Designing multi-agent systems: a framework and application

Soo Yong Park\textsuperscript{a}, Vijayan Sugumaran\textsuperscript{b,}\textsuperscript{*}

\textsuperscript{a}Department of Computer Science, Sogang University, Seoul 121-742, South Korea
\textsuperscript{b}Department of Decision and Information Sciences, School of Business Administration, Oakland University, Rochester, MI 48309, USA

Abstract

The era of distributed software environments is emerging and research on multi-agent systems (MAS), which tries to solve complex problems using entities called agents, is on the rise. This paper proposes an architecture-centric method for developing MAS that supports the important phases of systematic software development. In particular, this approach is geared towards supporting system properties specifically focused on agent coordination and autonomy. A goal-based approach is utilized for the problem domain analysis, and individual agents are mapped to the system’s refined goals. Further, architectural styles and patterns are applied to generate the overall design of MAS. UML (Unified Modeling Language) and ADL (Architecture Description Language) are used for modeling and formalizing the MAS architecture. The proposed architecture is applied to ITS (Intelligent Transport Systems) domain and a proof-of-concept prototype has been developed to demonstrate our approach.

\textsuperscript{q}2004 Elsevier Ltd. All rights reserved.

Keywords: Intelligent agent; Multi-agent systems; MAS architecture; Unified Modeling Language (UML); Architecture Description Language (ADL); Agent design

1. Introduction

Most of the complex real world problems are solved using distributed environments (Gomez-Sanz, Pavon, & Garijo, 2002; Jennings, 2001). Currently, there is tremendous pressure to design and develop systems in a short period of time. For example, a growing number of e-commerce applications are being deployed on a daily basis and this situation is only going to get worse with the incredible growth of the Internet and web-based applications. One approach commonly used to accelerate distributed systems development is to reuse previously developed components with similar functionalities (Davis, Luo, & Liu, 2003). A large distributed system could be developed through identifying reusable software components, customizing them to meet the new requirements and integrating them with newly developed software (Maturana et al., 2004). To address the issues of complex systems development in distributed environments, research on multi-agent systems (MAS) and their application is on the rise (Dzeng & Lin, 2004; Lee & Park, 2003; Lee & Tsai, 2003; Pontelli & Son, 2003). A MAS tries to solve complex problems with entities called agents, using their collaborative and autonomous properties (Liau, 2003). However, the effort in designing multi-agent systems suffers from lack of systematic approach that is grounded in software development methodologies.

In order to develop MAS in a systematic way, we need to analyze the system in terms of its ultimate goals and design the system both in the abstract as well as concrete by mapping the goals and the subgoals to software agents (Park, Kim, & Lee, 2000; Park, Kim, & Park, 2000). The implementation of a MAS is only as good as its design; hence, it is critical that correct design decisions are made (Xu & Shatz, 2003). A well thought out architecture for the system provides an effective blueprint for the system and leads to the right implementation with little error. Thus, the architecture of the system is its backbone and offers guidelines for its development. Every system has its own framework and the right establishment of this framework can lead to the right system, and even right analysis and design for extending the system in future (Symeonidis, Kehagias, & Mitkas, 2003). Furthermore, this is an efficient way to improve the system’s reliability and performance.

\textsuperscript{*} Corresponding author. Tel.: +1 248 3702831; fax: +1 248 3704275.
E-mail address: sugumara@oakland.edu (V. Sugumaran).
(Yim, Ahn, Kim, & Park, 2004). Consequently, it is imperative that a systematic analysis and architecture development process be undertaken and results formalized in order to generate the right system.

This paper proposes an architecture-based method for the systematic development of MAS. A goal-based approach is used for problem domain analysis, and agents are mapped to the refined goals of the system. In order to support the coordination and autonomy needs of agents, which are considered as main properties of MAS, architectural styles and patterns are utilized in representing the architecture. UML (Unified Modeling Language) (Fowler & Scott, 2000) and ADL (Architecture Description Language) (Clements, 1996) are used in modeling and formalizing the architecture. The proposed method has been applied to ITS (Intelligent Transport Systems) domain and a proof-of-concept prototype has been implemented to provide face-validity of the approach.

