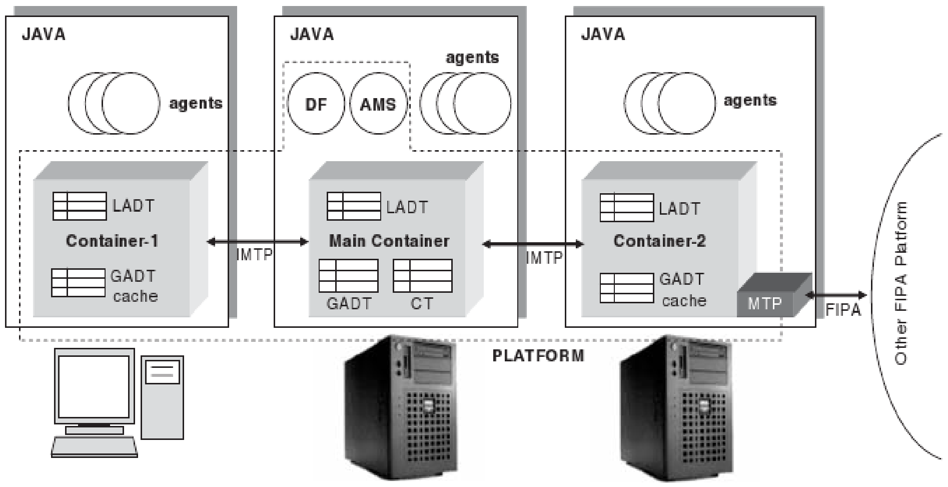

Figure 2.5. JADE Agent Platform Architecture[2]

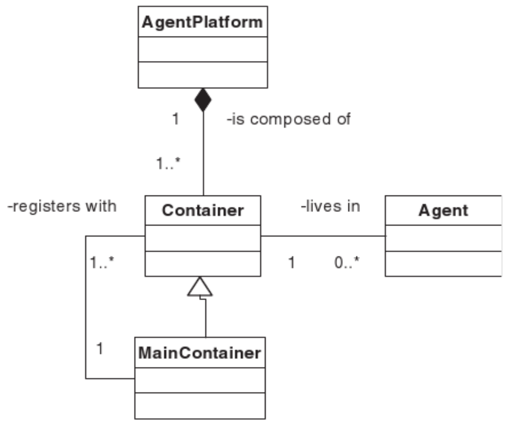

Figure 2.6. JADE Agent Platform Architecture Elements[2]

## 2.5. Gaia Methodology for Developing Multi-Agent Systems

The Gaia methodology is an attempt to define a complete and general methodology that is specifically tailored to the analysis and design of MASs. Gaia is a general methodology that supports both levels of the individual agent structure and the agent society in the MAS development process. [5] A typical Gaia effort is divided into three phases; an initial phase where requirements for the MAS is gathered from interested parties, an analysis phase where the initial requirements are used for creating the roles model and the interaction model and finally a design phase where the models from the analysis phase are used for creating the agent model, the services model and the acquaintance model.

### 2.5.1. Analysis Phase

The objective of the Gaia analysis process is the identification of the roles and the modelling of interactions between the roles found.[5] The outputs of the analysis phase provide an abstract model representation of the MAS. Roles correspond to the sources of activity that exist in the MAS. For each role four types of attributes are used when building a definition; responsibilities, permissions, activities and protocols.

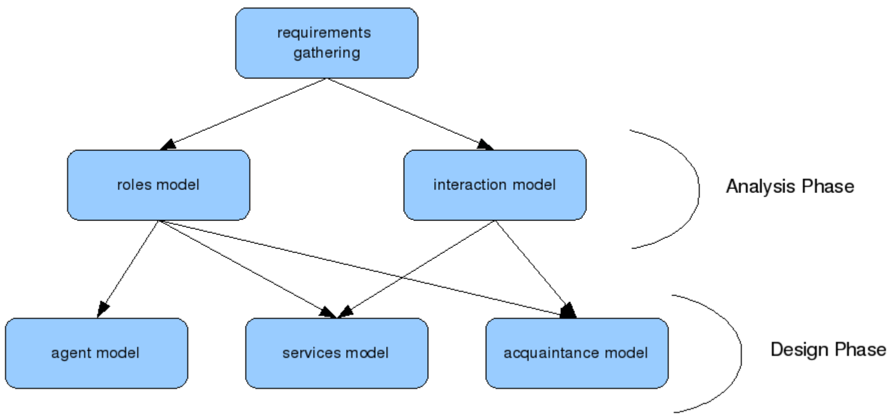

Figure 2.7. Gaia Framework Structure

Responsibilities determine the functionality of the role by defining the functional expectations the role must fulfil or serve. There are in turn two different types of responsibility properties; liveness and safety. Liveness properties determine the behaviour patterns of the role (which result in positive contributions to the MAS), they are defined as the tasks that a role must complete under certain environmental conditions. For example; in a course registration MAS a role representing a student will have the liveness property that tries to finalize its registration once the student's course program is completed. Safety properties determine constraints on the actions of the role and conditions the role will prevent from happening. A simple example for a liveness property definition is given below;

$$SAMPLE\_AGENT = \underline{InitializeSelf}.\underline{PickActivityPath}.DoActivity$$

Figure 2.8. Sample Gaia Liveness Property Definition

Here the "." operator is used for denoting the sequence the activities and properties are to be executed for the liveness sequence. The complete list of operators supported for liveness formulas are given in Table 2.2.

| **Operator** | **Interpretation** |
|---|---|
| $x.y$ | x is followed by y |
| $x\|y$ | x or y occurs |
| $x^*$ | x occurs 0 or more times |
| $x^+$ | x occurs 1 or more times |
| $x^\omega$ | x occurs infinitely often |
| $[x]$ | x is optional |
| $x \parallel y$ | x and y are interleaved |

Table 2.2. Gaia Operators for Liveness Formulas

Permissions define how each role can access resources available in the system. For example; the role representing a student in a course registration MAS will be able to read course data but will not be able to update them.

Activities define tasks that a role can carry out on its own without interacting with other roles in the MAS. For example; the functionality a student role uses for picking a course to register can be considered an activity (with the assumption that instances of the student role do not discuss registration options with other instances). Protocols represent patterns of interactions supported by each role.

