Logic Modules for Communicating Distributed Agents

Alejandro Zunino and Analía Amandi

ISISTAN Research Institute, Facultad de Ciencias Exactas,
Universidad Nacional del Centro de la Pcia. de Buenos Aires
Campus Universitario Paraje Arroyo Seco - (7000) Tandil - Bs. As., Argentine
E-mail: {azunino,amandi}@exa.unicen.edu.ar
WWW: http://www.exa.unicen.edu.ar/~isistan/

Abstract

The development of isolated agents involves the manipulation of components such as actions, mental attitudes and decisions. Multi-agent systems have to deal with these components from different agents, without to forget the privacy and autonomy of each agent. For programming simple agents, both object-oriented and logic paradigms have shown advantages in terms of behavior encapsulation and mental model manipulation, respectively. Multi-paradigm languages allowing the usage of both programming paradigms can solve some troubles in agent-oriented programming. However, problems about interaction among agents, especially in physically distributed environments, can not directly be solved with that solution. This paper presents an approach based on the usage of logic modules for supporting interaction among agents in a multi-paradigm environment. Logic modules encapsulate a part of a mental state of an agent, being each logic module composed by a sequence of clauses. Objects representing agents manipulate these logic modules. They can transport their own logic modules to the location of another agent. Thus, agents can integrate logic knowledge to their own decision process.

1 Introduction

Agent-oriented programming has been introduced as a specialization of object-oriented programming [Sho93]. This presentation was materialized by both languages using object-oriented concepts for supporting agent programming (i.e. AgentSpeak [WRR94] and Metatem [Fis94]) and the usage of object-oriented language for programming agents.

In spite of those evidences on the significance of object-oriented programming in agent programming, objects managing mental attitudes present one important limitation. The origin of such limitation is the lack of practical object-oriented approaches related to logic formalisms for representing mental attitudes.

Object-oriented programming solves the encapsulation of actions and the hiding of private knowledge of agents. Logic programming, in contrast, allows logic clauses being used for representing mental attitudes. Thus, an integration of both paradigms solves some problematic points to built agents.

For agent programming, we can certainly achieve a simple solution from a multi-paradigm point of view. For multi-agent programming, we must solve one problem more: interaction among agents. To solve this problem involves distribution considerations, since agents may be physically distributed.

This paper presents how interaction among agents can be handled from a multi-paradigm point of view for programming agents. In this way, an integration between Java and Prolog named JavaLog is first introduced.

The structure of this paper is as follow. In section 2 we set the ground for the following discussion making precise our assumptions and defining the conventions to be used in the rest of the article. In section 3 we introduce JavaLog. In section 4 we present how the manipulation of logic modules can resolve the interaction among agents. Finally, we comment on our results.

2 Basic Assumptions

We work on agents following the guides provided by a software architecture [SG96] for building agents named Brainstorm. The Brainstorm architecture [AP97, AP98] prescribes agents supported by an integration of object-oriented and logic paradigms. In Brainstorm, an object containing internal knowledge in logic modules represents agents. Each logic module contains a subset of the mental state of the agent expressed as logic clauses.

For introducing the components of the architecture that manage the integration between both paradigms, we illustrate it by an example, an object-agent salesman. This object has associated logic modules for representing the mental state of the agent. A logic module records a sequence of clauses. The decision component of agents can combine these modules, manipulating thus rules to achieve the wanted behavior.

Being logic modules are defined as a sequence of Horn clauses following the definition of O'-Keefe [O'K85], two algebraic operators are applied for combining logic modules. These operators are named union and overriding union [BLM94].

The Brainstorm approach also allows logic modules as part of methods for manipulating common dealing of mental attitudes of agents that belong to one agent class.

In resume, the manipulation of logic modules represents the basis of the agent programming from the Brainstorm point of view. Such a manipulation is constrained to each agent for supporting privacy on the decisions of each agent. We show a way for supporting such a view for programming agents and, additionally, we present a way for manipulating these logic modules on distributed environments.

3 Programming agents from a multi-paradigm view

JavaLog[1] is the result of integrating the Java language and a Prolog interpreter following the Brainstorm architecture. In a first step, we built a Prolog interpreter using the Java language, and then we integrate it with Java through preprocessing. We first built our Prolog interpreter because we aim that this interpreter composed by classes can be specialized for supporting logic extensions to manipulate complex mental attitudes.

JavaLog provides integration between Java and Prolog allowing developers to define logic modules into Java instance variables or inside methods. Furthermore, it enables developers to manipulate Java objects from Prolog clauses.

