The Gaia2JADE Process for Multi-Agent Systems Development

Pavlos Moraitis

Dept. of Computer Science
University of Cyprus
75 Kallipoleos Str.,
1678 Nicosia, Cyprus

European Projects Dept.
Singular Software S.A.
26th October 43, 54626,
Thessaloniki, Greece

Nikolaos Spanoudakis

European Projects Dept.
Singular Software S.A.
26th October 43, 54626,
Thessaloniki, Greece

LAMSADE
University of Paris IX-Dauphine
Place du Marechal de Lattre de Tassigny
75775 Paris CEDEX 16, France

***Abstract**: In this paper we present the Gaia2JADE process concerning how one can implement amulti-agent system with the JADE framework using the Gaia methodology for analysis and design purposes. This process is particularly dedicated to the convertion of Gaia models to JADE code. It is described using the Software Process Engineering Metamodel (SPEM) and extends the one proposed by FIPA for describing the Gaia modelling process. Thus, it proposes to potential MAS developers a process that covers the full software development lifecycle. This work is based on the experience we have acquired by applying this process for implementing a real word multi-agent system conceived for providing e-services to mobile users. With this paper, we share this experience with future multi-agent systems (MAS) developers, who would like to follow this process, taking into account several technical issues that emerged during the implementation phase, helping them to easily model and implement their systems.*

1. Introduction

During the last few years, there has been a growth of interest in the potential of agent technology in the context of software engineering. Some promising agent-oriented software development methodologies, as for example Gaia (Zambonelli et al. 2003) and MaSE (Wood and DeLoach, 2000) have been proposed but they cover only the requirements (MaSE), analysis and design phases (MaSE, Gaia) of the software development cycle (Sommerville, 2000). An exception in these works is Tropos (Bresciani et al., 2003), which in its recent version proposes the covering of the entire software development process. Recently, there have also been some attempts to provide roadmaps (e.g. Moraitis et al, 2003a) and tools (e.g. Cossentino et al, 2003, Gomez-Sanz and Pavon, 2003) for allowing analysis and design methodologies to be implemented using JADE (Bellifemine et al, 2003) or the FIPA-OS (Emorphia Ltd, 2003) open source frameworks. Other frameworks appear as complete solutions (methodology plus agents development environment) like Zeus (Collis and Ndumu, 1999) or AgentTool with MaSE (DeLoach and Wood, 2001), but they couldn't gain as large an audience as JADE at the implementation level.

In this paper we present the Gaia2JADE process (a preliminary version was presented in Moraitis et al., 2003a), which we used (see Moraitis et al., 2003b) in order to develop a real-world multi-agent system (MAS) that was analyzed and designed using the Gaia methodology and implemented with the JADE framework. The output of our experience is this process which allows developers to implement Gaia models using the JADE framework. The weak and strong points of Gaia when it comes to implementation using JADE were recognized and we can now propose a detailed process so that MAS developers can easily model and implement their systems. The Gaia2JADE process was described using the Software Process Engineering Metamodel (SPEM) proposed by the Object Management Group (2002). The FIPA Methodology Technical Committee has used it in order to describe the Gaia modelling process (Garro et al, 2004). We enhanced the latter, by adding the JADE development phase, thus proposing to potential MAS developers a process that covers the full software development lifecycle.

This paper is organized in the following way. In sections 2 and 3 we provide a Gaia and JADE overview. In section 4 we present the Gaia2JADE development process in detail. In section 5 we present a case study, where we instantiate the proposed process in a specific application. Finally, section 6 includes conclusions and future work.

2. Gaia Overview

The Gaia methodology (Zambonelli et al. 2003) is an attempt to define a complete and general methodology that it is specifically tailored to the analysis and design of MASs. Gaia supports both the levels of the individual agent structure and the agent society in the MAS development process. MASs, according to Gaia, are viewed as being composed of a number of autonomous interactive agents that live in an organized society in which each agent plays one or more specific roles. Gaia defines the structure of a MAS in terms of a role model. The model identifies the roles that agents have to play within the MAS and the interaction protocols between the different roles. The Gaia modelling methodology is a three phase process and at each phase the modelling of the MAS is further refined. These phases are the analysis phase, the architectural design phase and, finally, the detailed design phase.

The objective of the Gaia analysis phase is the identification of the roles and the modelling of interactions between the roles found. Roles consist of four attributes: *responsibilities, permissions, activities and protocols*. Responsibilities are the key attribute related to a role since they determine the functionality. Responsibilities are of two types: *liveness properties* – the role has to add something good to the system, and safety properties – the role must prevent something bad from happening to the system. Liveness describes the tasks that an agent must fulfil given certain environmental conditions and safety ensures that an acceptable state of affairs is maintained during the execution cycle. In order to realize responsibilities, a role has a set of permissions. *Permissions* represent what the role is allowed to do and, in particular, which information resources it is allowed to access. The *activities* are tasks that an agent performs without interacting with other agents. Finally, *protocols* are the specific patterns of interaction, e.g. a seller role can support different auction protocols. Gaia has formal operators and templates for representing roles and their attributes and also it has schemas that can be used for the representation of interactions between the various roles in a system. The operators that can be used for liveness expressions-formulas along with their interpretations are presented in Table 1. Note that activities are written underlined in liveness formulas.

Operator	Interpretation
x . y	x followed by y
x | y	x or y occurs
x*	x occurs 0 or more times
x+	x occurs 1 or more times
x^{ω}	x occurs infinitely often
[x]	x is optional
x || y	x and y interleaved

Table 1. Gaia Operators for Liveness Formulas

Furthermore, during the analysis phase, the possible interactions with a role's external environment are identified and documented in the environmental model. There, the possible actions that the role can perform to the environment along with the perceptions that it can receive are identified. It is a computational representation of the environment in which the MAS will be situated.

Finally, the rules that the organization should respect and enforce in its global behavior are defined. These rules express constraints on the execution activities of roles and protocols and are of primary importance in promoting efficiency in design and in identifying how the developing MAS can support openness and self-interested behavior.
In a next phase, namely the architectural design phase, the roles and interactions models are refined and finalised by the definition of the system's organizational structure in terms of its topology and control regime. This activity involves considering the organizational efficiency, the real-world organization in which the MAS is situated, and the need to enforce the organizational rules.

Lastly, the Gaia detailed design phase, maps roles into agent types and specifies the right number of agent instances for each type. Thus, an agent type is an aggregation of one or more agent roles. Moreover, during this phase, the services model, the services that a role fulfils in one or several agents, is described. A service can be viewed as a function of the agent and can be derived from the list of protocols, activities, responsibilities and the liveness properties of a role.

The FIPA Methodology Technical Committee (Garro et al, 2004) defined the process of analyzing and designing a MAS using Gaia by employing the Software Process Engineering Metamodel (SPEM), a standard developed by the Object Management Group (2002).

3. JADE Overview

JADE (Bellifemine et al, 2003) is a software development framework fully implemented in Java language aiming at the development of multi-agent systems and applications that comply with FIPA standards for intelligent agents. 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 of which executes one Java Virtual Machine.
- FIPA-Compliant agent platform, which includes the Agent Management System the Directory Facilitator and the Agent Communication Channel (FIPA TC Agent Management, 2002).
- Efficient transport of agent communication language (ACL) messages between agents (FIPA TC Communication, 2002).