An important aspect of MASs is that the agents existing in the system realize micro and macro level objectives by interacting with each other. Any framework related to MAS design thus must provide representations for the interactions between agents. In Gaia the first level of representing inter-agent interactions is done via the Interactions Model. Each entry in the Interactions Model corresponds to a protocol definition. In turn each protocol definition consists of the following attributes [4];

- *purpose*; brief textual description of the nature of the interaction;
- *initiator*; the role(s) responsible for starting the interaction;
- *responder*; the role(s) with which the initiator interacts;
- *inputs*; information used by the role initiator while enacting the protocol;
- *outputs*; information supplied by/to the protocol responder during the course of

the interaction;

- *processing*; brief textual description of any processing the protocol initiator performs during the course of the interaction.

The analysis process can be summarized by the following algorithm;

1. Identify roles in the system. For each role provide an informal, simple description.
2. For each role identify and describe the associated protocols.
3. Based on the protocol model in step 2 detail the roles model by adding permissions, responsibilities, protocols and activities for each role identified.
4. Repeat previous step until all possibilities in the requirements are exhausted.

Figure 2.9. Gaia Analysis Process

### 2.5.2. Design Phase

During the design phase the abstract models conceived in the analysis phase are converted into models that are at a sufficiently low level of abstraction [4] that they can be easily used as agent-design references for traditional software design methodologies. As a result three models are created; Agent Model, Services Model and Acquaintance Model.

**Agent Model**; is a model that identifies the agent types to be used in the system as well as agent instance information (how many instances will be instantiated). The agent model also provides the mapping between Roles and Agents. Agents in the Agent Model are defined as sets of roles defined in the analysis phase. The correspondence between roles and agents can be one-to-one or many-to-one (when related roles are packaged in to a single role). For each agent defined in the agent model the model also includes instance qualifiers which denote the number of instances expected to be active in the MAS. Table 2.3 gives a list of possible instance qualifiers along with descriptions.

| **Qualifier** | **Meaning** |
|---|---|
| $n$ | exactly $n$ instances will be instantiated |
| $m..n$ | between $m$ and $n$ instances will be instantiated |
| $*$ | 0 or more instances will be instantiated |
| $+$ | 1 or more instances will be instantiated |

Table 2.3. Gaia Agent Instance Qualifiers

**Services Model** identifies the main services that are required to realise the agent's role.[4] For each service available a service description is written with the properties; *inputs, outputs, pre-conditions and post-conditions.* Each one of the services is based on one or more of the protocols identified in the analysis phase thus inputs and outputs of services are derived from inputs and outputs of protocols. Pre- and post-constraints represent constraints on services. These are derived from the safety properties of a role.[4]

**Acquaintance Model** identifies paths of communications between agents based on the agent model and the interactions model. The model can be visualized as a directed graph between agents identified in the Agent Model; an arrow from agent $a$ to agent $b$ means that agent $a$ can send messages to agent $b$ (in effect call services defined in agent $b$).

The design phase can then be summarized as;

1. Create an agent model by mapping roles to agents and defining instance annotations.
2. Develop a services model, by examining activities, protocols and safety and liveness properties of roles.[4]
3. Develop an acquaintance model from the interaction model and agent model.[4]

Figure 2.10. Gaia Design Process

# 3. Modelling a University as a Multi-Agent System

## 3.1. Model Definition

This work proposes a multi-agent model of a university that can be implemented into a fully functional multi-agent system. The full definition of this model as well as implementation details are provided in the sections that follow.

### 3.1.1. Agent Platform and Container Definition

In this model the whole university is defined inside a single agent platform. For each faculty and administrative unit a separate agent container is defined. The idea here is to have localized data and processing services for each academic and administrative unit. During the implementation phase each agent container will run its own server system, thus achieving a level of load balancing in the system.

### 3.1.2. Agent Definitions

Agent definitions for this model are discussed in two groups; model-phase agents and implementation-phase agents. Model-phase agents represent actors that are defined in the conceptual model of the system and are also present at the implementation phase. Implementation-phase agents represent agents that are specific to the implementation and are either required for serving core functionalities or to meet programming needs (as in base Java classes).

### 3.1.3. Model-Phase Agents

3.1.3.1. Student-Agent. In its basic form software agents representing students will contain identification data, data representing the students view of his/her academic record (which in a optimum system will be complete, upto date and accurate) and students understanding of rules and regulations (which again in an optimum system is complete, accurate and upto date). Furthermore the agent will include certain actions (defined as JADE behaviours) for interacting with other agents. These include actions for course registration and information query.

Software agents representing students can be further be enhanced with behavioural rules representing the interests and demands of actual students during course registration periods. These behavioural rules or beliefs can then be attached to actual agent behaviours for modeling the activities of students during course registration periods. The resulting agent model for students thus contains;

1. beliefs regarding the students ability in specific courses and general subject matters
2. beliefs regarding the instructors (the students relations with the instructor, instructors behaviour patterns)
3. additional beliefs modeling more generic student behaviour
4. behaviours for course registration activities (searching for courses, course data queries, add/drop courses, etc.)
5. behaviours for interaction with instructors and administrative bodies (approval requests, information queries, etc.)

6. behaviours for interaction with fellow students (querying registered courses, course swapping, belief exchanges, etc.)

3.1.3.2. Faculty-Agent. Similarly agents representing instructors will have identification data as well as a representation of the instructors academic background (in terms of subject areas). Instructor agents will also have a representation of their understanding of rules and regulations (which again in an optimum system is complete, accurate and upto date). Instructor agents will also have a representation of their own beliefs and a set of behaviours that define the actions they can take.

1. beliefs regarding the students the instructor knows
2. beliefs regarding the instructors expectations from students registering to courses
3. beliefs regarding the agents understanding and execution of rules and regulations (important for instructors that also have the role of advisor)
4. additional beliefs modeling more generic instructor behaviour
5. behaviours for handling advisor requests (when applicable)
6. behaviours for handling course queries
7. behaviours for generic interactions with students
8. behaviours for interaction with instructors and administrative bodies

3.1.3.3. Administrative-Agent. The third and last kind of agents represent administrative units. These units include registrars office (can be decomposed to several delegate agents each handling a specific functionality), quota managers, delegate agents representing councils, secretaries and other clerks. These agents can either represent a real person (as in the case of secretaries) or can be delegate agents that either represent a single anonymous entity (as in quota managers) or represent a group of agents (as in councils and boards).

3.1.3.4. Aggregate-Agent. Aggregate-Agents represent structured collaborations (mostly for decision making purposes) between agents. They have their internal lists of collaborating agents, a message service for passing messages between participating agents and concensus control/decision making mechanisms. Aggregate-Agents can be used for representing continuous groups (like a faculty board) or can be used for one-shot decision making efforts.

