Combining Gaia and JADE for Multi-Agent Systems Development

Pavlos Moraitis[1,2]

[1]Dept. of Computer Science
University of Cyprus
75 Kallipoleos Str., Nicosia, Cyprus
email: moraitis@cs.ucy.ac.cy

Nikolaos I. Spanoudakis[2,*]

[2]European Projects Dept.
Singular Software S.A.
26th October 43, 54626, Thessaloniki, Greece
email: nspan@si.gr

Abstract

In this paper we present an enhanced version of a roadmap we have previously proposed, concerning how one can implement JADE agents using the Gaia methodology for analysis and design purposes. This effort is based on the experience we have acquired by using this roadmap for implementing a real word multi-agent system conceived for providing e-services to mobile users. Thus, our aim here is to share this experience with future MAS developers, who would like to follow this refined version of our roadmap, taking into account several technical issues that emerged during the implementation phase, in order 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 Gaia (Wooldridge et al, 2000), AUML (Odell et al., 2000), MaSE (Wood and DeLoach, 2000) have been proposed but they cover only the requirements, analysis and design phases 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) for allowing analysis and design methodologies to be implemented using JADE (Bellifemine et al, 2002) or the FIPA-OS (Emorphia Ltd, 2003) open source frameworks. Unfortunately, until today no real world applications have used these roadmaps and tools in order to evaluate them.

In this paper we discuss our experience using the roadmap proposed in (Moraitis et al., 2003a) in order to engineer a real-world multi-agent system (MAS) that was analyzed and designed using the Gaia methodology and implemented with the JADE framework. The weak and strong points of Gaia when it comes to

implementation using JADE were recognized and we can now refine the previously proposed roadmap with enough detail so as to enable future MAS developers to easily model and implement their systems.

This paper is organized in the following way. In section 2 we provide our system requirements and the resulting Gaia model. In section 3 we discuss on the added value of the roadmap for implementing Gaia models using JADE, we provide some examples and propose a new type of modeling needed before implementation. Finally, section 4 includes discussion and future work.

2 Analysis and Design Phases

The Gaia methodology was considered as quite easy to learn and use in order to analyze and design a multi-agent system. It 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 overall project management proceeded using the iterative principles of the Rational Unified Process (Kruchten, 2003) that is an iterative software development process and demanded that our plans changed some times during project development.

2.1 The System Requirements

In order for the reader to better understand our experience on how GAIA and JADE were combined to conceive and implement a multi-agent system (MAS) we will present a limited version of the system that was implemented in the framework of the IST IMAGE project. This version is extended with regard to the one presented in (Moraitis et al, 2003a) so that problems related to the complexity of our task can be presented adequately. We will show how this system was analyzed, designed and implemented. The aim of this system was to provide e-services for mobile users. For this system we had the following requirements:

- 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).

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

2.2 The Analysis phase

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 1. This is an enhanced version of the similar one presented in (Moraitis et al, 2003a). 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).

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 structure.
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 structure, read acquaintances data structure.
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. [DecidePOITypes. [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 structure.
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, RequestRoutes, RespondRoutes, RequestMap, RespondMap, RequestNearbyPOIs, RespondNearbyPOIs, RequestPOIsInfo, RespondPOIsInfo
Permissions: read GIS.
Responsibilities:
   Liveness:
     TRAVELGUIDE = RegisterDF. ([FindRoutes] || [ProximitySearch] || [CreateMap] || [GetPOIInfo]) $^\omega$
     FINDROUTES = RequestRoutes. RespondRoutes
     PROXIMITYSEARCH = RequestNearbyPOIs. RespondNearbyPOIs
     CREATEMAP = RequestMap. RespondMap
     GETPOISINFO = RequestPOIsInfo. RespondPOIsInfo
   Safety: A successful connection with the GIS is established.