The remainder of the paper is organized as follows. Section 2 briefly discusses MAS characteristics and agent architecture methods. Section 3 describes the architecture development process for MAS in view of coordination and autonomy, and provides a formal representation of the architecture using UML and ADL. Section 4 discusses the application of the proposed architecture to the ITS domain, and the implementation of a proof-of-concept prototype. Finally, Section 5 provides summary and future research.

2. Related work

A multi-agent system (MAS) is defined as a loosely coupled network of problem solvers that work together to solve problems that are beyond the individual capabilities or knowledge of each problem solver (Durfee & Lesser, 1989). The increasing interest in MAS research is due to significant advantages inherent in such systems, including their ability to solve problems that may be too large for a centralized single agent, provide enhanced speed and reliability and tolerate uncertain data and knowledge. Some of the key research issues related to problem-solving activities of agents in a MAS are in the areas of coordination, negotiation and communication (Nwana, 1996). Coordination is the process by which an agent allocates tasks to other agents and synthesizes the results from these agents to generate an overall output. Negotiation is the process by which agents solve their conflicts and reach a compromise. For coordination and negotiation, they need to communicate with one another and hence the system should provide a general communication mechanism.

In order to support MAS requirements such as coordination, negotiation, and communication, several architectures have been reported in the literature. The Gaia methodology (Wooldridge, Jennings, & Kinny, 2000; Zambonelli, Jennings, & Woolridge, 2003) supports the development of agents with formal models of knowledge and environment, role hierarchies, and representation of social structures and relationships. However, it requires that inter-agent relationships and agent abilities be static at run time and hence may be of less value in the unpredictable domain of Internet applications. RETSINA (Reusable Task Structure Based Intelligent Network Agents) (Sycara, Decker, Pannu, Williamson, & Zeng, 1996; Sycara, Paolucci, van Velsen, & Giampapa, 2003) is a general-purpose modeling framework which proposes goal, role, context and attitude as first class objects for modeling multi-agent systems in an open world. The Multiagent Systems Engineering (MaSE) methodology leads the designer from the initial system specification to the implemented agent system (DeLoach, Wood, & Sparkman, 2001). In the analysis phase, MaSE uses goal hierarchies and role models, whereas the design phase creates agent-class, communication and deployment diagrams. A UML (Bauer, Muller, & Odell, 2001) is an extension to UML to represent various aspects of agents by introducing new types of diagrams including agent class diagrams and protocol diagrams. Although these methodologies support cooperating agents, they do not support teams of agents very well.

The Tropos methodology (Bresciani, Perini, Giorgini, Giunchiglia, & Mylopoulos, 2004; Giunchiglia, Mylopoulos, & Perini, 2002) has a strong focus on requirements analysis and consists of different phases, namely, early and late requirements, architectural design, detailed design and implementation. However, one criticism of this approach is that it does not provide strong support for protocols and modeling the dynamic aspects of the system (Dam & Winikoff, 2003). The Prometheus approach (Padgham & Winikoff, 2002) supports software engineering activities and detailed processes for developing agent applications. It consists of the following three phases: (a) system specification, (b) architectural design, and (c) detailed design. While this method is targeted for people who do not have a background in agents, it is weak in terms of support for concurrency. Finally, DECAF (Distributed, Environment-Centered Agent Framework) (Graham, Decker, & Mersic, 2003) is an agent toolkit to design, develop and execute agents to achieve solutions in complex software systems. Since DECAF creates solutions to high-level tasks via decomposition using a predefined library of task schemas, it has the shortcoming of not being able to create new task decompositions.

The methodologies and architectures discussed above primarily take an implementation point of view and focus heavily on developing a system rapidly. However, due to their emphasis on the implementation viewpoint, these architectures fall short in non-functional capability considerations of systems development. For example, the implementation of a facilitator or a coordinator agent within a system may lead to bottleneck problems if sufficient attention is not paid to internal process control structures. In MAS development, not only functional aspects but also quality attributes, such as reliability, adaptability, etc. should be considered as main issues.
This paper presents a method for MAS development, supporting both functional—which provides services to solve complex problems in distributed environments—and non-functional properties—which provide the capability to reuse, easy to extend, adapt and process uncertain data, etc. In our approach, special attention is devoted to capturing agents’ coordination and autonomy requirements and we adopt a phased approach. Each phase of our architecture development process is represented and formalized using architectural styles, patterns, UML and ADL (Architecture Description Language).