For example, *CommerceAgent* is an agent class that models salesmen. Objects of this class have the ability to select and buy items taking into account user preferences. In order to enable changes on the preferences, they are recorded in logic modules. The following example presents a logic module, which records the preferences of a user for cards and motorcycles.

```
preference(car,[ford, Model, Price]) :-
    Model > 1998,
    Price < 60000.
preference(motorcycle,[yamaha, Model, Price]) :-
    Model >= 1999,
    Price < 9000.
```

Using JavaLog, we can define the *CommerceAgent* class in the following way:

```
public class CommerceAgent {
        private PlLogicModule userPreferences;
        public CommerceAgent( PlLogicModule userPreferences ) {
                this.userPreferences = userPreferences;
        }

        public boolean buyArticle( Article anArticle ) {
                userPreferences.enable()
                type = anArticle.type;
                if (?-preference((#anArticle#,[#type#,#brand#,
                    #model#, #price#]).) buy(anArticle)
                userPreferences.disable()
        }
}
```

[1]JavaLog is available as free software from http: /www.exa.unicen.edu.ar/~amandi

The example defines a variable named *userPreferences*, which references a logic module containing user preferences. When the agent needs to decide if he have to buy a given article, user preferences are analyzed. The *buyArticle* method first enables the *userPreferences* logic module to be queried. In this way, the knowledge included in that module is added to the agent knowledge. Then, an embedded Prolog query is used to test if it is acceptable to buy the article. The Prolog query is started by ?-, following it there is a Prolog term. To evaluate *preference(Type,[Brand, Model, Price]), userPreferences* clauses are used.

The query contains a Java variable enclosed into #. This mark allows developers to use Java objects inside a Prolog clause. In the query, send is used to send a message to a Java object from a Prolog program. For instance, *send(#anArticle#,brand,[],Brand)* in Prolog is equivalent to *Brand = anArticle.brand()* in Java. Finally, the *buyArticle* method ends disabling the *userPreferences* logic module. This operation deletes the logic module from the active database of the agent.

In the example, we have shown how to define Java methods with embedded Prolog, how to use knowledge (represented by Prolog clauses) referenced by a Java variable and how to use Java objects within a Prolog query.

Now we show how to define logic modules embedded in Java programs. For instance, to create an instance of the *CommerceAgent* class we must use its constructor with a logic module as argument:

```
CommerceAgent anAgent = new CommerceAgent(
{{ preference(car,[ford, Model, Price]) :-
        Model > 1998,
        Price < 200000.
   preference(motorcycle, [yamaha, Model, Price]) :-
        Model >= 1998,
        Price < 9000.}}
)
```

In this fragment of program, a logic module is defined as a sequence of Prolog clauses between {{ and }}. Such a module contains preferences of a user.

A preprocessor that translates a Java program with embedded Prolog into a pure Java program handles the integration between Java and Prolog offered by JavaLog.

4 Agent Interaction

In JavaLog, the interaction among agents is performed basically by messages. Messages are enable between two agents or to a common blackboard. As messages can carry logic modules out, logic knowledge can travel among agents both in point-to-point communications and blackboards repositories for group interactions.

For example, the *CommerceAgent* agent is able to find and buy articles based on preferences of users. A user can instruct to this agent about he wants to buy. This instruction is made by sending a message containing a logic module with its preferences and/or a list of articles to buy.

In the following example, the *CommerceAgent* tries to buy a car, a radio and a TV from different Shops. A blackboard is used to store offers. Shops publishing logic modules post these offers. Examples of these modules are:

```
logicModule(Shop1) :- {
    article(tv, [Hitachi TV, 20 in, 800])
    article(radio, [Panasonic, h7823, 80])
}

logicModule(Shop2) :- {
    article(tv, [Philips TV,20 in, 900])
}
```

When the *CommerceAgent* receives a buy message (figure 1), it iterates over the list of elements to buy (tv, car, radio) looking into the blackboard for the offers that satisfy user interests. The *buy* method of the *CommerceAgent* class is exposed:

```
buy( LogicModule userPreferences, Vector articles ) {
        userPreferences.enable()
        for( Enumeration e = articles.elements()   e.hasMoreElements()   )
        if( ?- send(#anArticle#,type,[],Type)
            rcall( //blackboard, article( Type, Offer ) )
            preference( Type, Offer )   )  buy( e )
        userPreferences.disable()
}
```

In the code, *rcall(//blackboard,article(Type,Offer))* looks for articles of type *Type* into the blackboard taking into account user preferences. The first argument of *rcall* specifies the network location of the blackboard. The second argument is a logic query. The coordinator executes this query on the knowledge base composed by logic modules stored into the blackboard. In this example, the first result to rcall is the instantiation *Offer = article(tv,[Hitachi TV, 20 in, 800])*. Using that offer, the *CommerceAgent* analyzes the similarities between article properties and user preferences.

In the example, we have shown how an agent could share its mental state with other agents and can send queries to others agents (i.e. *CommerceAgent* to the coordinator). In JavaLog, an agent could share one or more of its logic modules. This fact implies that other members of the community can use the mental state defined in these shared modules.