Table 1: The Gaia Roles Model

The Gaia interaction model denotes which action returns from a request along with the roles that can initiate a request and the corresponding responders. Figure 1 holds the necessary information for our model. However, we considered that the Gaia interaction model wasn't appropriate to represent complex coordination protocols. We overcame this difficulty by creating scenarios using AUML sequence diagrams in order to

write down complex liveness formulas (like the WhereAmI of the PersonalAssistant role).
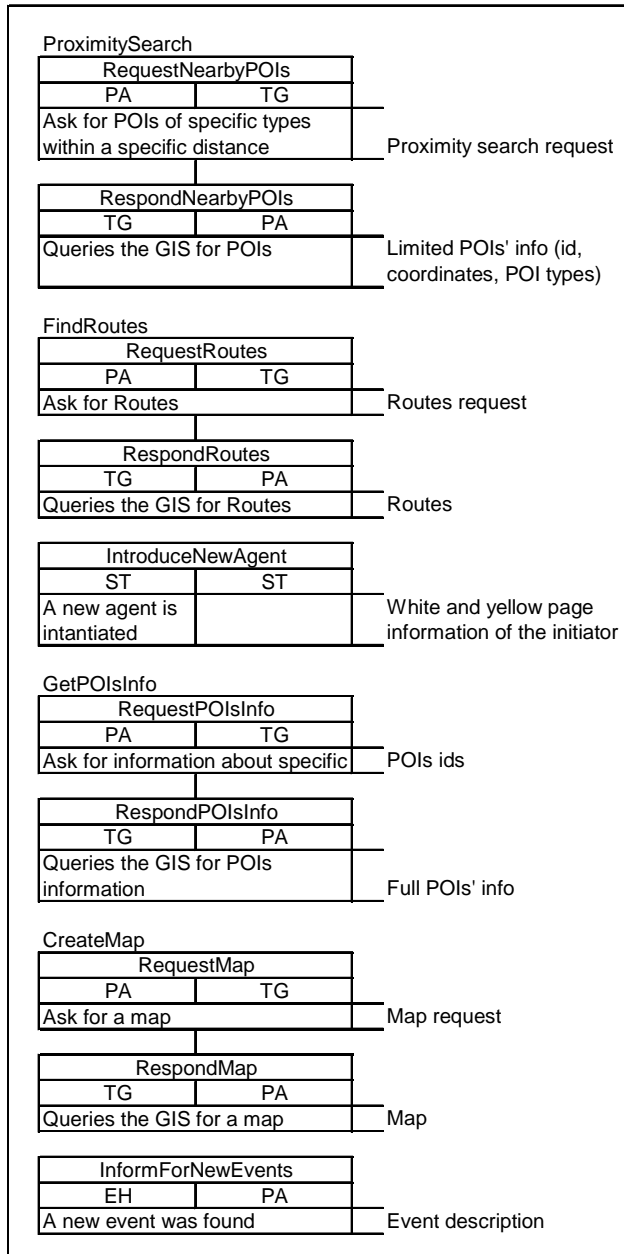
ProximitySearch

RequestNearbyPOIs	
PA	TG
Ask for POIs of specific types within a specific distance	Proximity search request

RespondNearbyPOIs	
TG	PA
Queries the GIS for POIs	Limited POIs' info (id, coordinates, POI types)

FindRoutes

RequestRoutes	
PA	TG
Ask for Routes	Routes request

RespondRoutes	
TG	PA
Queries the GIS for Routes	Routes

IntroduceNewAgent	
ST	ST
A new agent is intantiated	White and yellow page information of the initiator

GetPOIsInfo

RequestPOIsInfo	
PA	TG
Ask for information about specific	POIs ids

RespondPOIsInfo	
TG	PA
Queries the GIS for POIs information	Full POIs' info

CreateMap

RequestMap	
PA	TG
Ask for a map	Map request

RespondMap	
TG	PA
Queries the GIS for a map	Map

InformForNewEvents	
EH	PA
A new event was found	Event description

Figure 1: Gaia Interactions Model

2.3 The Design Phase

During this phase the Agent model was achieved, along with the services and acquaintance models. The Agent model is presented graphically in Figure 2.

The services model for our system is presented in Table 2. Finally we defined the acquaintances model (the reader could refer to Moraitis et al., 2003a in order to see the graphic representation). There, the PersonalAssistant agent was shown to interact with all agent types, while the others interacted only with the PersonalAssistant agent.