3. Framework for multi-agent systems development

In our proposed architecture-based method for MAS development, we follow three main phases as depicted in Fig. 1. We incorporate the general properties of MAS and suggest an essential architecture reflecting these properties. The first phase in our approach is called ‘Problem Analysis.’ The main focus of this phase is in gaining an understanding of what the system does in the abstract, which serves as a starting point for the architecture development process. The activities undertaken in this phase pertain to understanding the application domain, identifying goals and boundary of the system, and relating them to agent design. After gaining an understanding of the domain and the overall goals of the system from the problem analysis, we move on to the next phase in which agents are identified to satisfy the analyzed goals and their relationships. This phase is called ‘Agent Modeling’. In this phase, for each identified agent, its internal behavior and belief are modeled. The final phase is ‘MAS Architecting’, which focuses on the internal architecture of agents and setting up the federation of agents that collaborate with each other, while maintaining autonomy to a large extent. Agent coordination and communication is critical for the federation to function successfully. The remainder of this section describes the goal analysis and architecture development phases in detail.

3.1. Problem analysis

Problem analysis is essential for setting up system boundary and analyzing user requirements. It is generally agreed that the end users of a system often have difficulty expressing abstract requirements for the system and a goal oriented approach to requirements analysis is an effective mechanism to improve the elicitation process (Lamsweerde & Willemet, 1998). Thus, the concept of goal is central to understanding and designing systems, particularly, agent-based systems. There is an increasing trend towards designing agent-oriented software utilizing a goal-directed analysis process.

A goal is defined as ‘a non-operational objective to be achieved by the composite system (Rolland, Souveyet, & Ben Achour, 1998).’ Goal-directed analysis mainly involves identifying higher-level goals, and for each of these high level goals generating subgoals and also defining the relationships between them. Fig. 2 illustrates the general structure of a system’s goal hierarchy; the root of the hierarchy contains the root goal, which the MAS must achieve ultimately, and many subgoals, which are subdivided from the system root goal (Park, Kim, & Lee, 2000; Park, Kim, & Park, 2000). Goals are categorized into three types: system external goals, which are viewed from outside of the system; user goals, which are perceived by the users (Cockburn, 1997); and system internal goals viewed from inside the system. Goals can also be classified by their properties: achieve and maintain (Dardenne, Lamsweerde, & Fickas, 1993). The goal hierarchy diagram derived from goal-directed analysis is used to map agents to goals for developing an architecture for the MAS.

3.2. Agent modeling

Based on the analyzed goals, agents need to be identified and their relationships need to be modeled. For agent identification, relationship between goals and agents are
examined and we have developed a set of heuristic guidelines to create and map agents to goals. For example, ‘For each identified major goal, create a corresponding agent when necessary’ and ‘If the subgoals are mapped to agents, the upper level goal can then be mapped to a coordinator agent for supporting cooperation among agents.’

Since objects and agents can coexist in the real world, an Agent-Class diagram is created, which shows the relationships among agents and objects in the target problem domain. The notation we use in expressing different elements of the Agent-Class diagram is given below:

<table>
<thead>
<tr>
<th>Element</th>
<th>Relation</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Agent without mobility</td>
</tr>
<tr>
<td>M</td>
<td>Mobile agent</td>
</tr>
</tbody>
</table>

Fig. 3 shows an Agent-Class diagram from the ITS domain. There are four agent classes and two object classes. ‘User Info’ object class has been added to facilitate customization of the system for a specific user.