3.1.3.5. Pseudo-Agent. Pseudo-Agents are used to represent meta-actors that are generally related to a certain task/responsibility ( or a combination of these ) and that can be occupied by different agents at different points of time. An implementation of a pseudo-agent will contain a field for linking it to its acting agent, a list representation of tasks and privileges related to the position represented by the Pseudo-Agent.

### 3.1.4. Implementation-Phase Agents

3.1.4.1. MausAgent. MausAgent is the super class of all core agents implemented in the Maus system. It provides basic agent functionality and fields. The most important parts of the MausAgent implementation are the fields; *mausUniqID*, *mausType* and *actionTable*. *mausUniqID* and *mausType* are used for identifying and categorizing agents with respect to the Maus platform specification. *actionTable* is a representation of the actions known to an agent.

3.1.4.2. DataProviderAgent. DataProviderAgent is an agent implementation that provides services for accessing datastorages active in a Maus deployment. It can also be used as a base class for implementing specialized data access providers.

3.1.4.3. CoreServicesAgent. CoreServicesAgent is an agent aggregating basic services (format converters, utility access, etc.) for the Maus system.

3.1.4.4. JADECreatorAgent. JADECreatorAgent is a singleton (meaning only one instance is allowed per agent container) agent that is responsible for the initialization of a JADE container.

## 3.2. Implementation

Implementation details are discussed in two parts; programming implementation and deployment implementation.

### 3.2.1. Programming Implementation

The multi-agent university model is implemented as a Java agent application that runs on a JADE agent platform. For portability purposes (with respect to agent platform used) the whole model is developed using a dual class hierarchy where applicable. This dual approach requires that core agent object and related functionality is developed without any reference to any agent deployment platform. For each agent implementation in this first hierarchy a complementing stub implementation is developed for connecting to the actual agent run-time. What follows is a description of the package structure, core agent implementations and JADE stub implementations.

### 3.2.2. Package Structure

The base package for the multi-agent university system (MAUS) is **edu.zion.mas.maus**. Class implementations are distributed between sub-packages based on their types and/or functionalities.

3.2.2.1. edu.zion.mas.maus.core. This package contains core objects(classes and interfaces, not to be mixed with run-time objects) for the MAUS system. Core objects include top level base classes for the inheritance hierarchy, implementation specific datatypes and public access interfaces for providing common constants and functionalities. The package contents are;

- **MausAgentIdentifier**; represents a class implementation of the base agent identifier for agents in the MAUS system. At this time a single string is used as an identifier. For more complex implementations a composite identifier may be more appropriate.
- **MausCommonConstants**; is a public interface that holds common constant values (e.g; enumeration for Agent types) for the MAUS system.
- **MausCommonStrings**; is a public interface that holds common strings (such as system messages) for the MAUS system.
- **MausCoreObject**; is an abstract class that is the base class for all classes implemented for the MAUS system. At this time the only content of this base class is an abstract method *clone()* that is used for creating exact copies of MAUS objects. Every non-abstract class extending (directly or indirectly) **MausCoreObject** is expected to provide an implementation for the *clone()* method.

3.2.2.2. edu.zion.mas.maus.core.data.providers. This package contains class implementation for datasource providers. Datasource providers provide functionality for access control and data-access (queries,inserts,updates) operations. The package contents are;

- **MausCourseDataProvider**; a data provider that is specialized for course data related queries.
- **MausInternalCourseDataProvider**; an internal storage engine implementation of the **MausCourseDataProvider**
- **MausInternalGenericDataProvider**; an internal storage engine implementation of a generic data provider
- **MausInternalRegistrationDataProvider**; an internal storage engine implementation of the **MausRegistrationDataProvider**

- **MausInternalStudentDataProvider**; an internal storage engine implementation of the **MausStudentDataProvider**
- **MausRegistrationDataProvider**; a data provider that is specialized for registration data related queries.
- **MausStudentDataProvider**; a data provider that is specialized for student data related queries.

3.2.2.3. edu.zion.mas.maus.core.data.types. This package contains class implementations that are used for data storage and exchange purposes.

- **Course**; a data representation class for courses available in the Maus system.
- **CourseInstance**; a data representation of a specific course instance (a section opened in a semester) in the Maus system.
- **CoursesEnrolled**; a data representation of the list of courses enrolled by a student in a given registration period.
- **MausDataType**; a base class for all Maus specific data type classes.
- **Student**; a data representation class for a students record in the Maus system.

### 3.2.3. edu.zion.mas.maus.core.data.onto

This package contains ontology implementations for datatypes that can be used in message exchanges.

### 3.2.4. edu.zion.mas.maus.agents

This package contains Java classes for core agents and complementing JADE stub agents. This package contains the implementations for the following classes;

- **MausAgent**; This class forms the base class for all core agents in the MAUS implementation. The MausAgent defines three common fields for all MAUS agents; the actionTable (a hashtable holding the list of actions an agent can take), the agents type (with respect to MAUS) and the agent's unique identifier.

- **MausAgentJADEStub**; This class is the JADE stub for the top-level MausAgent and provides a default implementation of the *setup()* method for JADE agents.
- **AdministrativeAgent**; This is the core agent implementation for the administrative agents described earlier.
- **AdministrativeAgentJADEStub**; This is the JADE stub for the administrative agent implementation.
- **AggregateAgent**; This is the core agent implementation for the aggregate agents described earlier.
- **AggreagateAgentJADEStub**; This is the JADE stub for the aggregate agent implementation.
- **CoreServicesAgent**; This is the core agent implementation for the core-services agent described earlier.
- **CoreServicesAgentJADEStub**; This is the JADE stub for the core-services agent described earlier.
- **DataProviderAgent**; This is an abstract class that acts as a base class for all core agent that will be used for data serving purposes.
- **FacultyAgent**; This is the core agent implementation for the faculty agents described earlier.
- **FacultyAgentJADEStub**; This is the JADE stub for the faculty agent implementation.
- **JADECreatorAgent**; This is the implementation for the JADE agent responsible for initializing the MAUS run-time environment.
- **PseudoAgent**; This is the core agent implementation for the pseudo agents described earlier.
- **PseudoAgentJADEStub**; This is the JADE stub for the pseudo agent implementation.

- **StudentAgent**; This is the core agent implementation for the student agents described earlier.
- **StudentJADEStub**; This is the JADE stub for the student agent implementation.

