Design and Implementation of a FIPA based Agent Communication Model for a Logic Programming Framework

Mariano Tucat
mt@cs.uns.edu.ar

Alejandro J. García
ajg@cs.uns.edu.ar

Consejo Nacional de Investigaciones Científicas y Técnicas (CONICET)
Artificial Intelligence Research and Development Laboratory
Department of Computer Science and Engineering, Universidad Nacional del Sur,

Abstract

In this paper we consider the design of an agent communication model based on the FIPA Architecture and FIPA Interaction Protocols. Our goal is to allow the agents to reach each other by their characteristics and to exchange messages using a standard Agent Communication Language. We propose the design and implementation of a framework as an extension of Prolog, following the spirit of Logic Programming. In our framework, the agents are allowed to register themselves in MASs in order to allow other agents to locate them easily. The agents may search for agents having specific characteristics, or providing determined services in order to interact with them.

Keywords: Agent Communication, Interaction Protocols, Logic Programming

1 INTRODUCTION

Interaction is an essential characteristic of Multi-Agent Systems (MAS). Agent interactions are usually performed by exchanging messages according to some conversation policy or by executing services upon requests made by other agents. Therefore, the ability to locate other agents and communicate with them are features that need to be implemented in agents that are part of a MAS.

The goal of this paper is to design an agent communication model based on the FIPA Architecture [6] and FIPA Interaction Protocols [5]. Thus, our model should allow the agents both to reach each other by their characteristics and to exchange messages. The message exchange should be done using standard Agent Communication Languages (ACLs), in order to allow the interaction among heterogeneous agents.

Suppose as an example that there exist some agents selling books and other agents buying books on behalf of their users. The agents buying books will need to find the agents selling books

Partially supported by CONICET (PIP 5050) and SGCyT Universidad Nacional del Sur (24/ZN11) and interact with them, but only the ones that sells the books they are interested in. Another similar situation may arise when we have agents controlling the printers of an organization and there exist agents that are trying to print documents on behalf of their users.

The theory of agent communication languages and conversation policies has received a lot of attention in recent years [1, 3, 11, 14]. However, the creation of tools for implementing these formalisms has been progressing at a slower pace, and it is clear that the techniques resulting from such contributions will only be widely adopted when suitable programming languages and tools are available.

There exist different alternatives for implementing MASs. One alternative is to implement the whole MAS ad-hoc. Another way is to use a MultiAgent development framework such as JACK [10], JADEX [13] or 3APL [9]. Finally, another alternative is to implement an MAS using a Programming Language extension that provides the capabilities of finding other agents and exchanging messages.

The alternative of implementing the whole MAS ad-hoc means that the developer of the system is allowed to choose the architecture of each agent, the way they interact and also the way they locate each other. Thus, this alternative has the advantage of a great flexibility in the design and implementation of the system. However, the main disadvantage is that the developer may have to implement everything, including the mechanisms used to locate the agents and also the primitives for exchanging messages.

The alternative of using a MultiAgent development framework such as JACK, JADEX or 3APL, has the advantage of reducing the amount of work needed to implement the system. However, this alternative constrains the developer of the systems to use a specific agent architecture and may also determine the way in which the agents should exchange messages.

Finally, the alternative of implementing the MAS using a Programming Language extension or a framework that provides the capabilities of finding other agents and exchanging messages (such JADE [2] or MadKit [8]) allows a flexible design of the system. This way of implementing the MAS tries to maintain the advantages of the alternatives mentioned before also avoiding their disadvantages.

Our proposal corresponds to the last alternative. Since Logic Programming is widely adopted for the development of intelligent agents, we propose to design and implement a framework as an extension of this language. The extension consist of a set of primitives that follows the spirit of Logic Programming: *to provide a specification of the solution and to hide as much of the implementation details as possible.*

The proposed framework should provide a reliable way for programming communicative agents without dealing with low-level details such as the actual location of an agent. In our framework, agents register themselves in MASs in order to allow other agents to locate them easily. The agents may group themselves by their common characteristics, or by the services provided. Thus, any agent may search for agents having specific characteristics, or providing determined services. Whenever an agent registers itself, it sets the name that other agents will use to identify it.