Next, for each identified agent, its internals are modeled. During this process, an agent’s beliefs that represent its knowledge, and its plan that enables the agent to act autonomously are modeled. Belief is considered to be data that an agent possesses. It contains information about the environment and the agent itself. This data is updated on a continual basis. This information is used to establish the agent’s knowledge base, or Ontology, which is built using Ontolingua (Gruber, 1993) or KIF (Knowledge Interchange Format) (UMBC, 2004).

The rules that determine what kind of information goes into different ontologies are listed below.

1. Ontology that can be established in the early stage
   - Information about Protocol or ACL (Agent Communication Language) for the agent’s communication can be established as Ontology in the early stage.

2. Ontology that needs to be updated continuously
   - Attributes and operations in a Class diagram, established as an ontology.
   - Messages among objects in Sequence diagram, established as ontology.

![Fig. 3. Agent-Class diagram.](image-url)
Agent’s information incorporated into the knowledge base, established as ontology.

Each ontology is represented in XML using the SHOE approach (Heflin, Hendler, & Luke, 1999). Each ontology has an ontology ID and contains various categories and relationships between them. After capturing the beliefs of agents, a Plan template is modeled which shows agent’s behavior to achieve a goal. It focuses on agent’s behavior changes as time goes on, and shows messages exchanged between agents. Based on the analysis of dynamic aspects of the system, each agent can determine its own behavior, referencing its goal and belief. Agent plan sequence diagram, which extends the established UML Sequence diagram, represents agent’s behaviors. The agent plan model uses the following primitives:

- **Mobile(Goal[Objective]):** the agent with its goal tends to migrate to other domains
- **Update(Belief[Type_of_ontology]):** agent updates its ontology according to this information
- **employ:** the agent uses classes
- **msg[message_context]:** message that is exchanged between agents.

**Fig. 4** shows a plan based on the Agent-Class diagram shown in **Fig. 3.**

### 3.3. MAS architecture development

The MAS architecture development process focuses on two major aspects: (a) agent organization architecture in view of agents’ coordination, and (b) agent’s internal architecture in view of agent’s autonomy. The former considers system’s reliability and flexibility in terms of agents’ interaction. The latter deals with each agent being able to respond to changes that occur in a dynamic environment based on its current state and its set of beliefs.

Each view has to be considered in the process of developing the overall architecture for MAS. This process involves clearly articulating the definition of essential elements, adjustment of architectural style, architecture representation, and architecture validation. The following two subsections provide further details on this process and its related artifacts for each view.

#### 3.3.1. Agent coordination structure

In view of agents’ coordination, the architecture shows the structure of system organization and the relationships among system elements (agents). **Fig. 5** shows the process for setting up the agent organization and formalizing its architecture.

From goal-directed analysis, we define system goals and map agents to subgoals. Each agent can be defined as 
\( (G, \{R\}) \) pair, where \( G \) stands for agent’s goal and \( \{R\} \) for agent’s roles. Agent’s role can be mapped to the user goal in goal hierarchy diagram. A user goal is ‘the goal of the primary actor trying to get some work done (Cockburn, 1997)’ and it gives an actor a meaningful service. A specific meaningful service (user goal) can be achieved by one or more agents, called an agent group (Gr), that works for the same role. In one group, agents interact with one another to complete their role. This interaction requires inter-agent communication and the coordinator agent (C) facilitates this communication. The coordinator agent maintains a list of agents that are participating in a group and enables the communication between them and manages their interaction. **Table 1** summarizes the different notations that we use in describing the system elements of MAS, i.e. agents, agent groups, and coordinators.

After defining the major elements for the system, an Agent System Diagram and an Agent Interaction Diagram are generated to represent the structure of the system organization. The Agent System Diagram represents the agents that are part of the system and their relationships, as well as the different agent groups. It also shows which
Coordinators manage what agents for a specified role. The general representation of this agent organization is depicted in Fig. 6.