All inter-agent communication is performed through message passing and the FIPA ACL is the language that is used to represent the messages. 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 message filtering. The developer can apply advanced filters on the various fields of the incoming messages such as sender, performative or ontology.

FIPA specifies a set of standard interaction protocols such as FIPA-request, FIPA-query, etc. that can be used as standard templates to build agent conversations. For every conversation among agents, JADE distinguishes the role of the agent that starts the conversation (initiator) and the role of the agent that engages in a conversation started by another agent (responder). According to the structure of these protocols, the initiator sends a message and the responder can subsequently reply by sending a not-understood or a refuse message indicating the inability to achieve the rational effect of the communicative act, or an agree message indicating the agreement to perform the communicative act. When the responder performs the action he must send an inform message. A failure message indicates that the action was not successful. JADE provides ready-made behaviour classes for both roles, following most of the FIPA specified interaction protocols (FIPA TC Communication, 2002). JADE provides the *AchieveREInitiator* and *AchieveREResponder* classes, a single homogeneous implementation of interaction protocols such as these mentioned above. Both classes provide methods for handling all possible protocol states.
In JADE, agent tasks or agent intentions are implemented through the use of behaviours. Behaviours are logical execution threads that can be composed in various ways to achieve complex execution patterns and can be initialized, suspended and spawned at any given time. The agent core keeps a task list that contains the active behaviours. JADE suggests the use of one thread per agent instead of one thread per behaviour to limit the number of threads running in the agent platform. A scheduler, hidden to the developer, carries out a round robin policy among all behaviours available in the queue. The behaviour can release the execution control with the use of blocking mechanisms, or it can permanently remove itself from the queue in run time. Each behaviour performs its designated operation by executing the core method *action()*.

Behaviour is the root class of the behaviour hierarchy that defines several core methods and sets the basis for behaviour scheduling as it allows state transitions (starting, blocking and restarting). The children of this base class are *SimpleBehaviour* and *CompositeBehaviour*. The classes that descend from *SimpleBehaviour* represent atomic simple tasks that can be executed a number of times specified by the developer. Classes descending from *CompositeBehaviour* support the handling of multiple behaviours according to a policy. The actual agent tasks that are executed through this behaviour are not defined in the behaviour itself, but inside its children behaviours. The *FSMBehaviour* class, which executes its children behaviours according to a Finite State Machine (FSM) of behaviours, belongs in this branch of hierarchy. Each child represents the activity to be performed within a state of the FSM, with the transitions between the states defined by the developer. Because each state is itself a behaviour it is possible to embed state machines. The *FSMBehaviour* class has the responsibility of maintaining the transitions between states and selects the next state for execution. Some of the children of an *FSMBehaviour* can be registered as final states. The *FSMBehaviour* terminates after the completion of one of these children.

The developer creates his agents by extending the JADE *Agent* class. He can add any number of behaviours along with defining the agent's initialization and termination handling functionality. A special descendant of the *Agent* class, the *GUIAgent*, allows for the creation of agents with a graphical user interface (GUI), allowing for the agent's interaction with a human user. The latter is facilitated by a GUI event exchange mechanism that also allows the definition of parameters that accompany the event. Whenever a specified GUI event occurs the agent can add a new behaviour passing to its constructor the relevant parameters and a reference to the GUI so that the behaviour can reply to the user.

4. The Gaia2JADE Process

In this section we present the Gaia2JADE process for implementing the Gaia models using the JADE framework. It presupposes that a Gaia model is ready and that JADE framework is chosen for implementation purposes. In order to better present the process we described it using the SPEM (OMG, 2002) specification, as a JADE development phase. This process can be merged with the one provided by Garro et al (2004) as a next phase. In Figure 1 the JADE implementation phase has been added to the Gaia process, thus providing the complete Gaia2JADE process.

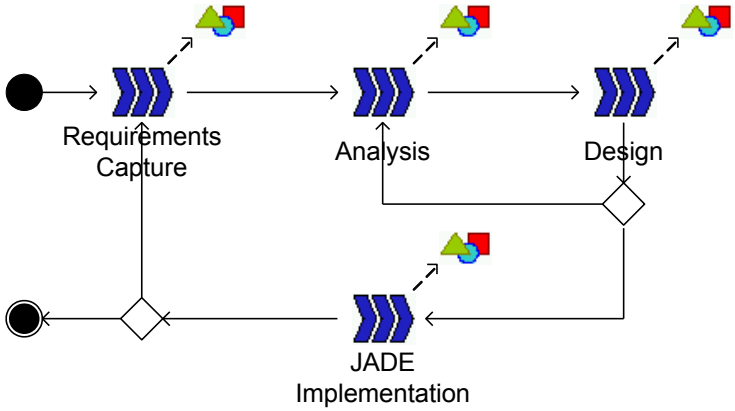

Figure 1: The phases of the Gaia2JADE process

The Gia2JADE process aggregates four process packages, the last one being the one we propose herein. The JADE implementation process package involves the developer role, that is the person(s) that will implement the Gaia models using the JADE framework and produces two work products, the Java code and a repository of behaviours (see Figure 2).

Here it must be noted that according to FIPA (FIPA TC Agent Management, 2002) there are some default roles involved in the MAS operation. They are the Directory Facilitator (DF) and the Agent Management System (AMS) roles that are supported by JADE. However, these roles concern the operational level of the MAS and not the application itself. Therefore, a Gaia modeler should not prepare a Gaia representation for these roles. Interactions between the application specific roles and the FIPA defined roles should not be modeled as protocols, but as activities. Indeed, activities like registration to the DF or querying for agents of specific types are DF services that

are provided in the form of function calls by the JADE framework. The JADE framework automatically handles the relevant messages exchange between the agent and the DF and returns the outcome to the agent. If, though, a Gaia modeler does create such roles and the relevant protocols, the JADE developer should replace those by relevant activities (e.g. registration to the DF) that will be directly mapped to JADE method calls.

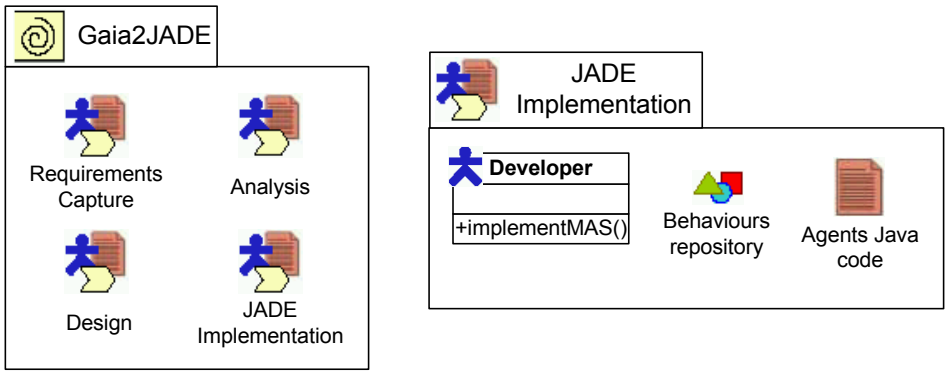

Figure 2: The Gaia2JADE process and the JADE implementation process package

We are now ready to present the JADE implementation phase that can be described as a four step process:

1. Define the communication protocol, meaning the definition of all the necessary ACL messages along with the definition of the possible ordering of their exchange.
2. Define the activities refinement table, where application dependent data, their structure and algorithms that are going to be used by the agents are defined.
3. Define the JADE behaviours
4. Construct the agent classes

Note that this phase produces more than the agents' java code. The JADE behaviours are another product since they are reusable pieces of code (components) that can be used for building agents or other complex behaviours.