2 THE PROPOSED AGENT COMMUNICATION MODEL

The main goal of our model is to allow the agents to reach each other by their characteristics or services, without worrying about their locations. Once reached, the agents will exchange messages. This message exchange should be held using standard Agent Communication Languages (ACLs), in order to allow the interaction among heterogeneous agents (*i. e.*, agents developed using different languages or technics).

Thus, in our model, agents register themselves in MASs in order to allow other agents to locate them easily. The agents may group themselves by their common characteristics, such as the type of printer the agent controls, or by the services provided, such as selling science fiction books. Thus, any agent may search for agents having specific characteristics, or providing determined services.

Whenever an agent registers itself, it sets the name that other agents will use to identify it. The agent may choose any desired name, especially a location independent name. Since this name will be used to identify the agent, it must be unique. The entity in charge of maintaining the registered agents is responsible for the uniqueness of the agent names. This entity may reject any agent trying to register itself with an already used name.

Figure 1 depicts our Agent Communication Model. An agent in our model is an standalone program with the capability of interacting with other agents through its Communication Library (CL). The CL provides communication, thus allowing the agent to register itself in different MASs, searching for specific agents and exchanging messages with them. In our model, there exists a special agent, called Yellow Pages Agent (`YPA`), responsible for maintaining all the information of the agents registration in the different MASs and also guarantying the uniqueness of the names identifying the agents.

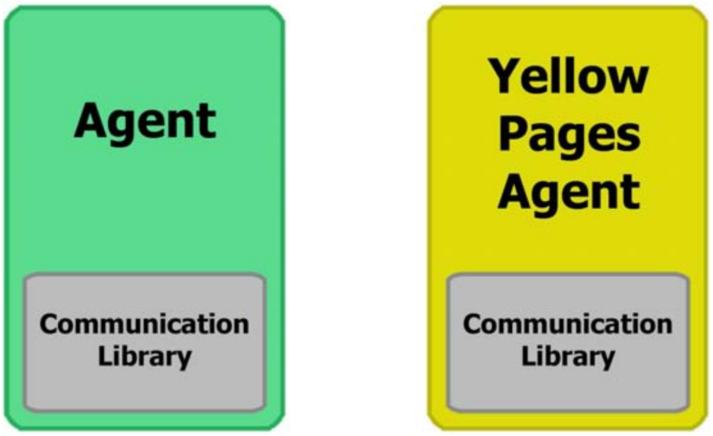


Figure 1: The Proposed Agent Communication Model

Since we based our model in the FIPA Standards [4], it has some similarities with the reference model provided by FIPA, but it also has some differences as we will explain next. The FIPA Reference Model (see Figure 2) includes the agents, the Agent Management System (AMS), the Directory Facilitator (DF) and the Message Transport System (MTS), all of them being part of the Agent Platform (AP), that provides the physical infrastructure in which agents are deployed.