While Agent System Diagram shows agent organization within the overall system, the Agent Interaction Diagram shows the coordination between agents for possible scenarios. This Interaction Diagram is represented using the Agent UML (AUML) (Odell, Van Dyke Parunak, & Bauer, 2000). Agent coordination within agent groups is expressed through messages passed between agents while performing their roles. In MAS, messages are exchanged between agents through a specific coordination protocol. Contract Net Protocol (Davis & Smith, 1983) is one of the protocols, which assigns agents to initiators or responders. When an agent needs to work with other agents, it sends a request to these agents within its group and these agents perform a role in the initiator/responder relationship. Derived from the standard protocol in FIPA (The Foundation of Intelligent Physical Agents) (FIPA, 2004), the Contract Net protocol is represented by Collaboration Diagram and Sequence Diagram.

The agent organization architecture needs to be validated by checking the system elements and their relationships. Typically, an ADL is used for validation purposes. While UML provides various graphical notations and models for capturing and representing the agent organization to help improve our understanding of the system, ADL supports a more formal representation of the system at the design level. This formal description of the system helps us validate the soundness of the system architecture by means of ADL supported tools. Thus, it is possible to validate different architectures during early stages of the system development as long as the architectures are expressed using an ADL. In our proposed method, we check the architecture for MAS using one of the ADL called Acme (Garlan, Monroe, & Wile, 1997; Garlan, Monroe, & Wile, 2000). Acme is the ADL that provides common interchange format for architecture design tools (Schmerl & Garlan, 2004).

Table 2 shows how Acme is used in describing a system. System elements, such as agents and coordinators are described as ‘component’, while the communication protocol among initiators and responders is represented as a ‘connector.’

3.3.2. Agent autonomy

After gaining an understanding of the system’s big picture through the agent organization architecture, the internal architecture of agents is then represented in the context of agent autonomy. Fig. 7 shows the process by which the internal architecture of agents is derived within

<table>
<thead>
<tr>
<th>Table 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Notation of system elements for MAS</td>
</tr>
<tr>
<td>(Agent: (A(G, {R})))</td>
</tr>
<tr>
<td>(G): Agent’s goal</td>
</tr>
<tr>
<td>({R}): A set of agent’s roles</td>
</tr>
<tr>
<td>(Agent Group: (\text{Gr}(R, {A})))</td>
</tr>
<tr>
<td>(R): The role that agent group achieves</td>
</tr>
<tr>
<td>({A}): A set of agents that participate in a role</td>
</tr>
<tr>
<td>(Coordinator: (C(R)))</td>
</tr>
<tr>
<td>(R): The role of agent group to which the coordinator manages</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architecture description in the view of coordination using Acme</td>
</tr>
<tr>
<td>System MAS = {</td>
</tr>
<tr>
<td>component Agent = {</td>
</tr>
<tr>
<td>Port AgentComm; }</td>
</tr>
<tr>
<td>component Coordinator = {</td>
</tr>
<tr>
<td>Port CoorComm; }</td>
</tr>
<tr>
<td>Connector Communication =</td>
</tr>
<tr>
<td>Roles { contractInitiator; contractResponder; }; }</td>
</tr>
<tr>
<td>Attachments</td>
</tr>
<tr>
<td>Coordinator.CoorComm to Communication.contractInitiator;</td>
</tr>
<tr>
<td>Agent.AgentComm to Communication.contractResponder;</td>
</tr>
<tr>
<td>}</td>
</tr>
</tbody>
</table>

Fig. 6. General representation of agent organization.

Fig. 7. Architecture process for MAS supporting agents’ autonomy.
a MAS. The internal architecture of agents should reflect the autonomous behavior of agents. ‘Autonomy’ refers to an agent not depending on the properties or the states of other components for its functionality (Shehory, 1998). That is, an agent has sole control over the activation of its services and may refuse to provide a particular service, or ask for a compensation for its services. The autonomy of agents play a big role in determining the internal architecture of agents and how they react to changes in its surrounding environment. As depicted in Fig. 7, the first step in creating the internal architecture of an agent is to define the internal modules for that agent. The structure of an agent consists of four parts: Goal, Belief, Plan, and Capability (Park, Kim, & Lee, 2000; Park, Kim, & Park, 2000). These are implemented as plan module, belief module, and capability module to provide inter-agent communication and application services. Fig. 8 shows how to map the structure of an agent to the elements of agent internal architecture.