The overall development process is, thus, top-down in the analysis and design phase (i.e. by using Gaia) and bottom-up in the implementation phase, according to the most successful software engineering practices (Sommerville, 2000). The JADE implementation phase is presented graphically in Figure 3. All steps resemble a SPEM work definition.

The "define communication protocols" work definition involves developing the problem domain ontology that will be used in order to refine the protocols of the Gaia interaction model and produce the ACL messages with respect to the FIPA ACL Message Structure Specification (FIPA TC Communication, 2002). AUML sequence diagrams (Odell et al, 2001) can be very useful in the case of complex protocols.

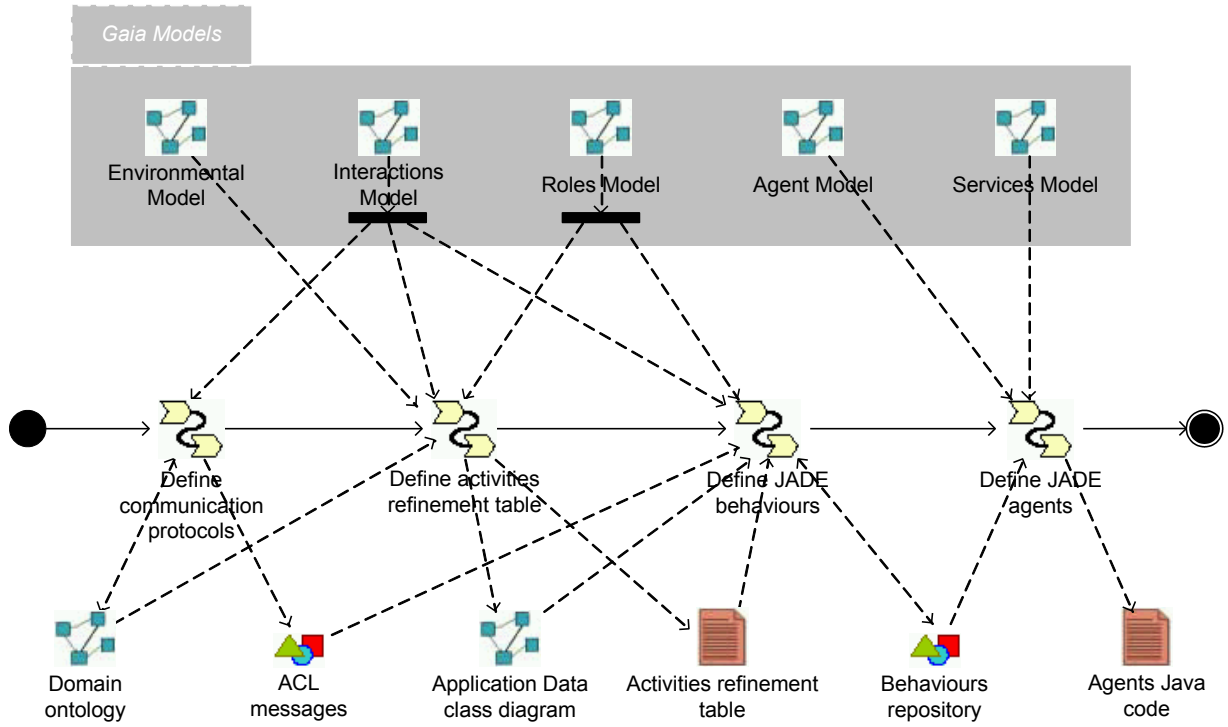

Figure 3: The JADE implementation phase

The next work definition, namely the "define activities refinement table" is about the definition of the application specific data and their structure, algorithms for liveness formulas activities and possible interfaces to external systems. It defines the needed classes for encapsulating the application data that will be used by the agents, preferably in UML format since these will be implemented using the object oriented Java. In order to do that, the domain ontology is taken into account. If the Protégé tool (Noy et al, 2001) is used for ontology development, then all the ontology classes can be extracted as Java classes for use by the agents. The classes names are used in order to represent such data to the activities refinement table. In this table, the necessary algorithms that are used by each activity described in the liveness properties of each role in the Gaia roles model are documented. We present an example of an activities refinement table record later in §5.4. The environmental Gaia model is used here so that the relevant interfaces to external application are also defined. During this process the safety conditions of each role are taken into account. Since safety conditions must be valid for normal role functionality, the implications of their failure must be explicitly defined. Thus, for example, it is possible that a safety condition failure results in a relevant message to an administrator user.

The "define JADE behaviours" work definition is where the coding procedure begins. A detailed work description is presented in Figure 4, since it is the most complicated work and the real mapping of Gaia to JADE features takes place here. All activities further extend the behaviours repository work product. For the reader's convenience we "zoom" in the work product after each activity in order to see the type of extension.