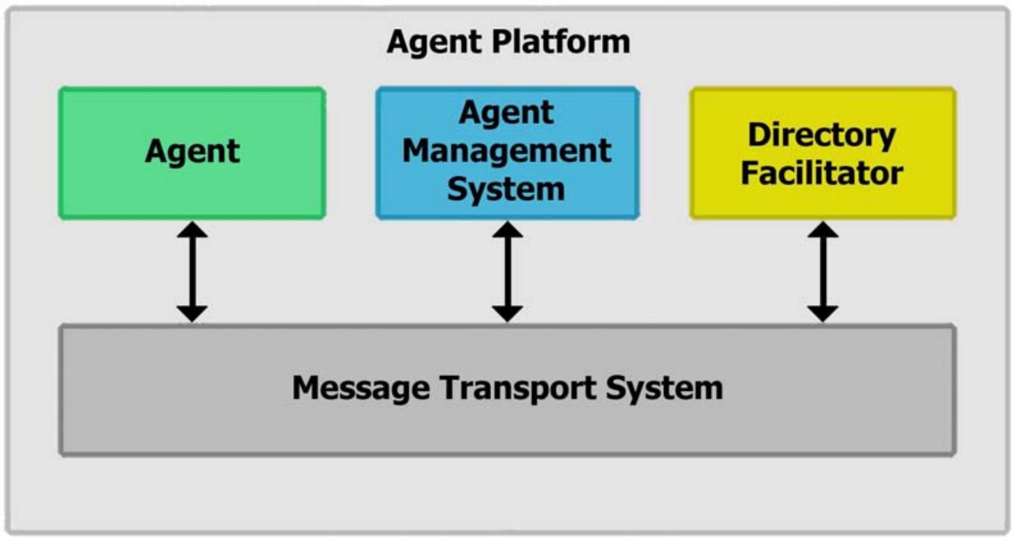


Figure 2: FIPA Reference Model

The AP consists of the machine(s), operating system, agent support software, FIPA agent management components (DF, AMS and MTS) and agents. In our model, we propose that each agent should be an stand-alone program capable of interacting directly with other agents. Thus, we do not provide an AP in which the agent are deployed.

Another difference with the FIPA Architecture corresponds to the existence of the AMS. The AMS exerts supervisory control over access to and use of the AP. The AMS also maintains a directory of AIDs which contain transport addresses for agents registered with the AP. In our model, we avoid the existence of the AMS since the AP does not exist. The directory of AIDs containing transport addresses will be held by the `YPA`, and these addresses will be kept hidden to the agent developer, which will use only the name chosen by the agents.

Our `YPA` is based on the DF provided by the FIPA Reference Model. The DF provides yellow pages services to other agents. Agents may register their services or characteristics with the DF or query the DF to find out what services are offered by other agents. In our case, our `YPA` has also the responsibility of guarantying the uniqueness of the names identifying the agents registered.

As we will see in detail in Section 3, the agents in our model uses FIPA ACL and FIPA Interaction Protocols in order to interact with the `YPA`, thus allowing any agent FIPA complaint to interact with them. In order to interact, the agents have the CL, which is similar to the MTS, with the only difference that our CL is used to exchange messages among any agent, whereas the MTS is the default communication method between agents on different APs.

In order to facilitate the development of agents in our model, we propose a small set of primitives that allow them to interact with the `YPA`, registering themselves and also searching for specific agents. This set of primitives includes two primitives that allow the agent to connect to a specific `YPA` (`connect` and `disconnect`), two primitives for registering specific services or characteristics (`register` and `deregister`) and finally, two primitives for searching for agents with specific characteristics or providing certain services (`which_agents` and `which_MASs`).

- connect(+Name, -Error)

- disconnect(-Error)
- register(+Characteristics, +Ontology, -Error)
- deregister(+Characteristics, +Ontology, -Error)
- which_MASs(-List_Of_MASs, +Ontology, -Error)
- which_agents(-List_Of_Agents, +Characteristics, +Ontology, -Error)

Suppose that we want to develop a system that decides, on behalf of the user, which is the better printer to use in order to print a specific document. Thus, one way to accomplish this task is to develop an agent for each printer and agents that interact with them in order to print documents. In our model, the agents controlling the printers will register themselves in MASs according with their characteristics and the agents willing to print documents will search for agents in specific MASs depending on the desired characteristics of the printer.

The agents controlling the different printers will connect to the `YPA`, using the primitive `connect/2`. Suppose that we have two monochromatic laser printers and that we also have three color ink-jet printers. Thus, the agents controlling these printers will use the primitive `connect(Name, Error)` (see 1 on Figure 3), in order to be able to interact with the `YPA`.