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.
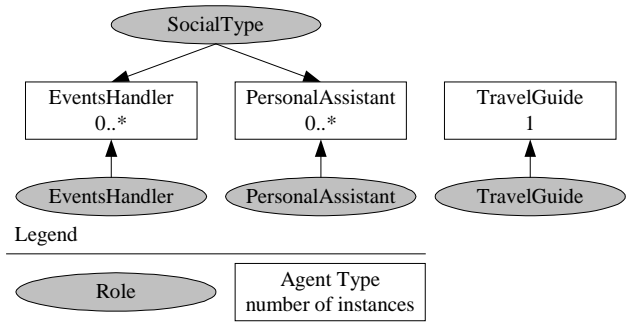

Legend

Figure 2: Gaia Agent Model

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 2: Gaia Services Model

3 Detailed Design and Implementation Phases

In this section we present an enhanced version of the roadmap proposed in (Moraitis et al 2003a) in order to design and implement the Gaia models using the JADE framework. The novelty of this version concerns mainly steps 2 and 4. The reasons of this update will be explained later within this section. Therefore, the steps now are:

1. Define all the ACL messages by using the Gaia protocols and interactions models.

2. Define the needed data structures and software modules that are going to be used by the agents by using the Gaia roles and agent models. Create the activities refinement table (see Table 3).

3. Decide on the implementation of the safety conditions of each role.

4. Define the JADE behaviours. Start by implementing those of the lowest levels, using the various Behaviour class antecedents provided by JADE. The Gaia model that is useful in this phase is the roles model. Behaviours that are activated on the receipt of a specific message type must either add a message receiver behaviour (if they are complex-FSM behaviours), or receive a message (with the appropriate message filtering template) at the start of their action. Gaia activities that execute one after another (sequence of actions that require no interaction between agents) with no interleaving protocols can be aggregated in one activity (behaviour method or action). However, for reusability, clarity and programming tasks allocation reasons, we believe that a developer could opt to implement them as separate methods (or actions in an FSM like behaviour). Use state diagrams in order to model FSM-like behaviours and recognize the common data structures used by the lower level behaviours. Initialize those data structures at the upper level behaviour and pass them as parameters to lower level behaviours.

5. Keep in mind that Gaia roles translated to JADE behaviours are reusable pieces of code. In our system, the same code of the behaviours GetAcquainted and MeetSomeone will be used both for the personal assistant and events handler agents.

6. At the setup method of the Agent class invoke all methods (Gaia activities) that are executed once at the beginning of the top behaviour (e.g. RegisterDF). Initialize all agent data structures. Add all behaviours of the lower level in the agent scheduler.

The overall development process is, thus, top-down in the analysis and design phase (Gaia) and bottom-up in the implementation phase, according to the most successful software engineering practices.

During the detailed design (steps 1 and 2) we introduced the activities refinement table in order to facilitate the Gaia Roles Model activities design and implementation. In this table we wrote down the necessary data structures and algorithms. As an example, the Decide-POITypes activity of the PersonalAssistant role refinement is presented in Table 3. In fact, the goal here is to facilitate the link of such data to a JADE behaviour. Thus, when a behaviour is created the developer can use this table in order to write the constructor of the behaviour and define its functionality by implementing the algorithm within its *action* method.

We used UML class diagrams in order to model the data structures that would be used by each role's permissions field and defined interfaces for external services usage (GIS, database, etc) and the ontology for our system. Finally we defined the ACL messages that would be used by each protocol (FIPA performatives, protocols, content).

Role	Activities	Data Structures		Description
		Read	Update	
PA	Decide-POITypes	user profile user request	-	`if UserRequest.POITypes.length>0` **Then** `RequestNearby-POIs(UserRequest.POITypes)` **else if** `UserProfile.POITypes.length>0` **then** `RequestNearby-POIs(UserProfile.POITypes)` **else** `CreateMap`

Table 3: The Gaia roles' activities refinement table

Then we defined the ways to safeguard the roles' safety conditions (step 3 of the roadmap). For the TravelGuide role we decided that whenever a connection with the GIS fails the relevant protocols will be replying with FAILURE FIPA performative (FIPA, 2000) to the PersonalAssistant role and the system administrator will be informed about it with a dialog. For the EventsHandler role the same dialog is used in order to inform the administrator about connectivity problems with the events database.