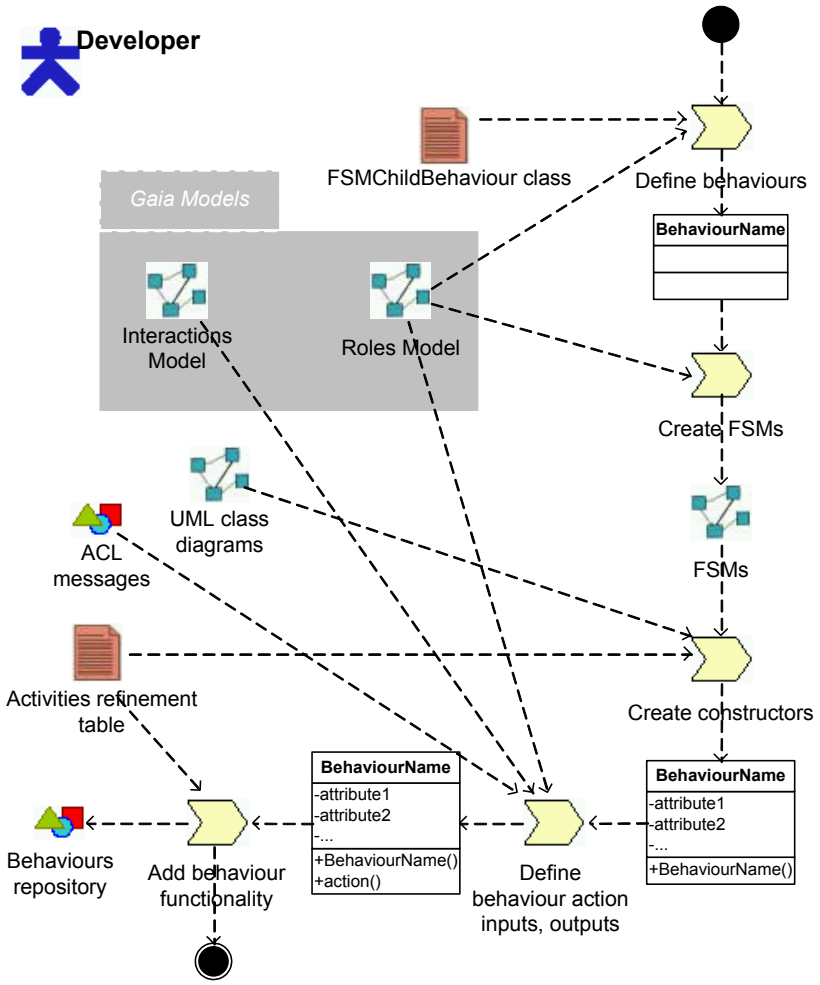

Figure 4: The "define JADE behaviours" work definition

The first activity is to define the JADE behaviours that are to be implemented. In order to do this we need to map some key Gaia concepts, identified mainly in the roles model, to JADE concepts. Let's consider the liveness part of each role (in the Gaia roles model) as its behaviour (usually having the same name with the role) in correspondence with the JADE terminology. Thus, a simple or a complex behaviour will represent each role. This behaviour is considered as the top-level behaviour of the role. Each behaviour may contain other behaviours, as in the JADE behaviours model. Let the contained behaviours be called lower level behaviours. The contained behaviours (lower level) are those on the right side of the Gaia roles model liveness formulas (see Figure 5 for an illustrative presentation of this procedure). Generally, complex behaviours are implemented as descendants of the JADE *CompositeBehaviour* and its subclasses (e.g. *FSMBehaviour*), while simple behaviours are implemented as descendants of the JADE *SimpleBehaviour* class. Note that the *Activity1* behaviour of our example is lower to *Role1* behaviour, but upper to *Activity3* behaviour. A behaviour that is an upper behaviour (on the left hand side) in any liveness formula of a role is a complex behaviour.

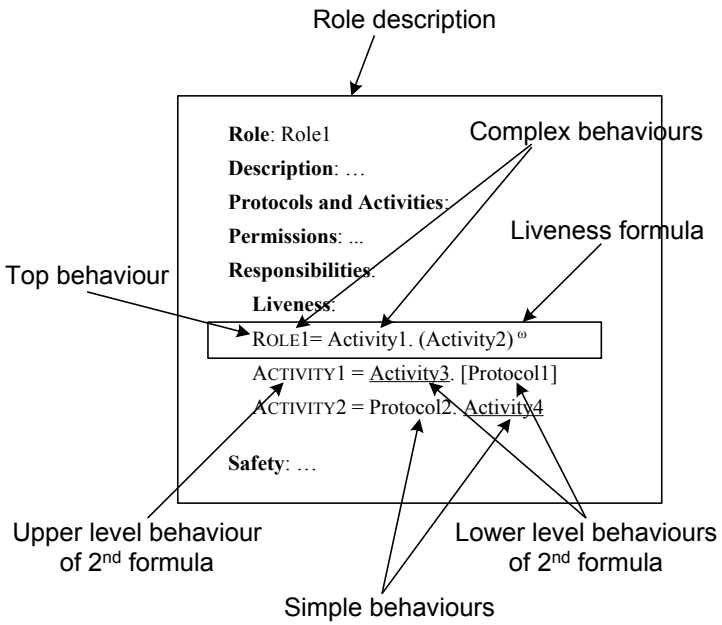

Figure 5: The transformation of liveness formulas to JADE behaviours

The ω and || operators on Gaia liveness formulas now have the following meaning. The ω means that a lower level behaviour is added by the behavior that contains it in the Gaia liveness formula and is only removed from the agent's scheduler when the behavior that added it, is removed itself. If such behaviours (that are added to the agent scheduler) are more than one, they are connected with the || symbol which denotes that they execute "concurrently". Concurrency in JADE agent behaviours is simulated. As noted before, only one thread executes per agent and behaviour actions are scheduled in a round robin policy.

As a rule of thumb, we found out that the "." operator in a liveness formula denotes that the behaviour at the left hand side (of the formula) is a complex behaviour (one that aggregates one or more others), while the [ ], +, *, | operators denote that the left hand side behaviour (of the formula) can be implemented as a finite state machine (*FSMBehaviour*).

The developers should start implementing the behaviours of the lowest levels (activities in the Gaia roles model), using the various Behaviour class descendants provided by JADE.

Finally, we propose the *FSMChildBehaviour* class (see Figure 6) that helps in automating a lot of repeating code in simple behaviours within FSM behaviours (*FSMBehaviour* class instances). This class defines two useful attributes, *finished* and *onEndReturnValue* and implements the methods *done* (returns *true* if a behaviour has finished, so that it is not inserted again in the agent behaviour scheduler) and *onEnd* (returns the state of the behaviour when it stopped executing). The *FSMChildBehaviour* class is extended by behaviours that are going to be used by FSM behaviours. These behaviours would normally need to implement the *done* and *onEnd* methods along with the *action* methods,

the latter implementing their functionality. By extending the *FSMChildBehaviour* class, they now only need to implement the action methods.

```
package image.agents;
import jade.core.behaviours.SimpleBehaviour;
import jade.core.Agent;
public class FSMChildBehaviour extends SimpleBehaviour {
  protected boolean finished = false;
  protected int onEndReturnValue;
  public FSMChildBehaviour(Agent a) {
    super(a);
  }
  public void action() {};
  public boolean done() {
    return finished;
  }
  public int onEnd(){
    return onEndReturnValue;
  }
}
```

Figure 6: The *FSMChildBehaviour* class

The next activity concerns the creation of state diagrams in order to model FSM-like behaviours and recognize the common application data classes that are used by the lower level behaviours. After all FSM diagrams are ready, the "create constructors" activity initializes the necessary data classes at the upper level behaviour and pass them as parameters to lower level behaviours by creating the JADE behaviours' constructors. Thus, the behaviour repository classes have attributes and constructors after the completion of this activity.

The next activity is about behaviours that are activated on the receipt of a specific message that can be either a graphical user interface (GUI) event or the arrival of a message by another agent. Those that expect an inter-agent message must receive it (with the appropriate message filtering template) at the start of their action. For behaviours that start by an event from the GUI, the relevant parameters must be configured and added to their constructor and attributes list. A GUI event receiver method will be later implemented on the agent that starts the corresponding behaviour. After this step each behaviour has its action method ready with its input and output functionality implemented.

In the last activity of this work definition, each behaviour is enriched with the desired functionality. Actually, the developer implements the algorithms defined in the activities refinement table within the *action* method of the behaviour. Now the behaviours repository has all needed behaviours.

In the final step of the JADE implementation phase, the agent class is constructed by aggregating behaviours from the relevant repository. Moreover, the developer initializes all agent data classes and adds all behaviours of the lower level in the agent scheduler at the setup method of the JADE *Agent* class. The GUI events should all be caught in this level and add the corresponding event receiver behaviour.

The Gaia Services model should now be satisfied by the implemented MAS. It can be used in order to test the adequate functionality of the system. In case that there are errors the process should return to a previous step and be repeated from then on (iteratively).

5. A Case Study

In order for the reader to better understand how the Gaia2Jade process works, we will present a limited version of the multi-agent system (MAS) that was conceived and implemented in the framework of the IST IMAGE project. A preliminary version of this work is presented in (Moraitis et al, 2003a). We will show how this system was analyzed, designed and implemented. The aim of this system was to provide e-services for mobile users and it included the software development of different modules, one of which was the MAS, namely the Intelligent Module (IM).