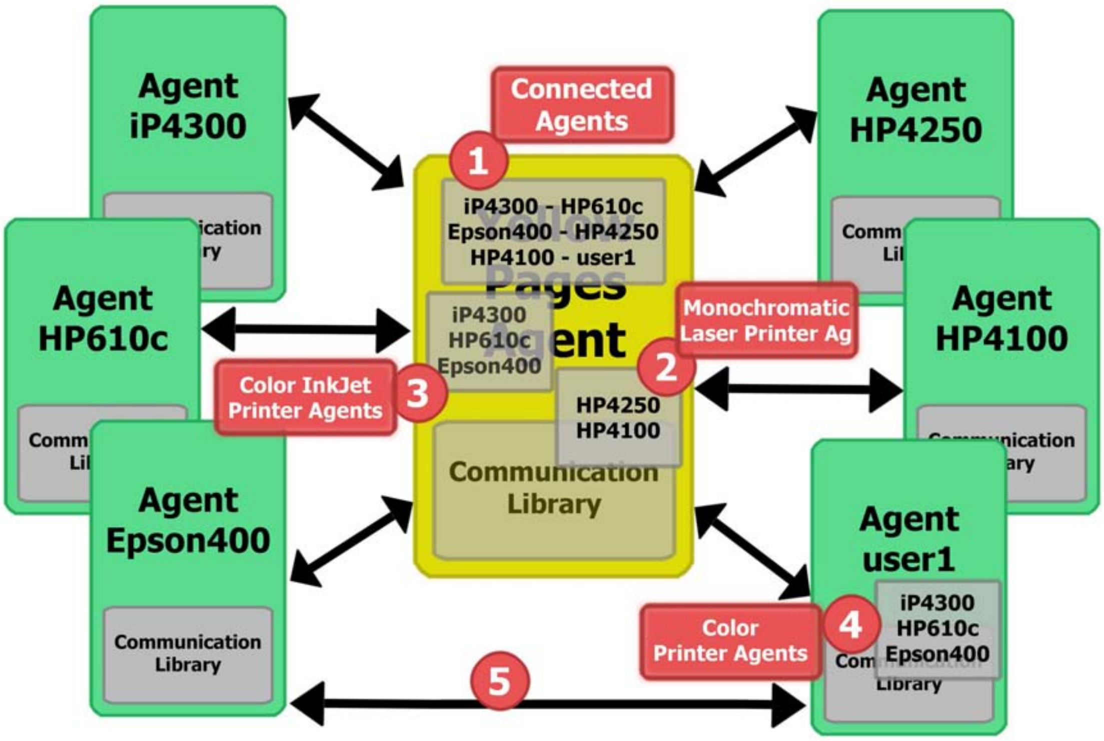


Figure 3: The agents interacting with the `YPA`

Once connected with the `YPA`, they will register themselves, depending on their characteristics, using the primitive `register/3`. For example, an agent controlling a monochromatic laser printer may registry itself calling `register(printers([color(no), type(laser)]), computer_printers, Error)` (see 2 on Figure 3), thus identifying its characteristics and determining the ontology of the terms used. An agent controlling a color ink-jet printer may use `register(printers([color(yes), type(ink-yet)]), computer_printers, Error)` (see 3 on Figure 3).

After that, any agent willing to print a specific document will have to connect to the `YPA` and then search for the needed printer. In order to search for the existent MASs, the agents may use `which_MASs/3`. It may call `which_MASs(List, computer_printers, Error)`, obtaining in `List` the existent MASs (*i. e.*, `printers([color(no), type(laser)])` and `printers([color(yes), type(inkjet)])`).