JavaLog defines communications mechanisms based on logic modules. They are enumerated by: publication of logic modules (mental state); subscription to a published logic module owned by another agent; delegation using mobility of logic modules (to an agent or a coordinator).

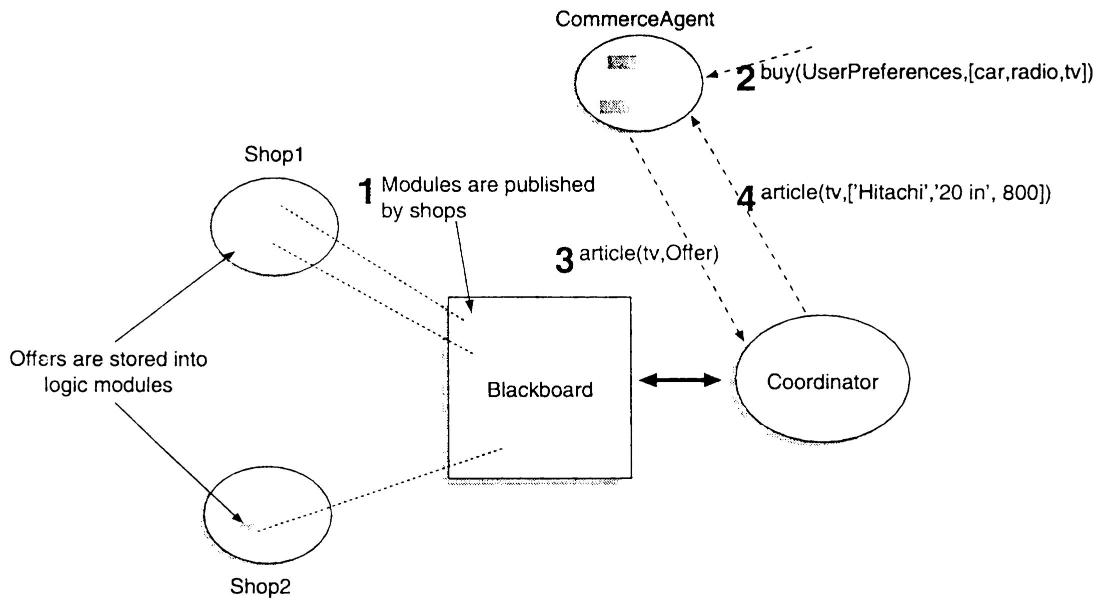
CommerceAgent
2 buy(UserPreferences,[car,radio,tv])
Shop1
1 Modules are published by shops
4 article(tv,['Hitachi','20 in', 800])
3 article(tv,Offer)
Offers are stored into logic modules
Blackboard
Coordinator
Shop2

Figure 1: Interaction among agents

5 Conclusions and Future Work

In this paper, we have addressed agent interaction in multi-agent systems. Our multi-paradigm approach allows developers to solve the interaction problems by combinations of logic modules. This support for programming distributed agents has been used in several developments, showing advantageous related to manipulation of mental attitudes with logic clauses. These advantageous are based on the efficiency and simplicity of JavaLog.

We are working on extending JavaLog to support the construction of mobile agents. The low level mobility mechanisms will be supported by the Aglets framework [LO98].

References

[AP97] A. Amandi and A. Price. Object-oriented agent programming through the brainstorm system. In *PAAM'97 (Practical Applications of Intelligent Agents and Multi-Agents)*, London, April 1997.

[AP98] A. Amandi and A. Price. Building object-agents from a software meta-architecture. *Lecture Notes in Computer Science*, LNCSD9 1515:21–30, 1998.

[BLM94] Michele Bugliesi, Evelina Lamma, and Paola Mello. Modularity in logic programming. *The Journal of Logic Programming*, 19 & 20:443–502, May 1994.

[Fis94] M. Fisher. Representing and executing agent-based systems. In *ECAI-94 Workshop on Agent Theories, Architectures, and Languages*, pages 307–323, Amsterdam, The Netherlands, August 1994.

[LO98] Danny Lange and Mitsuru Oshima. *Programming and Deploying Java Mobile Agents with Aglets*. Addison-Wesley-Longman. Computer & Engineering Publishing Group, 1998.

[O'K85] R. O'Keefe. Towards an algebra for constructing logic programs. In J. Cohen and J. Conery, editors, *Proceedings of IEEE Symposium on Logic Programming*, pages 152–160, New York, 1985. IEEE Computer Society Press.

[SG96] Mary Shaw and David Garlan. *Software Architecture*. Prentice-Hall, New Jersey, 1996.

[Sho93] Y. Shoham. Agent-oriented programming. *Artificial Intelligence*, 60(1):51–92, March 1993.

[WRR94] D. Weerasooriya, A. Rao, and K. Ramamohanarao. Design of a concurrent agent-oriented language. In *ECAI-94 Workshop on Agent Theories, Architectures, and Languages*, pages 386–401, Amsterdam, The Netherlands, August 1994.