This MAS was analyzed and designed using the Gaia methodology and then was implemented using the JADE framework. The full system capabilities, architecture and functionalities, along with the business model and requirements can be found in (Moraitis et al, 2003b).

The reader should note that the purpose of the case study is not to provide guidelines as to how one can model a system using Gaia. Its purpose is to demonstrate the Gaia2JADE process. However, we try to fully address the software development process from the requirements capturing phase to implementation, mostly in order to discuss engineering issues.

5.1. The System Requirements

The Image system requirements that are relevant to MAS development are:

- A user can request a map with his position on it and, possibly other points of interest (POIs) around him that can belong to different types (e.g. banks, restaurants, etc). A user can request for a map with few or even no parameters.
- A user can request a route from a specific place to another specific place, specifying the means of travel (e.g. public transport, car, on foot) and, possibly, the desired optimization type (e.g. shortest, fastest, cheapest route). He can select among a variety of routes that are produced by the Geographical Information System (GIS). A user can request for a route with limited or even no parameters.

- The MAS maintains a user profile so that it can filter the POIs or routes produced by the GIS and send to the user those that most suit his interests. The profiling is based on criteria regarding the preferred transport type (private car, public transport, bicycle, on foot) and the preferred transport characteristics (shortest route, fastest route, cheapest route, etc). Moreover, as far as the POI types are concerned, the system not only allows the user to store in his profile the types that he/she is interested in, but it also exhibits self-learning ability in order to learn the user's preferences by monitoring his behaviour and adapting the service to his needs.
- The system keeps track on selected user routes aiming to receive traffic events (closed roads) and check whether they affect the user's route (if that is the case then inform the user).

5.2. The Analysis and Architectural Design Phases

The Gaia methodology proved to be robust, reliable and the produced models and schemata were used throughout the project development phases as a reference. Moreover, it proved to be flexible enough, so that it was easy to iterate through the design and implementation phases, as is demanded by modern information systems development. The analysis phase led to the identification of four roles: *EventsHandler* that handles traffic events, *TravelGuide* that wraps the GIS, *PersonalAssistant* that serves the user and, finally, *SocialType* that handles other agent contacts. A Gaia roles model for our system is presented in Table 2. We must note that interactions with the Directory Facilitator (DF) FIPA agent are presented as activities since JADE allows for using DF services by method invocations (e.g. *QueryDF*). Note that any role can, through its permissions property, establish connection points with its environment. In this case we have connections with external systems, namely the on-line traffic database and the GIS (these connections were initially identified in the Gaia environmental model).

The Gaia interaction model denotes actions to be undertaken after possible messages exchange between the involved agents. Figure 7 holds the necessary information for our model. However, we considered that the Gaia interaction model wasn't appropriate to represent complex interaction protocols. Gaia encourages the inclusion of such protocols in the liveness properties of a role. We overcame this difficulty by creating scenarios using AUML sequence diagrams (Odell et al, 2001, Zambonelli et al., 2003) in order to write down complex liveness formulas (like the *WhereAmI* of the *PersonalAssistant* role – see Table 2).

5.3. The Detailed Design Phase

During this phase the Gaia Agent and Services models were achieved. The Agent model creates agent types by aggregating roles. Each emerging agent type can be represented as a role that combines all the aggregated roles attributes (activities, protocols, responsibilities and permissions). The agents' model for our system will include three agent types: the *personal assistant agent* type, who fulfills the *PersonalAssistant* and *SocialType* roles, the events *handler agent* type, who fulfills the *EventsHandler* and *SocialType* roles and the *travel guide agent* type, who fulfills the *TravelGuide* role. The *SocialType* role is realized by all agents that need to keep a contact list of other agents. Thus, the personal assistant agent needs to know the travel guide agents so that he requests for their services

and the events handlers need to know the personal assistants in order to forward information to them. The travel guide agent, on the other hand, doesn't need to know other agents because he just fulfils all incoming requests.

Role: SocialType (ST) **Description**: It requests agents that perform specific services from the DF. It also gets acquainted with specific agents. **Protocols and Activities**: RegisterDF, QueryDF, SaveNewAcquaintance, IntroduceNewAgent. **Permissions**: create, read, update acquaintances data class. **Responsibilities**: **Liveness**: SOCIALTYPE = GetAcquainted. (MeetSomeone)$^{\omega}$ GETACQUAINTED = RegisterDF. QueryDF. [IntroduceNewAgent] MEETSOMEONE = IntroduceNewAgent. SaveNewAcquaintance **Safety**: true
Role: PersonalAssistant (PA) **Description**: It acts on behalf of a profiled user. Provides the user with personalized routing and mapping services. These routes are presented to the user. Moreover, it can adapt (i.e. using learning capabilities) to a user's habits by learning from user selections. Finally, it receives information on traffic events, it checks whether such events affect its user's route and in such a case it informs the user. **Protocols and Activities**: InitUserProfile, DecideOrigin, DecidePOITypes, DecidePOIs, DecideDestination, LearnByUserSelection, CheckApplicability, PresentEvent, UserRequest, RespondToUser, InformForNewEvents, FindRoutes, ProximitySearch, CreateMap, GetPOIInfo **Permissions**: create, read, update user profile data class, read acquaintances data class. **Responsibilities**: **Liveness**: PERSONALASSISTANT = InitUserProfile. ((ServeUser)$^{\omega}$ || (ReceiveNewEvents)$^{\omega}$) RECEIVENEWEVENTS = InformForNewEvents. CheckApplicability. [PresentEvent] SERVEUSER = UserRequest. (PlanATrip | WhereAmI). LearnByUserSelection WHEREAMI = DecideOrigin. [GetPOIsInfo] [DecidePOITypes. [ProximitySearch. DecidePOIs. [GetPOIsInfo. GeocodeRequest]] CreateMap] RespondToUser PLANATRIP = DecideOrigin. [GetPOIsInfo] [DecideDestination. [ProximitySearch. [DecidePOIs. GetPOIsInfo. GeocodeRequest]]] [FindRoutes. DecideRoutes. [CreateMap]] RespondToUser **Safety**: true
Role: EventsHandler (EH) **Description**: It acts like a monitor. Whenever a new traffic event is detected it forwards it to all personal assistants. **Protocols and Activities**: CheckForNewEvents, InformForNewEvents. **Permissions**: read on-line traffic database, read acquaintances data class. **Responsibilities**: **Liveness**: EVENTSHANDLER = (CheckForNewEvents. InformForNewEvents)$^{\omega}$ **Safety**: A successful connection with the on-line traffic database is established.
Role: TravelGuide (TG) **Description**: It wraps a Geographical Information System (GIS). It can query the GIS for routes, from one point to another. **Protocols and Activities**: RegisterDF, QueryGIS, InvokeGetRouteGISFunction, InvokeGetNearbyPOIsGISFunction, InvokeGetMapGISFunction, InvokeGetPOIsInfoGISFunction, RequestRoutes, RespondRoutes, RequestMap, RespondMap, RequestNearbyPOIs, RespondNearbyPOIs, RequestPOIsInfo, RespondPOIsInfo **Permissions**: read GIS. **Responsibilities**: **Liveness**: TRAVELGUIDE = RegisterDF. ([FindRoutes] || [ProximitySearch] || [CreateMap] || [GetPOIInfo])$^{\omega}$ FINDROUTES = RequestRoutes. InvokeGetRouteGISFunction. RespondRoutes PROXIMITYSEARCH = RequestNearbyPOIs. InvokeGetNearbyPOIsGISFunction. RespondNearbyPOIs CREATEMAP = RequestMap. InvokeGetMapGISFunction. RespondMap GETPOISINFO = RequestPOIsInfo. InvokeGetPOIsInfoGISFunction. RespondPOIsInfo **Safety**: A successful connection with the GIS is established.