In the case that the document to print is in color, the agent will try to find a color printer, looking for agents in the MAS having printers with this characteristic. Thus, it may call `which_agents(List, printers([color(yes)]), computer_printers, Error)`. In our example, the agent `user1` will obtain in `List` the names of the three color ink-jet printers (see 4 on Figure 3). Then, the agent may interact with any of the agents controlling the color printers, for example, it may interact with the agent `Epson400` (see 5 on Figure 3).

3 FRAMEWORK IMPLEMENTATION

As mentioned above, in order to facilitate the development of agents in our model, we propose a set of primitives that allow the agents to interact with the `YPA`. This set of primitives is provided by the CL, and in this section we will explain in detail these primitives and one way of implementing them in Prolog. Our framework also provides an implementation of the `YPA` that supports the registration of agents and the search of them through their characteristics or services provided.

This set of primitives was designed and implemented as an extension of Logic Programming since this language is widely adopted for the development of intelligent agents. These primitives provide a reliable way for programming communicative agents without dealing with low-level details such as the actual location of an agent. The framework also facilitates the use of sophisticated Knowledge Representation and reasoning formalisms already developed for LP.

As we mentioned in section 2, we use FIPA ACL and the FIPA Interaction Protocols in order to interact with the `YPA`, thus allowing any agent FIPA complaint to interact with our agents. The `YPA` implements the interaction protocols defined by FIPA, therefore any agent, developed from scratch or using another FIPA complaint framework, interacting with it and following these standard protocols, will obtain the corresponding answers.

The set of primitives proposed allows the agents to register themselves and also to search for specific agents. This set includes two primitives to connect to a specific `YPA` (`connect` and `disconnect`), two primitives for registering specific services or characteristics (`register` and `deregister`) and two primitives for searching for agents with specific characteristics (`which_agents` and `which_MASs`).

The primitives to connect to a specific `YPA` have as primary goal to establish the connection between the agent and the specified `YPA`. Once the connection is established, the agent will register itself with a proposed name and the `YPA` may agree or refused the request. In the case that the chosen name is unused, the `YPA` will agree, whereas in the other case, it will refuse.

A simplified version of the predicate `connect/2` can be seen in Figure 4. The predicate creates an address to receive messages from other agents (calling `listen/1`), and then it establishes the connection with the `YPA` (calling `establish_connection/3`), using the obtained address (`get_ypa_address/2`). The address of the `YPA` can be set using a specific predicate (`set_ypa_address/2`). Note that it may connect with any `YPA` running on the same network.

```
connect(+Name, -Error) :-
  listen(-Port),
  get_ypa_address(-Host, -Port),
  establish_connection(+ypa, +Host, +Port),
  send_connection_request(+ypa, +Name),
  get_answer(-Answer),
  ( Answer = done,
    Error = no_error
  ;
    Answer = failure,
    Error = used_name(+Name)
  ).
```

Figure 4: A simplified version of the `connect/2` predicate

After the connection is established, the agent starts the interaction sending the corresponding request message to the `YPA` (`send_connection_request/2`) and waits for the answer. Depending on the result of the interaction, the term `Error` will be instantiated with `no_error`, indicating that the connection was successfully accomplished, or it may be instantiated with `used_name(Name)`, indicating that the name is already used and that the agent should choose another one.

The predicate `disconnect/1` closes the connection established with the `YPA`, also eliminating any registered characteristic. Once done, the agent will not be allowed to register any characteristic or service provided, or even to search for any specific agent, until it connects again with an `YPA`. In order to do this, the predicate will first interact with the `YPA` using the corresponding protocol and then it will close the connection.

The primitive `register` allows the agents to register specific services or characteristics into different MASs, thus, allowing other agents to locate them easily. In the case of the `deregister`, it allows the agents to eliminate any characteristic or service registered before. Both primitives require that the agent and the `YPA` have already established a connection using the primitive `connect/2`, explained above.