<u>3.2.4.1. edu.zion.mas.maus.agents.actions.</u> This package contains Java implementations for core agent actions. Core agent actions define functionalities representing actions core agents can use to interact with their environments and with other agents. Each action implementation (at the very least) includes;

- a field that gives a list of agent classes this action can be attached to
- a method for attaching the action to a JADEStub agent
- a method for transforming the agent action to a valid JADE agent behaviour

This package contains the following class implementations (at this time);

- **MausAgentAction**; an abstract class that acts as the base class for agent action implementations. MausAgentAction defines the common fields and base functionality for all agent action implementations.
- **EnrollToNewCourse**; a sample agent action implementation that is applicable only to student agents. This action allows a student agent to create a JADE behaviour for enrolling to a course (the request and negotiation phases).
- **GetPossibleCourses**; a sample agent action implementation that is used for querying the course database for courses fitting a given search criteria.

<u>3.2.4.2. edu.zion.mas.maus.behaviour.</u> This package contains actual JADE behaviour implementations that can be created by MAUS agent actions.

- **EnrollToNewCourseBehaviour**

- **QueryCourseListBehaviour**

3.2.4.3. edu.zion.mas.maus.kb.believes. The first of the two knowledge-base packages contains class implementations representing believes of agents in the MAUS system and interface implementations used for categorizing believes. This package includes the following sample class implementations;

- **MausCoreBelief**; This abstract class acts as a base class for all belief objects in the MAUS system
- **AvoidClassMate**; This belief implementation is used for defining which agents a certain student agent will try to avoid as class mates when registering for courses.
- **AvoidInstructor**; This belief implementation is used for defining which instructor agents a certain student will try to avoid when registering for courses.
- **PreferredClassMate**; This belief implementation is used for defining which agents a certain student will try to have as class mates when registering for courses.
- **PreferredInstructor**; This belief implementation is used for defining which instructor agents a certain student agent will prefer when registering for courses.
- **StrongCourse**; This belief implementation is used for defining a course in which a certain student agent believes is strong at.
- **StrongSubject**; This belief implementation is used for defining a course subject area in which a certain student agent believes is strong at.
- **WeakCourse**; This belief implementation is used for defining a course in which a certain student agent believes is weak at.
- **WeakSubject**; This belief implementation is used for defining a course subject area in which a certain student agent believes is weak at.

This package also include the following interface implementations;

- **AbilityBelief**; This interface is used for identifying believes regarding an agents ability in subjects and actions.
- **PreferenceBelief**; This interface is used for identifying believes regarding an agents preferences against other objects in the MAUS system.

<u>3.2.4.4. edu.zion.mas.maus.kb.rules.</u> The rules package of the knowledge-base packages contains class implementations representing rules that are used for checking validity of agent actions and for decision making purposes. This package includes the following sample class implementations;

- **MausCoreRule**; This abstract class forms a base for other rule implementations to build on. The following abstract methods are declared for extending classes to override;
  - ***boolean*** *initialize(**MausCoreObject** subject, **Hashtable<String, Object>** params)*; initializes the rule with the given set of parameters (as a Hashtable of Java objects) and attaches itself to the given subject (a MausCoreObject instance)
  - ***boolean*** *isInitializable()*; a method for querying if the initialize method can be called for the rule class in question. Some rule implementations may not require initialization procedures other than their own constructor methods.
  - ***boolean*** *check(**MausCoreObject** object, **Hashtable<String, Object>** params)*; a method that tests its rule against the object and parameters provided to the method. Not all rules will have check procedures.
  - ***boolean*** *isCheckable()*; a method for querying if the check method can be called for the rule class in question.
- **GenericRule**; This class provides an empty rule that can be extended for implementing custom rules during run-time.
- **AvoidInstructorRule**; This class provides a complete implementation of a rule that defines negative preference relations between a student and an instructor

agent.

- **CreditOverloadRule**; This class provides a complete implementation of a rule that defines the maximum credits a student can enroll to in addition to the normal credit load allowed.
- **ScheduleConflictRule**; This class provides a complete implementation of a rule that defines the maximum number of schedule conflicts a student can have.

3.2.4.5. edu.zion.mas.maus.util. This package contains general utility classes and applications for the MAUS system.

- **ScheduleConstants**

3.2.4.6. edu.zion.mas.maus.util.management. This package contains management utilities for the MAUS system.

- **DBManagerGenericInternal**

3.2.4.7. edu.zion.mas.maus.util.ui. This package contains UI centric utility classes for the MAUS system.

- **DBViewGenericInternal**

3.2.4.8. edu.zion.mas.maus.util.web. This package contains web utility classes and other JADE-WSIG related classes.

## 3.3. Data Storage

The MAUS system can use a combination of data storage options from Java-based internal storage to standalone SQL-compliant database servers. In this section possibilities for all considered options are discussed.

The sample implementation uses data providers that use JavaDB database engine for storage and data retrieval.

## 3.4. Deployment

In a complete deployment environment the MAUS system will be deployed on a set of computer where each academic unit is represented by its own agent container and in turn each agent container is deployed on its own server machines.

In the example deployment scenario the whole sample system will be deployed on a single agent container running on a single server. For data storage purposes JavaDB based data providers are used.

The aforementioned agent platform will be deployed on a set of computers where each agent container executes on its own server machine. Agents will be built against the 3.5 version of the JADE libraries. For purposes of data storage Java-Derby embedded database engine will be used. For general simulation purposes the database engine used is not important and any jdbc-connectable database engine will suffice. Using Derby at this step further simplifies data handling procedures. In any case using jdbc to connect to the databases allows for a database engine independent coding (unless some specialized/implementation specific functionality (like transactions) is used that makes the system database engine dependent). For purposes of web-based front-ends JADE-WSIG will be used and the functionality of the system will be exposed as WSDL compliant web services. The actual web-based front ends will be implemented as Java servlets and will be deployed on a J2EE application server. For agent communication purposes the SL-codec will be used with a custom ontology defined specifically for this MAS.

# 4. Applications of this Model

The model presented can be used both as a simulation and as an actual system for automating various activities that can be found in a university environment. In the following subsections sample applications for this model are discussed.

## 4.1. Quota Forecasting

In this application the agents in the MAS are initialized with student academic records, behavioral data (possibly collected through behavioural profiling questionnaires), rules-regulations. Into this MAS we can introduce the list of available courses (along with schedule information, eligibility criteria and quota limitations). In this state the agents can be allowed to run fully autonomously to interact with each other as real students would do during a course registration period.

Data gathered through this simulation can then be used for forecasting the demand on courses and provide a better judgement on course and quota planning. This simulation will also allow for trials on course parameters like schedule data and instructor relationships and effects of changes on these parameters can safely be observed.