Table 2: The Gaia roles model

There will be one travel guide agent, as many personal assistants as the users of the system and zero or more events handlers. The Agent model is presented graphically in Figure 8. The services model for our system is presented in Table 3.

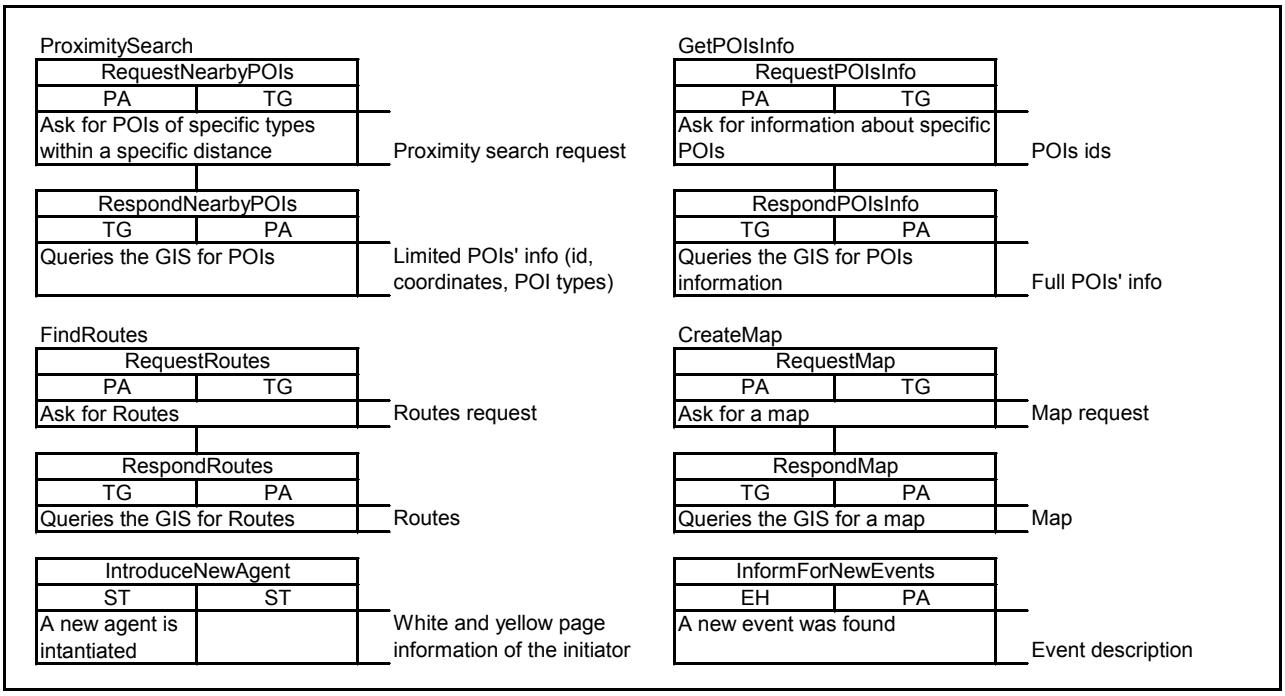

Figure 7: Gaia interactions model

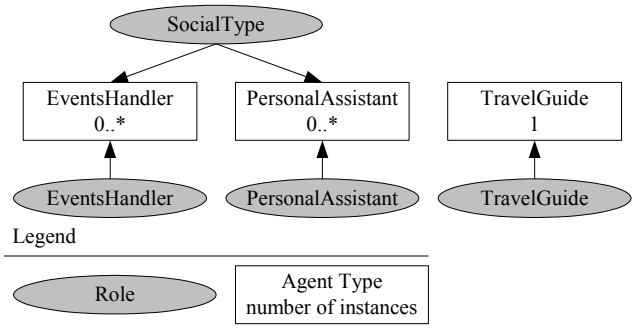

Figure 8: Gaia agent model

At this point the abstract design of the system was complete, since the limit of Gaia had been reached. More effort needed to be done in order to obtain a good design though. At the end of the design process the system should be ready for implementation.

5.4. JADE Implementation Phase

The JADE implementation phase starts with the definition of the domain ontology (transport and tourism domain), followed by the definition of the communication protocol. As we said in section 4, this step concerns the definition of the ACL messages through which the agents will realize the interactions presented in the Gaia interactions

diagram. The ACL messages were defined according to the FIPA specifications (see FIPA TC Communication, 2002).

Service	***Obtain a map***	***Obtain route***	***Get traffic event***
Inputs	Origin, [POI types], [visibility radius]	Origin, [destination], [travel means/characteristics]	-
Outputs	A map, [information about POIs shown on the map]	A set of routes	The description of the event
Pre-condition	A personalized assistant agent is instantiated and associated with the user	A personalized assistant agent is instantiated and associated with the user	A personalized assistant agent is instantiated and associated with the user. The user has selected a route to somewhere. A traffic event that is relevant to the user's route has happened
Post-condition	-	User selects a route	-

Table 3: Gaia services model (items in brackets are optional)

In the next step, each activity in the Gaia roles model was described with regard to its functionality that is what data classes it accesses and what it does with them. The activities refinement table helped us document this step's outcome. As an example, the *DecidePOITypes* activity of the PersonalAssistant role refinement is presented in Table 4. The *DecidePOITypes* activity will be followed either by the *ProximitySearch* or by the *CreateMap* protocols depending on whether the user has requested a specific POI type to view around him (e.g. hotels), or such information can be found in his profile.

Role	**Activities**	**Data Classes**		**Description**
		Read	**Update**	
PA	***DecidePOITypes***	user profile user request	-	**if** UserRequest.POITypes.length>0 **Then** ProximitySearch(UserRequest.POITypes) **else if** UserProfile.POITypes.length>0 **then** ProximitySearch(UserProfile.POITypes) **else** CreateMap **endif** **endif**

Table 4: The Gaia roles' activities refinement table

As we explained in Section 4, we preferred the use of UML class diagrams in order to model the application data classes that would be used by each role's permissions field and defined interfaces for external services usage (GIS, database, etc). The roles' safety conditions were also taken into account so that the behaviours functioned in a determined way whenever the former failed. For the *TravelGuide* role, for example, we decided that whenever a

connection with the GIS fails, the system administrator should be informed about it with a dialog. For the *EventsHandler* role the same dialog was used in order to inform the administrator about connectivity problems with the events database. The activities refinement table was updated accordingly, for example the action of sending a relevant GUI event was added to the *InvokeGetPOIsInfoGISFunction* (activity of the *TravelGuide* role, see Table 2) algorithm.