Step 4 of the roadmap proved to be the most cumbersome one, since most of the implementation takes place in this step. The implementation phase enabled us to refine the roadmap steps (usually through problems that came up) as presented above.

One of the technical issues that emerged is that, usually, many roles/behaviours need to access the same data structure. For example, in our case, the PersonalAssistant role behaviour reads the Acquaintances structure while the SocialType role behaviour updates it with new acquaintances. In these cases, data structures must be instantiated in an upper level. In our example, this structure has been declared and instantiated at the agent's constructor and then passed as a parameter to each of the two behaviours. Thus, they can both access it. Here we can remark that we have no synchronization problems regarding access to the same data structure by different behaviours, since only one behaviour is executed at any given time by the JADE scheduler.

Secondly, the Gaia roles model, allowed for complex behaviours to be modeled, but the transformation to JADE Finite State Machine behaviour (FSMBehaviour) instances, wasn't obvious. Thus we had to create state diagrams for these FSM behaviours (like the PlanATrip and WhereAmI behaviours of PersonalAssistant). The state diagram for the WhereAmI behaviour of the PersonalAssistant role is presented in Figure 3. These diagrams provide another important information that is crucial for easy development. By observing the information exchanged between the different simple behaviours within the FSM behaviour, we recognize the data structures that must be defined in the FSM behaviour level so that more than one of its children behaviours can access them.

For example in the WhereAmI behaviour the necessary data structures are 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. Again these data structures must be initialised at the constructor of the FSM behaviour. It is also worth observing how many sub-behaviours of these two high level behaviours are common, for example the Request/RespondMap behaviours that are used in order to implement the CreateMap protocol by both the PlanATrip and WhereAmI roles/behaviours.
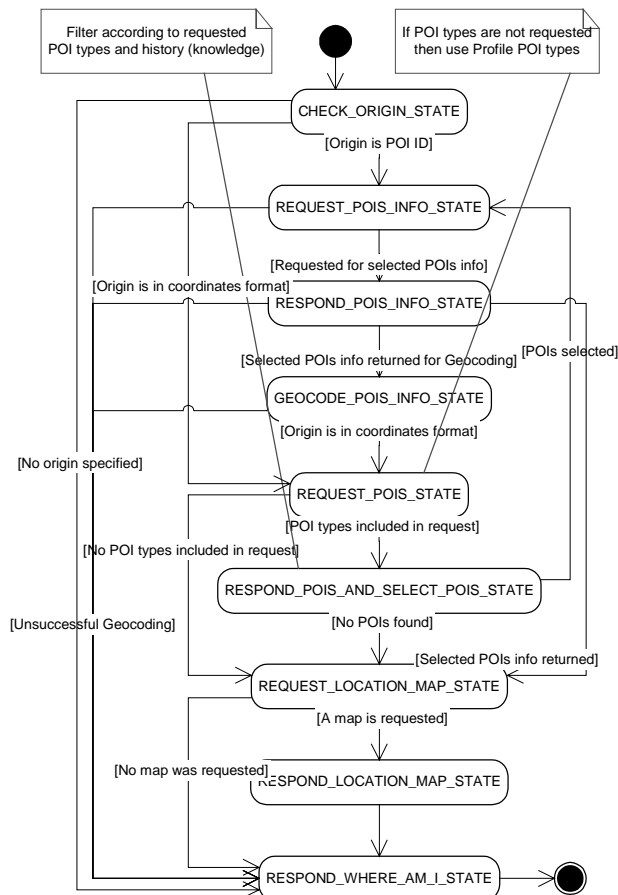

Figure 3: The WhereAmI behaviour detailed design

We also propose the FSMChildBehaviour class (see Figure 4) that proved very useful, since we were able to automate a lot of repeating code in simple behaviours within FSMBehaviours. 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 FSMBehaviours. 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 4: The FSMChildBehaviour class