## 4.2. Online Course Registration

The MAS used in the "Quota Forecasting" application can be enhanced to provide interactivity to the extend that agents can be controlled by their respective owners to alter their behaviour based feedback from the university environment. Such a system can be used for forming a semi-autonomous online course registration system. In this scenario student agents will have a base understanding of their owners agendas regarding course registration (based on student's academic profile and further calibration data collected through student surveys or other means). Accordingly faculty and administrative agents will have a total understanding of the regulations that must be adhered during the course registration. Further individual agents will be initialized with owner specific profile data.

Once deployed each agent will try to satisfy its expectations autonomously to the extend allowed by the regulations and their owners. For example; an agent representing a repeat status student will try and take all repeated courses without the need of approval from its owner. In case there are multiple possibilities the agent can either try to make a decision based on its beliefs (leave one day of the week free, register to the same course section with a fellow student, avoid a certain instructor, etc.). Likewise an agent representing an advisor will try to handle the requests coming from students based on its profile definition.

## 4.3. Regulation Simulations

As mentioned earlier regulations concerning the university can be implemented as rules in the **edu.zion.mas.maus.kb.rules** package extending the **MausCoreRule** or some other more specialized rule implementation class. Using these rule implementations simulation runs can be made on a Maus system initialized with training data and with fully autonomous agents and observations can be made on the behaviour of the agents and activity patterns in the MAS. Such a simulation application can then be used for testing the effects of regulation changes. The effects can be observed in a multitude of observation points; registration patterns of students, bottlenecks in inter-agent activities, patterns in requests arriving at AggregateAgents representing governing bodies, etc.

## 4.4. Online Student-Faculty Collaboration System

A final application proposal is that of a Student-Faculty Collaboration System. In this scenario the MAS acts as a storage for collaboration data (schedules, announcements, public/private messages, journals, etc.) and a platform for social networking (agents acquaintance graphs representing real life interpersonal relations and affiliations). Each agent active in the system will provide the following services for their respective owners;

- harvest data from the system according to interests and affiliations of their owners
- provide gateway services for data manipulations and custom queries of the central storage
- provide communication services for one-to-one and one-to-many conversations.
- provide social networking services such as group formation, friend lists, organization affiliations, etc.

# 5. Sample Application

The model discussed so far is a complex system that requires the development of many software components. To show how a multi-agent approach can be applied to a realworld application regarding a university a simplified sample application for course registrations is provided in this section. The application is discussed in four sections; in the *description* section a description of the application along with application requirements is given, in the *analysis* section results of the Gaia analysis are given with descriptions of the Gaia Roles Model (based on Gaia roles identified) and Gaia Interactions Model, in the *design* section UML design of the Gaia Agent Model is given, and finally in the *implementation* section details on the JADE implementation of the Gaia Agent Model are discussed.

## 5.1. Description

This application provides a simple multi-agent system that handles course registration activities. On a preliminary analysis the following requirements have been identified;

- A student has a list of course slots that he/she must fill in order to complete his/her registration.

- A student can request a list of courses eligible for a course slot.
- A student can request to register a course to a course slot in his/her registration program.
- A registration service provider (registrar) handles and responds to course registration requests. On a successfull request the registrar will inform the requesting student and update data records accordingly.
- A data service provider handles and responds to course list queries made by students.
- A data service provider also maintains a view of the system defined as;
  - The registration programs of every student
  - The complete list of open courses
  - The list of registrants for each open course

## 5.2. Analysis

### 5.2.1. Roles Model

The analysis of the above requirements leads to the following Gaia Roles Model;

- **StudentAssistant (SA)**

  *Description:* Represents the student in the course registration MAS. SA can request courselists, pick courses for registration and tries to complete the students registration.

  *Protocols & Activities:*

  | | |
  |---|---|
  | `RequestCourseList` | `GetStudentProfile` |
  | <u>`DetermineTargetSlot`</u> | `GetRegistrationCard` |
  | <u>`DetermineCourseToRegister`</u> | `FinalizeRegistration` |
  | `RequestCourseRegistration` | |

  *Permissions:*

| | |
|---|---|
| `read` | `supplied` *courseSlot* |
| | *courseList* |
| `read` | `supplied` *courseID* |
| | *courseInfo* |
| `read` | *registrationCard* |
| `change` | *personalSchedule* |

*Responsibilities:*

*Liveness:*

$$\begin{aligned}
\text{STUDENTASSISTANT} &= \text{InitStudentProfile.}\\
&\quad (\text{RegistrationCycle})^{+}.\\
&\quad \text{FinalizeRegistration}\\
\text{INITSTUDENTPROFILE} &= \text{GetStudentProfile.}\\
&\quad \text{GetRegistrationCard}\\
\text{REGISTRATIONCYCLE} &= \text{DetermineTargetSlot.}\\
&\quad \text{RequestCourseList.}\\
&\quad \text{DetermineCourseToRegister.}\\
&\quad \text{RequestCourseRegistration}
\end{aligned}$$

*Safety:* `true`

- **RegistrationHandler (RH)**

  *Description:* Represents registrar's office in the course registration MAS. RegistrationHandler receives, verifies and processes course registration requests issued by StudentAssistants.

  *Protocols & Activities:*

  | | |
  |---|---|
  | `RegisterDF` | `DropStudentFromCourse` |
  | `QueryStudentRecord` | `InformStudent` |
  | `VerifyRegistrationEligibility` | `ReceiveRegistrationRequest` |
  | `CheckDuplicateRegistration` | `QueryCourseRecord` |
  | `FinalizeRegistration` | `AddStudentToCourse` |

  *Permissions:*

| | |
|---|---|
| `read` | `supplied` *courseID* |
| | *courseRecord* |
| `read` | `supplied` *courseID* |
| | *courseRegistrantList* |
| `change` | `supplied` *studentID* |
| | *studentRecord* |
| `change` | `supplied` *studentID* |
| | *registrationCard* |

*Responsibilities:*

*Liveness:*

$$
\begin{aligned}
\text{REGISTRATIONHANDLER} &= \underline{\text{RegisterDF}}.\\
&\quad (\text{RequestHandleCycle})^{\omega}\\
\text{REQUESTHANDLECYCLE} &= \underline{\text{ReceiveRegistrationRequest}}.\\
&\quad \text{QueryRequestParams}.\\
&\quad (\text{VerifyRegistrationRequest} \mid\\
&\quad \text{VerifyRegistrationRequest}.\\
&\quad \text{ProcessRequest}).\\
&\quad \text{InformStudent}\\
\text{QUERYREQUESTPARAMETERS} &= \text{QueryStudentRecord} \parallel\\
&\quad \text{QueryCourseRecord}\\
\text{VERIFYREGISTRATIONREQUEST} &= \underline{\text{VerifyRegistrationEligibility}}.\\
&\quad \text{CheckDuplicateRegistration}\\
\text{PROCESSREQUEST} &= \text{AddStudentToCourse} \mid\\
&\quad \text{DropStudentFromCourse} \mid\\
&\quad \text{FinalizeRegistration}
\end{aligned}
$$

*Safety:* `IncompleteRegistrations > 0`

- **DataQueryHandler (DQH)**

  *Description:* Provides read-only access to system datasources.