The following step was the definition of the JADE behaviours. Complex Gaia roles model activities (like the *PlanATrip* and *WhereAmI* activities of the *PersonalAssistant* role) were implemented as composite JADE behaviours (i.e. *FSMBehaviour*). Our first activity within this step was to define the behaviours that we would have to implement. In Figure 9 we present the behaviour mapping activity graphically for one of the lower level liveness formulas of the *PersonalAssistant* role.

Having identified some complex behaviours we proceeded to the definition of the state diagrams that would help us on one hand to identify exchanged data between behaviours and on the other hand to easily model the corresponding JADE *FSMBehaviour*. The state diagram for the *WhereAmI* behaviour of the *PersonalAssistant* role is presented in Figure 10. The necessary application data classes for the *WhereAmI* behaviour were the user request, the user response, the user profile and history meta-data (from where missing information is derived), the agent's acquaintances (from which the different sub-behaviours will find the relevant contacts for achieving the *GetPOIsInfo*, *ProximitySearch* and *CreateMap* protocols) and, finally, the different states identification numbers that are returned by each finishing sub-behaviour and allow the *FSMBehaviour* to decide which behaviour is next to be added to the agent's scheduler. These data classes, according to the Gaia2JADE process, were initialised at the constructor of the FSM behaviour. All sub-behaviours were defined as descendants of the *FSMChildBehaviour* class (see Figure 6).

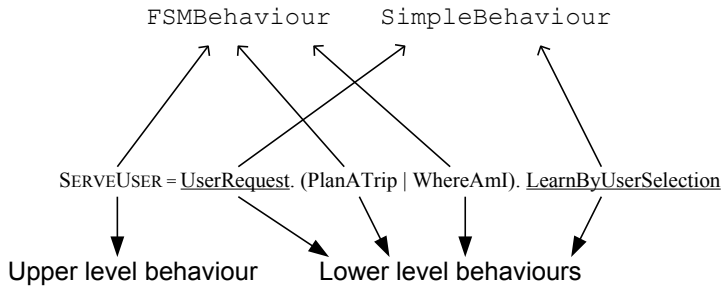

Figure 9: Mapping Gaia role model liveness formulas to JADE behaviours

Starting by implementing the lower behaviours first, we reused some of them while implementing the higher level behaviours. Such behaviours were the *RequestMap* and *RespondMap* behaviours that were used in order to implement the *CreateMap* protocol that is used by both the *PlanATrip* and *WhereAmI* behaviours. The three

remaining activities in the "define JADE behaviours" work definitions were easily carried out as pure programming tasks.

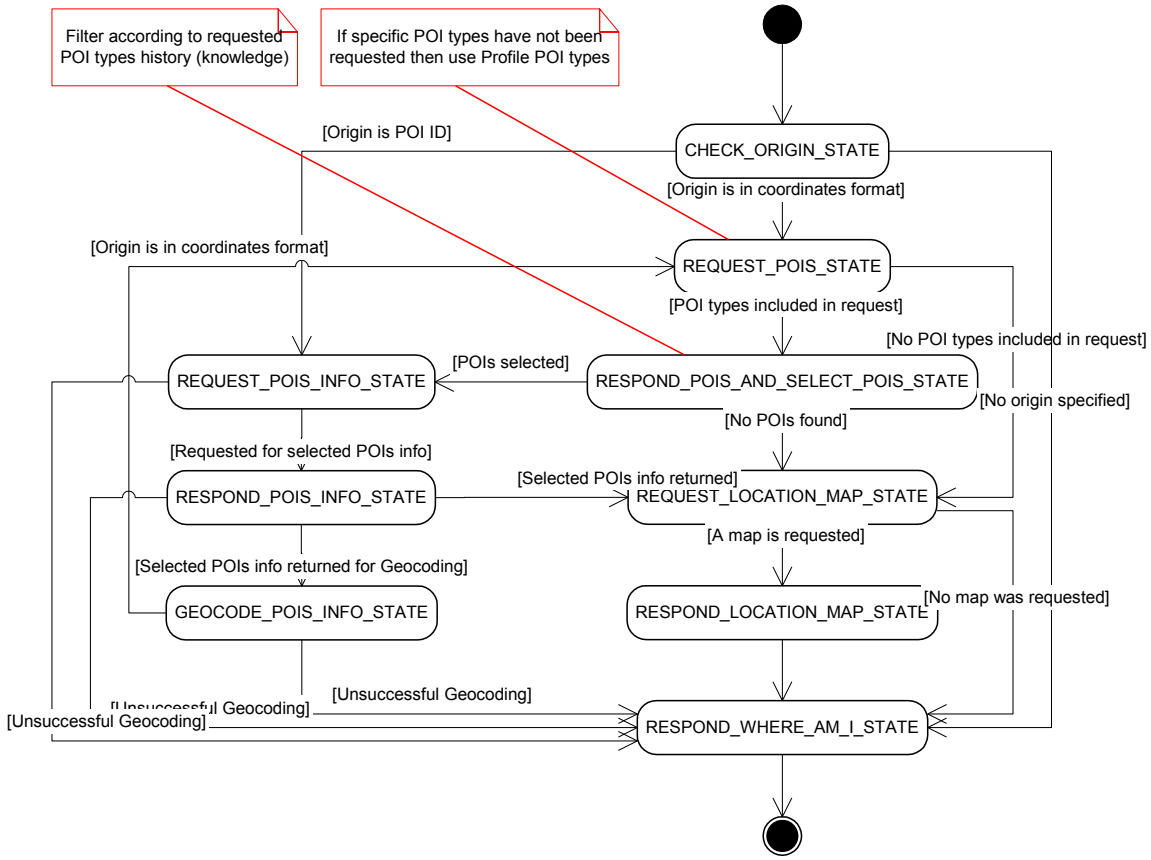

Figure 10: The *WhereAmI* behaviour detailed design

For illustration purposes, in Figure 11, we present the *PersonalAssistant* agent class, where the 5th step of the JADE implementation phase is demonstrated. The reader can see that the *SocialTypeBehaviour* takes as parameters the type of the agent, the type(s) to who he wants to introduce himself and the type(s) that he wants to add to his acquaintances list. This behaviour is used as is by all social agents just by changing the agent types parameters.

The *PersonalAssistantBehaviour* is presented in Figure 12. The reader can see that it is simply one behaviour that adds the *ServeUser* and *ReceiveNewEvents* roles/behaviours and this is a consequence of the bottom-up development process that is proposed by the Gaia2JADE process (i.e. the *ServeUserBehaviour* and *ReceiveNewEventsBehaviour* are already implemented when the overall *PersonalAssistantBehaviour*'s time for implementation has come) and allows for modularity in the system's development process.

```
public class PersonalAssistantAgent extends Agent {
  //declare agent level data classes
  protected Acquaintances contacts = null;
  protected void setup(){
    //get arguments - user profile
    Object [] args = this.getArguments();
    UserProfile userProfile = (UserProfile)args[0];
    //initialize agent data classes
    contacts = new Acquaintances();
    //activate SocialType and PersonalAssistant behaviours
    addBehaviour(new SocialTypeBehaviour(this,contacts,
      //find agent types: TravelGuide and add them to contacts
      new String[]{Acquaintances.TRAVEL_GUIDE},
      //Introduce agent as of type PersonalAssistant to agent types: EventsHandler
      new String[]{Acquaintances.EVENTS_HANDLER}, Acquaintances.PERSONAL_ASSISTANT));
    addBehaviour(new PersonalAssistantBehaviour(this, contacts, userProfile));
  }
}
```

Figure 11: The *PersonalAssistant* agent type class

```
public class PersonalAssistantBehaviour extends SimpleBehaviour {
 public PersonalAssistantBehaviour(Agent ag, Acquaintances contacts, UserProfile
userProfile){
  //activate ServeUser and ReceiveNewEvents sub-behaviours
  addBehaviour(new ServeUserBehaviour (this.myAgent(), contacts, userProfile));
  addBehaviour(new ReceiveNewEventsBehaviour(this.myAgent(), contacts, userProfile));
 }
}
```

Figure 12: The *PersonalAssistant* role/behaviour

The adequate system functionality was tested by requiring the agents to perform the services described in the Gaia services model.