In Figure 5, a simplified version of the primitive `register/3` is shown. The predicate interacts with the `YPA` by sending a request for registering the corresponding characteristics (`send_register_request/3`) and then it waits for the result of the interaction. Finally, it instantiates the term `Error` accordingly.

```
register(+Characteristics, +Ontology, -Error) :-
  send_register_request(+ypa, +Characteristics, +Ontology),
  get_answer(-Answer),
  ( Answer = done,
    Error = no_error
  ;
    Answer = failure,
    Error = error_registering_characteristics
  ).
```

Figure 5: A simplified version of the `register/3` predicate

In the case of the primitive `deregister/3`, it is similar to the register primitive, with the only difference that it will have exactly the opposite result. That is, it will eliminate characteristics or services from the `YPA`, that have been previously registered.

Finally, we proposed two primitives for searching for agents with specific characteristics:

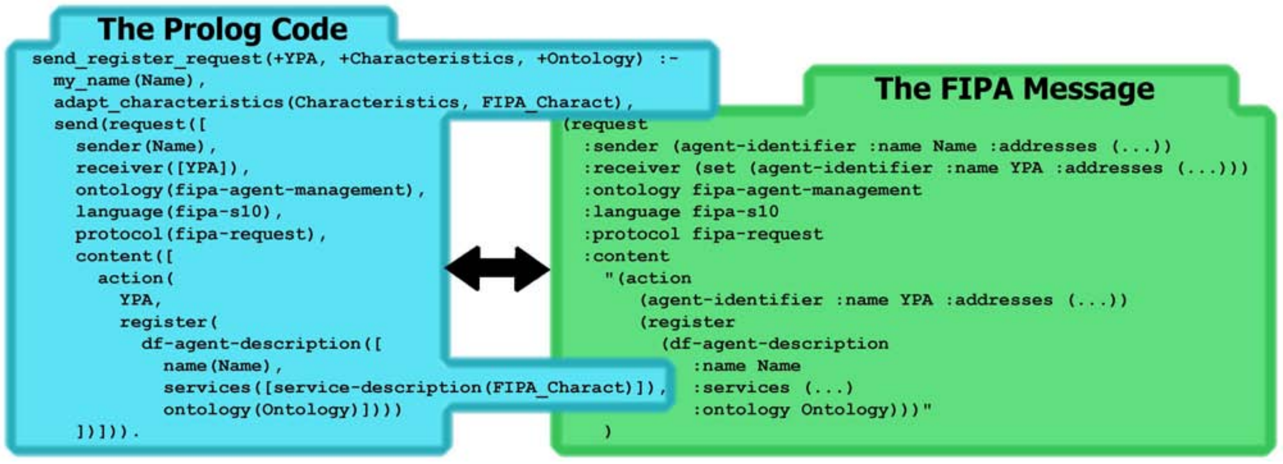
- which_MASs(-List_Of_MASs, +Ontology, -Error)
- which_agents(-List_Of_Agents, +Characteristics, +Ontology, -Error)

The primitive `which_MASs` allows the agent to obtain the different MASs existing in the `YPA`, containing agents with similar characteristics. It will return the list of MASs in the first parameter, corresponding to the ontology specified in the second parameter. Since the implementation of this primitive and the one explained next is similar to the implementation of the predicate `register/3` (shown in Figure 5), it will be omitted.

In the case of the primitive `which_agents`, it allows the agents to acquire the information of all the registered agents having equal characteristics or providing equivalent services. Thus, this primitive will return the list of agents in the first parameter, having the characteristics specified in the second parameter and corresponding to the ontology defined in the third parameter.

Note that, although the primitive `which_agents` will only return the names of the agents, the CL will know the agents addresses, in order to be able to send messages. Thus, the CL also provides a set of primitives for the deliver and retrieval of messages, in order to allow the agent to interact with any other agents.