  *Protocols & Activities:*

| | |
|---|---|
| `RegisterDF` | `AuthenticateRequest` |
| `ProcessStudentQuery` | `ReturnQueryResult` |
| `ProcessCourseQuery` | `ProcessRegistrationQuery` |
| `ProcessCourseListQuery` | `ReceiveQueryRequest` |
| `RejectQueryRequest` | |

*Permissions:*

| | |
|---|---|
| `read` | `supplied` *courseSlot* |
| | *courseList* |
| `read` | `supplied` *courseID* |
| | *courseRecord* |
| `read` | `supplied` *studentID* |
| | *studentRecord* |
| `read` | `supplied` *studentID* |
| | *registrationCard* |
| `read` | `supplied` *courseID* |
| | *courseRegistrantList* |

*Responsibilities:*

*Liveness:*

| | | |
|---|---|---|
| DATAQUERYHANDLER | = | RegisterDF. |
| | | (QueryHandleCycle)$^{\omega}$ |
| QUERYHANDLECYCLE | = | AuthenticateQueryRequest. |
| | | ((ReceiveQueryRequest. |
| | | ProcessQuery. |
| | | ReturnQueryResult) \| |
| | | RejectQueryRequest) |
| PROCESSQUERY | = | ProcessCourseQuery \| |
| | | ProcessCourseListQuery \| |
| | | ProcessStudentQuery \| |
| | | ProcessRegistrationQuery |

*Safety:* `true`

- **DataUpdateHandler (DUH)**

  *Description:* Provides insert and update access to system datasources.

  *Protocols & Activities:*

  | | |
  |---|---|
  | `RegisterDF` | `AuthenticateRequest` |
  | `AddStudentToCourse` | `DropStudentFromCourse` |
  | `InformRequestOwner` | `RejectRequest` |
  | `ReceiveRequest` | `FinalizeRegistration` |

  *Permissions:*

  | | |
  |---|---|
  | `change` | `supplied` *courseID* *courseRecord* |
  | `change` | `supplied` *courseID* *courseRegistrantList* |
  | `change` | `supplied` *studentID* *studentRecord* |
  | `change` | `supplied` *studentID* *registrationCard* |

  *Responsibilities:*

  *Liveness:*

  $$\begin{aligned}
  \text{DATAUPTADEHANDLER} &= \underline{\text{RegisterDF}}.(\text{RequestHandleCycle})^{\omega}\\
  \text{REQUESTHANDLECYCLE} &= \underline{\text{AuthenticateRequest}}.((\underline{\text{ReceiveQueryRequest}}.\text{ProcessRequest}.\text{InformRequestOwner}) \mid \text{RejectRequest})\\
  \text{PROCESSREQUEST} &= \underline{\text{AddStudentToCourse}} \mid \underline{\text{DropStudentFromCourse}} \mid \underline{\text{FinalizeRegistration}}
  \end{aligned}$$

*Safety:* `true`

### 5.2.2. Interactions Model

The interactions model is given in Table 5.1.

## 5.3. Design

During the design phase; an agent model, a services model and a acquaintances model of the system is achieved based on the roles model and interactions model from the analysis phase.

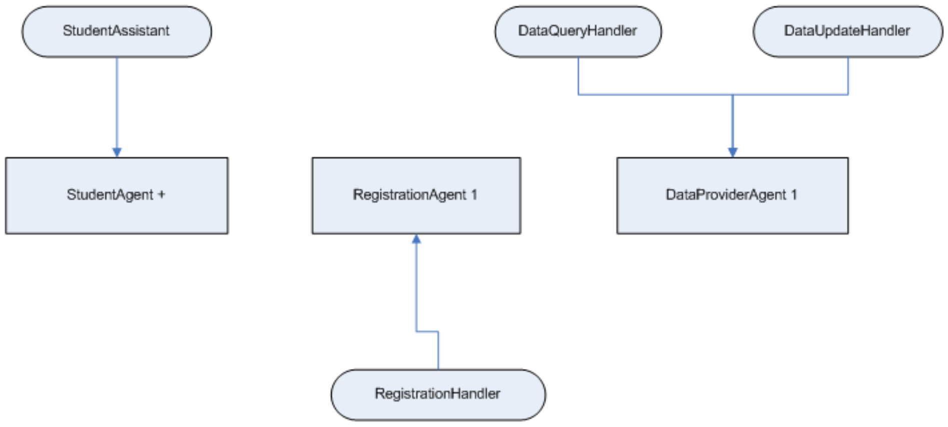

Figure 5.1. Gaia Agent Model for Course Registration System

### 5.3.1. Agent Model

After a study of the Gaia Roles Model three agents are identified to be modeled in the design phase; a *StudentAgent* that corresponds to the *StudentAssistant* role, a *RegistrarAgent* that corresponds to the *RegistrationHandler* role and finally a *DataProviderAgent* that represents the *DataQueryHandler* and *DataUpdateHandler* roles. In our sample system the *RegistrationAgent* and the *DataProviderAgent* will each have single instances for the sake of implementation simplicity. In a more advanced implementation several *RegistrationAgent*s and *DataProviderAgent*s may coexist with proper request routing, load-balancing and synchronization support. The system may host 1 or more *StudentAgent*s. An instance qualifier of **"n..m"** may also be deemed appropriate for *StudentAgent*s.

### 5.3.2. Services Model

Based on the activities and protocols identified in the Roles Model and the Interactions Model the following services are designed for the Services Model. The services model is discussed in three parts; each part corresponding to an agent defined in the Agent Model.