In MAS environments, an agent tries to achieve its goal and the agent’s plan controls and executes other modules to complete its goal. Agent’s plan module is regarded as the driver, which facilitates the activation and termination of services. Agents recognize and react to current situations in accordance with the belief system that is contained in the belief module, and invokes appropriate capability modules to perform their functions.

Before performing a function, agent understands its current state from its goal and the corresponding information from the belief module. Based on the current state, the plan module chooses proper strategy and performs its function based on the strategy. With the architectural styles, various patterns can be applied to represent the plan module’s choice of proper capabilities. State pattern and Strategy pattern (Gamma, Helm, Johnson, & Vlissides, 1995) can be used for dynamically choosing the appropriate capability module to execute, and plan module’s state change can be expressed using State Diagram in UML. As we validate agent organization architecture using Acme, agent internal architecture is formalized by describing its major elements and their connections. Belief module and capability module are treated as ‘components,’ while the plan module is treated as the ‘connector.’

4. Application to intelligent transport systems (ITS)

The proposed approach has been applied to ITS domain and a Pre-trip Traveler Information Systems (PTIS) has been developed based on the proposed architecture. In this problem domain, the user gets information about flight schedule, weather, and subway schedule from the PTIS service site. Fig. 9 describes the problem domain. For example, the traveler can obtain not only the flight schedules, but also obtain weather reports, current subway utilization and schedule reports, and road and traffic condition reports on the day of travel at any time. This section highlights some of the salient features of our approach.

4.1. Problem analysis and goal hierarchy diagram

For the ITS domain, we start with the global analysis and define system goals (external and internal goals) and user goals. Fig. 10 shows the result of problem analysis phase and resulting the goal hierarchy diagram. From the goal analysis, we identify agents for this domain; subgoals are
mapped to agents, user goals are mapped to system’s roles. In this sample scenario, 6 agents are needed; UserInteraction agent, DB Wrapper agent, FlightInfo agent, RoadInfo agent, SubwayInfo agent, and WeatherInfo agent. They are derived from internal goals of the goal hierarchy diagram.

4.2. Agent organization architecture in view of agent coordination

After identifying appropriate agents for the system, in order to facilitate coordination among these agents, we define agent groups, and coordinators for each of these groups that control the overall behavior of the system. In the previous section, 6 agents have been identified for this problem domain, as well as three roles that are mapped to the following user goals: to get flight information; to get weather information; and to get road condition information. To manage these three roles of the system, three agent groups are formed and the coordinator for each group facilitates the interaction among agents in each group.

An Agent Interaction Diagram that describes possible interactions among agents is shown in Fig. 11. Typically,
agent interaction diagram is developed for each agent group and then the integrated interaction diagram is generated to show the general structure of the agent organization applied to the problem domain and the interaction among them. Fig. 11 shows the agents that are part of the Flight Info Group in this domain and the interactions between them.

4.3. Agent internal architecture in view of agent autonomy

An agent’s behavior is dictated by its internal architecture and can be modeled to exhibit certain behavior based on the state that the agent is in, and the changes that have taken place in the environment. Agents may choose how to react to a certain situation based on their belief. As mentioned earlier, an agent can have more than one plan to achieve a particular goal. The Plan module consults the belief module, and chooses a proper capability module to execute in light of the current state. In the sample scenario, the internal modules of the UIAgent are shown in Fig. 12.

UIAgent registers itself with the directory facilitator and the group coordinators corresponding to its roles. Upon receiving a user’s request, the UIAgent responds to that request by checking its plan module and selecting the appropriate capability module that should be executed in order to satisfy the request. Generally, the Plan module controls other internal modules and thus a Process-Control paradigm is applied.