6. Discussion and Future Work

This paper presented the Gaia2JADE process for implementing Gaia models using the JADE framework. A preliminary version of this work was presented in (Moraitis et al. 2003). This process was used in order to implement a real world application, the Image system (see Moraitis et al., 2003b). The Gaia2JADE process is based on the technical issues that were pointed out during the development phase of our system.

Similar works like PASSI (Cossentino et al, 2003) and INGENIAS (Gomez-Sanz and Pavon, 2003) offer tools that allow for MAS design and implementation. The added value of our work is that it builds on the combination of a widely accepted and well-known modeling methodology like Gaia and on JADE, a widely recognized MAS implementation platform, in the MAS community.

Here we must note that the aim of this paper is not to promote the use of Gaia methodology against to other existing methodologies although we consider that it captures all the necessary elements for a MAS development and it is easy to understand and apply. Our aim is to show how one who decided for his own reasons, to use Gaia for the analysis and design phases, can use JADE for the implementation phase by following the Gaia2JADE process.

The use of SPEM for the modeling of this process allowed us to present it as the next phase of the one provided by Garro et al (2004) for the description of Gaia modeling process, thus providing a complete process for MAS analysis, design and development that is easy to apply for a development team that is familiar with Gaia and JADE. Moreover, the Gai2JADE process can also be used for developing a MAS that is part of a larger system. In our case study the MAS was developed in parallel with other software modules (e.g. GIS), whose development teams used classical object oriented development processes like the rational unified process (Kruchten, 2003). These processes are iterative and our ability to return even to the requirements phase from the JADE implementation phase (see Figure 1) allowed for a smooth cooperation between the development teams.

As future work, we plan to create a modeling tool that would allow the automatic generation of JADE classes after analysis and design using Gaia.

Acknowledgements

We gratefully acknowledge the Information Society Technologies (IST) Programme and specifically the specific targeted research project (STRP) “Intelligent Mobility AGents, Advanced Positioning and Mapping Technologies, INtEgrated Interoperable MulTimodal location based services” (IM@GINE IT, IST-2003-508008) project for contributing in the funding of our work.

References

Bellifemine, F., Caire, G., Trucco, T., Rimassa, G.: Jade Programmer’s Guide. JADE 3.1 http://sharon.cselt.it/projects/jade/, 2003

Bresciani, P., Giorgini, P., Giunchiglia, F., Mylopoulos, J. and Perini, A.: TROPOS: An Agent-Oriented Software Development Methodology. Journal of Autonomous Agents and Multi-Agent Systems (JAAMAS), Vol. 8, No. 3, May 2004

Collis, J. and Ndumu, D.: Zeus Methodology Documentation Part I: The Role Modelling Guide. Intelligent Systems Research Group. BT labs, British Telecommunications, 1999

Cossentino, M., Burrafato, P., Lombardo, S., Sabatucci, L.: Introducing Pattern Reuse in the Design of Multi-Agent Systems. In R. Kowalszyk, et al. (eds), "Agent Technologies, Infrastructures, Tools, and Applications for e-Services", LNAI. 2592, Springer-Verlag, 2003

DeLoach S. and Wood, M.: Developing Multiagent Systems with agentTool. In: Castelfranchi, C., Lesperance Y. (Eds.): Intelligent Agents VII. Agent Theories Architectures and Languages, 7th International Workshop (ATAL 2000, Boston, MA, USA, July 7-9, 2000),. Lecture Notes in Computer Science 1986, Springer Verlag, 2001

Emorphia Ltd. FIPA-OS: A component-based toolkit enabling rapid development of FIPA compliant agents. http://fipa-os.sourceforge.net/, 2003

FIPA TC Agent Management: FIPA Agent Management Specification. Foundation for intelligent Physical Agents (FIPA - SC00023J), http://www.fipa.org, 2002

FIPA TC Communication: FIPA ACL Message Structure Specification. Foundation for intelligent Physical Agents (FIPA - SC00061G), http://www.fipa.org, 2002

Garro, A., Turci, P., Huget, M.P.: Meta-Model Sources: Gaia. FIPA Methodology Technical Committee, Foundation for Intelligent Physical Agents, http://www.fipa.org/ , 2004

Gomez-Sanz, J. and Pavon, J.: Agent Oriented Software Engineering with INGENIAS. In V. Maik, J. Müller, M. Pchouek (Eds.), Multi-Agent Systems and Applications III (p3rd International Central and Eastern European Conference on Multi-Agent Systems, CEEMAS 2003), LNCS 2691, Springer Verlag, pp. 394–403, 2003

Kruchten, P.. The Rational Unified Process: An Introduction, Third Edition. Addison-Wesley Pub Co, 3rd edition, 2003

Moraitis, P., Petraki, E., Spanoudakis, N.: Engineering JADE Agents with the Gaia Methodology. In R. Kowalszyk, et al. (eds), "Agent Technologies, Infrastructures, Tools, and Applications for e-Services", LNAI 2592, Springer-Verlag, pp. 77-91, 2003a

Moraitis, P., Petraki, E. and Spanoudakis, N.: Providing Advanced, Personal-ised Infomobility Services Using Agent Technology. In: Twenty-third SGAI International Conference on Innovative Techniques and Applications of Artificial Intelligence (AI2003), Peterhouse College, Cambridge, UK, December, 2003b

Noy, N. F., Sintek, M., Decker, S., Crubezy, M., Fergerson, R. W. & Musen, M. A.: Creating Semantic Web Contents with Protégé-2000. IEEE Intelligent Systems 16(2):60-71, http://protege.stanford.edu/, 2001

Object Management Group: Software Process Engineering Metamodel Specification. OMG, 2002, http://www.omg.org

Odell, J., Parunak, H. and Bauer, B. Extending UML for Agents. In Proc. of the Agent-Oriented Information Systems Workshop at the AAAI00, pp 3-17, 2000

Sommerville, I.. Software Engineering. Addison-Wesley Pub Co, 6th edition, 2000

Wood, M.F. and DeLoach, S.A.: An Overview of the Multiagent Systems Engineering Methodology. In the 1st Int. Workshop on Agent Oriented Software Engineering (AOSE-2000). Limerick, Ireland, 2000

Zambonelli, F., Jennings, N. R. and Wooldridge, M.: Developing multiagent systems: the Gaia Methodology. ACM Transactions on Software Engineering and Methodology 12 (3), pp. 317-370, 2003