5.3.2.1. StudentAgent Services. The first service for the *StudentAgent* is related to the initialization of the agent and involves the fetching of initialization data (studentProfile and registrationCard) based on a unique key (studentID) identifying the specific agent instance. The studentProfile data includes student identification information such as name, GPA, class, department, etc. The registrationCard data includes registration related data such as course slots available, maximum credits that can be used, etc. Upon retrieval the StudentAgent is initialized using internal methods and joins the agent run-time platform.

| **Protocol** | RequestCourseList | GetStudentProfile | GetRegistrationCard | RequestCourseRegistration | FinalizeRegistration |
|---|---|---|---|---|---|
| **Initiator** | StudentAssistant | StudentAssistant | StudentAssistant | StudentAssistant | StudentAssistant |
| **Receiver** | DataQueryHandler | DataQueryHandler | DataQueryHandler | RegistrationHandler | RegistrationHandler |
| **RespondingAction** | ReturnQueryResult | ReturnQueryResult | ReturnQueryResult | InformStudent | InformStudent |
| **Purpose/ Parameters** | Request list of courses that match a given registration slot | Request profile information regarding a certain student | Request registration card (list of slots to be filled, etc.) for a certain student | Request to register a student to a specified course | Request to finalize the registration of a student |

| **Protocol** | QueryStudentRecord | QueryCourseRecord | CheckDuplicateRegistration | AddStudentToCourse | DropStudentFromCourse |
|---|---|---|---|---|---|
| **Initiator** | RegistrationHandler | RegistrationHandler | RegistrationHandler | RegistrationHandler | RegistrationHandler |
| **Receiver** | DataQueryHandler | DataQueryHandler | DataQueryHandler | DataUpdateHandler | DataUpdateHandler |
| **RespondingAction** | ReturnQueryResult | ReturnQueryResult | ReturnQueryResult | InformRequestOwner | InformRequestOwner |
| **Purpose/ Parameters** | Request details on a specific student | Request details on a specific course | Query the registration relation between a given student and course | Register a student to a specified course | Remove a student from a specified course |

| **Protocol** | FinalizeRegistration | InformStudent | ReturnQueryResult | RejectQueryRequest | InformRequestOwner |
|---|---|---|---|---|---|
| **Initiator** | RegistrationHandler | RegistrationHandler | DataQueryHandler | DataQueryHandler | DataUpdateHandler |
| **Receiver** | DataUpdateHandler | StudentAssistant | StudentAssistant/ RegistrationHandler | StudentAssistant/ RegistrationHandler | RegistrationHandler |
| **RespondingAction** | InformRequestOwner | - | - | - | - |
| **Purpose/ Parameters** | Finalize the registration of a student | Inform owner of a registration request about action result. | Return results of a query request | Reject a query request | Inform the owner of an update request |

| **Protocol** | RejectRequest | | | | |
|---|---|---|---|---|---|
| **Initiator** | DataUpdateHandler | | | | |
| **Receiver** | RegistrationHandler | | | | |
| **RespondingAction** | - | | | | |
| **Purpose/ Parameters** | Reject an update request. | | | | |

Table 5.1. Gaia Interactions Model

| **Service** | **Inputs** | **Outputs** | **Pre-Condition** | **Post-Condition** |
|---|---|---|---|---|
| Obtain student details and initialize student | studentID | studentProfile, registrationCard | true | true |
| Determine target course slot | none | courseSlot | $\neg$ registrationComplete $\wedge$ emptySlotCount $> 0$ | true |
| Request course list for slot | courseSlot | courseList | courseSlot $\neq$ *nil* | courseList $\neq$ *nil* |
| Pick course to register for slot | courseList, courseSlot | courseID | courseSlot $\neq$ *nil* $\wedge$ courseList $\neq$ *nil* | courseID $\neq$ *nil* |
| Register to course | courseSlot, courseID | registrationResult | courseSlot $\neq$ *nil* $\wedge$ courseList $\neq$ *nil* $\wedge$ emptySlotcount $> 0$ | regisrationResult = true |
| Finalize registration | none | registrationStatus | emptySlotCount = 0 $\wedge$ $\neg$ registrationComplete | registrationComplete = true |
| Register to DF (RegistrarAgent) | none | none | true | true |
| Register student to a course | studentID, courseSlot, courseID | registrationResult | studentIsEligible | $studentID \in studentList_{courseID}$ |
| Drop student from a course | studentID, courseSlot, courseID | registrationResult | $studentID \in studentList_{courseID}$ | $studentID \notin studentList_{courseID}$ |
| Finalize registration | studentID | registrationResult | $\neg$ registrationComplete(studentID) | registrationComplete(studentID) = true |
| Inform student | studentID, registrationResult | none | studentID $\neq$ *nil* $\wedge$ registrationResult $\neq$ *nil* | student knows registrationResult |

Table 5.2. Gaia Services Model for Course Registration System (StudentAgent, Registrar Agent)

| | | | | |
|---|---|---|---|---|
| Register to DF (DataProviderAgent) | none | none | true | true |
| Process student records query | studentID | queryResult | true | true |
| Process course list query | courseSlot | queryResult | true | true |
| Process course records query | courseID | queryResult | true | true |
| Process registration records query | studentID, courseSlot, courseID | queryResult | true | true |
| Return query result | queryResult, requester | none | queryResult $\neq nil \wedge$ requester $\neq nil$ | requester receives query result |
| Add student to course records | studentID, courseSlot, courseID | queryResult | true | true |
| Drop student from course records | studentID, courseSlot, courseID | queryResult | $studentID \in studentList_{courseID}$ | $studentID \notin studentList_{courseID}$ |
| Finalize registration (DB) | studentID | queryResult | $\neg$ registrationComplete | registrationComplete = true |
| Inform query request owner | queryResult, requester | none | queryResult $\neq nil$ requester $\neq nil$ | requester is informed |

Table 5.3. Gaia Services Model for Course Registration System (DataProviderAgent)

The second *StudentAgent* service represents the internal activity of choosing a courseSlot from the registrationCard for which the agent will later try to pick a course for. Upon deciding which courseSlot to register for, the third service comes into action and queries the *DataProviderAgent* for a list of courses available for that specific courseSlot. The next service - using internal decision making algorithms - chooses a course from the available courseList which maximizes the benefits the *StudentAgent* is expecting from the registration. The next service creates and issues a registration request to the *RegistrarAgent* for the courseSlot,courseID pair determined by previous services. Finally when the agent has completed filling slots in its registrationCard the final service is used for submitting and finalizing the registration.

5.3.2.2. RegistrarAgent Services. The first service for the *RegistrarAgent* is related to the initialization of the agent and its registration to the DF service in the JADE run-time platform. The initialization of the agent does not require any parameters and is done according to a predefined configuration provided by the system. A registration to the DF is required so that *StudentAgent*s can lookup the *RegistrarAgent* and later communicate with it.

The second and third *RegistrarAgent* services deal with registration requests (for adding and dropping courses respectively) initiated by *StudentAgent*s. Based on the parameters extracted from the request the agent will verify the request and upon a successful verification do post processing (including requesting updates from the *DataProviderAgent*) for completing the request. The next service processes registration finalization request from *StudentAgent*s and follows a path similar to the previous services. The last service informs owners of requests processed about their outcome.