UIAgent has various plans in its plan factory and they manage other modules through the controller. In choosing a proper strategy and appropriate capabilities, the UIAgent’s plan module consults its belief module and its goal, which is ‘Achieve [interact with user].’ It also checks the current situation and decides on the future state based on its state diagram. In our approach, the plan module’s situation checking mechanism is represented in a State Diagram using UML. Fig. 13 shows the UIAgent’s states to achieve its goal. For each state, UIAgent’s plan can have substates that have to be achieved and corresponding strategies to achieve them.
4.4. Implementation

This section provides some details of how a MAS could be implemented using the proposed architecture-based approach. As a proof-of-concept, we have implemented a prototype called PTIS, in the intelligent transportation system domain. We employ a simple scenario of a traveler interacting with the system to make travel arrangements and on the day of the travel, receive up-to-date information about weather conditions, subway and road conditions at the time of arrival at the destination city. The prototype has been implemented using JADE (Java Agent DEvelopment Framework) from CSELT, Turin, Italy. JADE is a middle-ware that could be used to develop agent-based applications in compliance with the FIPA specifications for inter-operable intelligent multi-agent systems (Bellifemine, Poggi, & Rimassa, 1999). JADE is java-based and provides the infrastructure for agent communication in distributed environments, based on FIPA standards. Table 3 shows our implementation environments for the prototype.

To implement PTIS prototype, we used or adapted the APIs of JADE; agents and coordinators were implemented by extending the ‘ade.core’ classes, and the agent’s belief modules were programmed using the ‘ade.onto’ classes. JADE supports different agent communication protocols, and depending upon the environment that the receiving agent resides in, it uses the appropriate protocol to deliver the message.

Figs. 14 and 15 show some sample interaction with the system. Fig. 14 shows how to request the flight schedule from the system and Fig. 15 shows how the information is presented to the user. When the system starts, the user logs into the system by providing the userid and a password as shown in the initial screen in Fig. 14. After validating the user, the system provides another screen where the PTIS services are listed from which the user can request a particular service. For example, if the user is interested in the flights information, he or she can click on the ‘Flights’ pulldown menu and make appropriate selections. The user can specify the departure and arrival dates and get the appropriate flight information. Similarly, the user can also request other information such as weather, road conditions, subway timings, etc. using the appropriate pulldown menu.

Once the user makes a request, the UIAgent sends a message to the coordinator agent, which gets the information gathering process started. It coordinates the various activities necessary to satisfy the user’s request.
For example, to get the current information about traffic conditions at the destination city, the system has to send a message to a data acquisition agent which in turn might send a message to another DBWrapper agent to contact the appropriate remote database that resides in an external environment. Agents in this federation collaborate with each other in order to satisfy user requests. Fig. 15 shows some sample message passing between agents during the course of working collaboratively on a problem. The results from various agents are synthesized and presented to the user through the UIAgent. Fig. 15 shows the results of flight schedule search for a particular destination requested by the user. While the sample scenario explained here is a fairly simple application, it does demonstrate the feasibility of our approach and provides face validity for our MAS architecture development method.

5. Summary and future research

There is an increasing trend towards implementing distributed software applications using multi-agent systems (MAS) architecture. This paper has presented an architecture-based method for MAS development, which advocates a phased approach for systematic software development. Our approach focuses on important MAS properties such as agent coordination and autonomy. A goal-based approach is used for the problem domain.
analysis, and agents are mapped to system’s goals. UML and ADL are used for modeling and formalizing the architecture, and finally, we have applied our architecture-based method to PTIS domain and implemented a proof-of-concept prototype.

It is necessary to develop agent-oriented software based on agents’ coordination and autonomy, making it possible to provide complicated services in distributed environments. To this end, agent-oriented software development methodology should be systematized and this paper has presented a robust architecture by suggesting an architecture-based development method for MAS. While we have focused on the two most important characteristics of a MAS (agent coordination and agent autonomy), we did not include other desired properties such as agent security and mobility. Currently we are extending our approach to integrate these characteristics into the MAS architecture and investigate how to model the internals of agents to reflect these additional characteristics. Ideally, modeling of agents to support coordination and autonomy, as well as capturing system security and supporting agent mobility should proceed in parallel during MAS development. Our future work also includes development of architecture validation tools and support for traceability between the multiple views.

**Acknowledgement**

This research was supported by the MIC (Ministry of Information and Communication), Korea, under the ITRC (Information Technology Research Center) support program supervised by the IITA (Institute of Information Technology Assessment) and Oakland University.

**References**