Whenever an agent tries to send a message to another agent, the CL will try to establish a connection with the corresponding agent directly and send the message. The CL may not be able to establish the connection with the agent, depending on different reasons. These possible reasons may be that the CL does not know the agent address or that the agent is not reachable at the location known by the CL. In either case, the CL will contact the `YPA` querying for the last address known of the agent and it will try again to send the message.



```
send_register_request(+YPA, +Characteristics, +Ontology) :-
  my_name(Name),
  adapt_characteristics(Characteristics, FIPA_Charact),
  send(request([
    sender(Name),
    receiver([YPA]),
    ontology(fipa-agent-management),
    language(fipa-sl0),
    protocol(fipa-request),
    content([
      action(
        YPA,
        register(
          df-agent-description([
            name(Name),
            services([service-description(FIPA_Charact)]),
            ontology(Ontology)])))
    ])])).
```

```
(request
  :sender (agent-identifier :name Name :addresses (...))
  :receiver (set (agent-identifier :name YPA :addresses (...)))
  :ontology fipa-agent-management
  :language fipa-sl0
  :protocol fipa-request
  :content
    "(action
       (agent-identifier :name YPA :addresses (...))
       (register
         (df-agent-description
            :name Name
            :services (...)
            :ontology Ontology)))"
  )
```

Figure 6: The `send_register_request` Predicate and the resulting FIPA Message

The CL uses the FIPA ACL Message Representation in String in order to allow agents developed using this framework to interact with any FIPA Complaint agent. Thus, it provides a way of sending and receiving messages that allow the agent to exchange FIPA Messages. See for example the predicate `send_register_request` of Figure 5. The implementation of this predicate is shown in Figure 6, also showing the FIPA message that the CL will sent.

4 RELATED WORK

There are several agent platforms available for developing multi-agent systems [12]. Most of these platforms are focused either in the cognitive part of the architecture or in the infrastructural one. In other words, some platforms are FIPA-compliant concerning only the infrastructural problem, such as middle-ware issues, whereas other platforms are reasoning-centered, focusing on the behavior of a single agent.

JADE, a Java Agent Development Framework [2], is a software framework to develop agent applications in compliance with the FIPA specifications for interoperable intelligent multi-agent systems. Similar to us, the goal of JADE is to simplify development while ensuring standard compliance through a comprehensive set of system services and agents. Thus, JADE can then be considered an agent middle-ware that implements an Agent Platform and a development framework, dealing with all those aspects that are not peculiar of the agent internals and that are independent of the applications.

However, there exist some differences between our approach and the one proposed by JADE. JADE offers a FIPA-compliant Agent Platform, including the AMS, the DF and the MTS, whereas our approach presents some variations to the FIPA Reference Model explained in detail in Section 2. While JADE has been fully coded in Java and an agent programmer should code his/her agents in Java, our framework is based on Logic Programming (LP), and the agents should be developed using Prolog.

Another approach corresponds to the MadKit Agent Platform Architecture [8], a generic multi-agent platform. This toolkit is based on a organizational model. It uses concepts of groups (similar to our MASs) and roles (similar to our agents characteristics or services) for agents to manage different agent models and multi-agent systems at the same time.

Unlike our approach, the MadKit architecture is based on a minimalist agent kernel decoupled from specific agency models. Thus the platform is not an agent platform in the classical sense. It presents the concept of "agent micro-kernel", handling the control of local groups and roles, the agent life-cycle management and the local message passing. In our approach, the `YPA` maintains the information of the different MASs and the agent characteristics and services, the agents are stand-alone programs and there is no agent life-cycle management, and finally, the CL provides message exchange between agents, independently of their location.

MadKit is focused only the infrastructural problem and it is not centered in the cognitive part of the architecture. In the case of JADE, it provides a full integration with JESS [7] offering a so-called JessBehaviour, whereas our framework, as we already mentioned, is implemented as an extension of LP, and thus, sophisticated Knowledge Representation and reasoning formalisms developed for LP can be easily used.