5.3.2.3. DataProviderAgent Services. The first service for the *DataProviderAgent* is related to the initialization of the agent and its registration to the DF service in the

JADE run-time platform. The initialization of the agent does not require any parameters and is done according to a predefined configuration provided by the system. A registration to the DF is required so that *StudentAgent*s and the *RegistrarAgent* can lookup the *DataProviderAgent* and later communicate with it.

The next four services are related to read-only queries that can be requests by other agents in the system. For each record type available for query (student records, course lists, course records and registration records) there is a corresponding service for processing the query. The next service deals with returning query results to respective owners. The following three services are related to requests that require an update to database records (add a student to a course, drop a student from a course and finalize a registration) and contain all update actions required to complete the request. The final service deals with returning update query results to respective request owners.

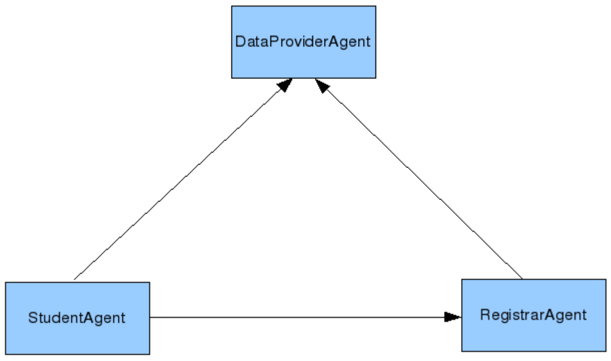

Figure 5.2. Gaia Acquaintance Model for Course Registration System

### 5.3.3. Acquaintance Model

The acquaintance model shows possible interactions paths among agents existing in the system. In the sample system all agents are aware of the existence of each other so no such information is depicted in the diagram, instead only paths that show service consumption directions are given. As it can be seen from the graph and deducted from previous discussions on interaction and services models *StudentAgent*s can interact with both the *RegistrarAgent* and the *DataProviderAgent* consuming services available from these services. However no agent in the system can interact with *StudentAgents*, communication from a *RegistrarAgent* or *DataProviderAgent* to a *StudentAgent* is only possible as a response to a communication initiating from a *StudentAgent*. The *RegistrarAgent* can communicate with the *DataProviderAgent* for query purposes.

## 5.4. Implementation

During implementation agent types conceptualized previously in the complete model definition are used. Namely **StudentAgent**, **DataProviderAgent** and **RegistrarAgent** types of the Maus hierarchy and their corresponding JADEStub Agent implementations are used in the sample applications. Agent services are modeled to agent actions in the **edu.zion.mas.maus.agents.actions** and **behaviours in edu.zion.mas.maus.behaviour** packages. Constraints on agent actions are implemented as rules under the **edu.zion.mas.maus.kb.rules** package. A utility for accessing the internal database is provided in the **edu.zion.mas.maus.util.management** package.

# 6. Conclusion

Multi-agent systems present an interesting approach to designing and implementing complex systems where multiple actors are interacting to satisfy micro- and macro-level goals. Modelling complex behaviours of multiple interacting actors as interactions between agents allows for a better design approach compared to traditional programming approaches. Multi-agent systems also provide valuable tools of modelling and simulating behavioural patterns in real and pseudo societies.

This work concentrates on the possibility of representing a university in the multi-agent domain. To that end two separate efforts are made; first a preliminary multi-agent model of a university is proposed with possible implementation details. The proposed model is to be used as a basis for further studies in building a comprehensive university multi-agent system. The second effort is to provide a practical (but simple) implementation example for a MAS representing course registration activities observed in a university.

Based on the preliminary model suggestions on MAS applications of the model are also discussed. A complete university MAS system can be used for both simulation and practical purposes. In this work four such application ideas are proposed. As a simulation tool the MAS can be used for quota forecasting simulations where simulations runs on training data can be used to determine which courses (or as a general observation subject areas) students will tend to concentrate on. The same simulation system can also be modified to provide an actual course registration system. The rules system proposed in the model can also be used for simulating effects of regulation changes on the university ecosystem. Finally a MAS can be used as an online collaboration and social networking tool.

## References

1. M. Wooldridge, ”An Introduction to MultiAgent Systems”, John Wiley & Sons, Ltd., West Sussex, England (2002).

2. F. Bellifemine, G. Caire, D. Greenwood, ”Developing Multi-Agent Systems with JADE”, John Wiley & Sons, Ltd., West Sussex, England (2007)

3. Natalya F. Noy, Deborah L. McGuinness, ”Ontology Development 101: A Guide to Creating Your First Ontology”, Stanford University

4. M. Wooldridge, N. R. Jennings, D. Kinny, ”The Gaia Methodology for Agent-Oriented Analysis and Design”, England (2000)

5. Pavlos Moraïtis, Eleftheria Petraki, Nikolaos I. Spanoudakis, ”Engineering JADE Agents with the Gaia Methodology”, Cyprus (2003)

6. FIPA TC Communication, ”FIPA ACL Message Structure Specification” [ http://www.fipa.org/specs/fipa00061/SC00061G.html ], Geneva, Switzerland, 2002

## References not Cited

1. F. Zambonelli, N. R. Jennings, M. Wooldridge, "Developing Multiagent Systems: The Gaia Methodology", ACM Transactions on Software Engineering and Methodology, Vol. 12, No.3, July 2003

2. G. Wang, H. Yu, J. Xu, S. Huang, "A Multi-agent Model Based on Market Competition for Task Allocation: a Game Theory Approach, Shenyang, China (2004)

3. Z. Guessoum, "A Hybrid Agent Model: a Reactive and Cognitive Behavior, Paris, France (1997)

4. N. Liu, M. A. Abdelrahman, Srini Ramaswamy, "A Multi-Agent Model for Reactive Job Shop Scheduling", Tennessee Technological University, USA (2004)

5. P. Moraïtis, N. I. Spanoudakis, "Combining Gaia and JADE for Multi-Agent Systems Development", 4th International Symposium "From Agent Theory to Agent Implementation" (AT2AI4), in: Proceedings of the 17th European Meeting on Cybernetics and Systems Research (EMCSR 2004), Vienna, Austria (2004)

# Curriculum Vitae

Önder Özkan was born on November $10^{th}$, 1980, in İstanbul. He received his BS degree in Information Technologies in 2005 and BS degree in Computer Sciences and Engineering in 2006 both from Işık University. He worked as a research assistant at the department of Information Technologies of Işık University from 2005 and 2008. His research interests include multi-agent modelling and multi-agent systems, mobile agent technology and content management systems.