For illustration purposes, in Figure 5, we present the PersonalAssistant agent class, where both steps 5 and 6 of the roadmap are demonstrated. The reader can see the SocialTypeBehaviour takes as parameters the type of the agent, the one to who he wants to introduce himself and the one that he wants to add to his acquaintances structures. Thus, this behaviour is used as is by all social agents (we could say as a component).

```
public class PersonalAssistantAgent extends Agent {
  //declare agent level data structures
  protected Acquaintances contacts = null;
  protected void setup(){
    //get arguments - user profile
    Object [] args = this.getArguments();
    UserProfile userProfile = (UserProfile)args[0];
    //initialize agent data structures
    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 5: 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 6: The PersonalAssistant role/behaviour

The PersonalAssistantBehaviour is presented in Figure 6. The reader can see that it is simply one behaviour that adds the ServeUser and Receive-NewEvents roles/behaviours. He might wonder why weren't they invoked directly from the agent class.

This is a consequence of the bottom-up development process that is proposed by the roadmap (i.e. the ServeUserBehaviour and ReceiveNewEventsBehaviour are already implemented when the overall PersonalAssistantBehaviour's time for implementation has come).

4   Discussion and Future Work

This paper presented en enhanced version of the roadmap proposed in (Moraitis et al 2003a). This version is based on the technical issues that were pointed out during the development phase and we propose it in order to further facilitate the implementation of Gaia models using the JADE framework.

In general, the process of developing our system can be considered as an agile process for multi-agent systems development (Larman, 2003). It allowed for modularity during design and implementation phases and for incremental, iterative development. This is also supported by the fact that we successfully implemented a complex system with 7 agent types that used about 80 behaviours and about 900Kbytes of source code in one year. For the overall system design we used the RUP methodology.

Moreover, the top-down design followed by the bottom-up implementation seemed a very good practice to us. We actually used behaviours as components for building agents. The latter provided services, thus becoming system level components. Sycara et al (2003) discuss the large MAS modeling issue and the problems related to introducing agents in existing communities, where the new agents can use already provided services in order to provide new services. Another comment that is appropriate here is that the services model that comes at the Gaia design phase is useful for checking the system requirements and whether those are satisfied by the modeled system.

The MAS that we developed was a module of a larger system. It interfaced with a legacy GIS system and a web-based user interface (UI). The interfaces with the GIS system were implemented as web services while the interface with the UI was the exchange of XML documents through TCP/IP sockets. The reader can refer to (Moraitis et al, 2003b) for details on why we selected Gaia and JADE for our MAS development.

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

References

[Bellifemine et al., 2002] Bellifemine, F., Caire, G., Trucco, T., Rimassa, G.: Jade Programmer's Guide. JADE 2.5 http://sharon.cselt.it/projects/jade/, 2002

[Bresciani et al., 2003] Bresciani, P., Giorgini, P., Giunchiglia, F., Mylopoulos, J. and Perini, A.: TROPOS: An Agent-Oriented Software Development Methodology. Accepted in JAAMAS, 2003

[Cossentino et al., 2003] 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

[FIPA, 2000] FIPA specification XC00061E: FIPA ACL Message Structure Specification. http://www.fipa.org, 2000

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

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

[Larman, 2003] Larman, C.. Agile and Iterative Development: A Manager's Guide. Addison-Wesley Pub Co; 1st edition, 2003

[Moraitis et al., 2003a] 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, 2003, pp. 77-91

[Moraitis et al., 2003b] Moraitis, P., Petraki, E. and Spanoudakis, N.: "Providing Advanced, Personalised 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, 2003

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

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

[Sycara et al., 2003] Sycara, K., Giampapa, J.A., Langley, B.K. and Paolucci, M.: The RETSINA MAS, a Case Study. Software Engineering for Large-Scale Multi-Agent Systems: Research Issues and Practical Applications, Garcia, A., Lucena, C., Zambonelli, F., Omici, A. and Castro, J. (eds.), LNCS 2603, Springer-Verlag, 2003, pp. 232-250

[Wood and DeLoach, 2000] Wood, M.F. and DeLoach, S.A.: An Overview of the Multiagent Systems Engineering Methodology. AOSE-2000, The 1st Int. Workshop on AOSE. Limerick, Ireland, 2000

[Wooldridge et al., 2000] Wooldridge, M., Jennings, N.R., Kinny, D.: The Gaia Methodology for Agent-Oriented Analysis and Design. JAAMAS Vol. 3. No. 3, 2000, pp. 285-312