5 CONCLUSIONS AND FUTURE WORK

In this paper we have considered the development of agent interaction in Multi-Agent Systems. We have proposed a communication model among agents based on the FIPA Architecture and FIPA Interaction Protocols. Our main goal was to allow the agents both to reach each other by their characteristics and to exchange messages. In order to allow the interaction among heterogeneous agents, we have used standard Agent Communication Languages in the message exchange.

We have designed and implemented a framework as an extension of Prolog since this language is widely adopted for the development of intelligent agents. The framework corresponds to a set of primitives that follows the spirit of Logic Programming. Agents may register themselves in MASs in order to allow other agents to locate them easily. Any agent is allowed to search for agents having specific characteristics, or providing determined services. Thus, the framework proposed provide a reliable way for programming communicative agents without dealing with low-level details such as the actual location of an agent.

The resulting framework has some limitations that we have addressed as future work. One limitation concerns security aspects, for example, the `YPA` may allow the agents to create private MASs and restrict the access to specific agents. We are also planning to extend the framework adding implementations of the standards Interaction Protocols defined by FIPA. Another possible extension corresponds to agent mobility, since in our framework, agents identifies each other by their names, and this feature simplifies the implementation.

REFERENCES

[1] B. Bauer, J. P. Mller, and J. J. Odell. Agent UML: A Formalism for Specifying Multiagent Interaction. In *Agent Oriented Software Engineering,*. Ciancarini y Wooldridge (ed.), 2001.

[2] F. Bellifemine, A. Poggi, and G. Rimassa. JADE – A FIPA-compliant agent framework. In *Proceedings of the 4th International Conference and Exhibition on The Practical Application of Intelligent Agents and Multi-Agent Technology (PAAM'99)*, pages 97–108, London, UK, April 1999.

[3] T. Finin and Y. Labrou. Agent Communication Languages. In *Proceedings of ASA/MA'99, First International Symposium on Agent Systems and Applications, and Third International Symposium on Mobile Agents*, 1999.

[4] FIPA. Foundation for Intelligent Physical Agents. http://www.fipa.org.

[5] FIPA. Interaction Protocol Library Specification, November 2000.

[6] FIPA. Abstract Architecture Specification, December 2002.

[7] Ernest J. Friedman-Hill. *Jess, The Java Expert System Shell.* Sandia National Laboratories, Livermore, CA, USA, March 1998. Version 4.0.

[8] Olivier Gutknecht and Jacques Ferber. The MADKIT agent platform architecture. In Thomas Wagner and Omer F. Rana, editors, *Agents Workshop on Infrastructure for Multi-Agent Systems*, volume 1887 of *Lecture Notes in Computer Science*, pages 48–55. Springer, 2000.

[9] K. V. Hindriks, F. S. De Boer, Hoek Wiebe van der, and J. Jc Meyer. Agent programming in 3APL. *Autonomous Agents and Multi-Agent Systems*, 2(4):357–401, 1999. Publisher: Kluwer Academic Publishers, Netherlands.

[10] JACK. JACK Intelligent Agents Framework. http://www.agent-software.com/.

[11] Y. Labrou. Standardizing agent communication. In *Proceedings of the Advanced Course on Artificial Intelligence (ACAI'01)*. Springer-Verlag, 2001.

[12] Eleni Mangina. Review of Software Products for Multi-Agent Systems, 2002. http://www.agentlink.org/resources/software-report.html.

[13] Alexander Pokahr, Lars Braubach, and Winfried Lamersdorf. Jadex: A BDI reasoning engine. In Rafael H. Bordini, Mehdi Dastani, Jürgen Dix, and Amal El Fallah-Seghrouchni, editors, *Multi-Agent Programming*, volume 15 of *Multiagent Systems, Artificial Societies, and Simulated Organizations*, pages 149–174. Springer, 2005.

[14] M. Wooldridge. Semantic issues in the verification of agent communication languages. In *Journal of Autonomous Agents and Multi Agent Systems*, 2000.