Primitives for Building Interaction Protocols

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 propose a set of protocol primitives for the implementation of interaction protocols in Multi-agent systems. These primitives are based on the performatives most frequently used for agent interaction, and provide an abstract way of achieving conversations between agents, facilitating the creation of both standard or domain dependent interaction protocols. The proposed primitives automatically handle the exchange of messages needed to achieve the expected behavior of the interactions. Thus, the agent developer avoids the burden of implementing message processing and also the creation of messages corresponding to specifics agent communication languages. Examples of specific applications for the proposed primitives will be given.

Keywords: Multi-agent systems, Agent interaction, Interaction Protocols.

1 INTRODUCTION

Interaction is an essential characteristic of Multi-Agent Systems (MAS). There, societies of agents interact to collectively perform tasks by entering into conversations-sequences of messages that may be as simple as request/response pairs or may represent complex negotiations. These conversations between agents often fall into typical patterns, called conversation policies or Interaction Protocols (IPs) by FIPA [4]. In such cases, certain message sequences are expected, and, at any point in the conversation, other messages are expected to follow.

Agent communication languages (ACLs), such as KQML [2] or FIPA ACL [5], allow agents to effectively communicate and exchange knowledge with other agents despite differences in hardware platforms, operating systems, architectures, programming languages and representation and reasoning systems [3, 8, 9, 14]. An ACL is used as the medium through which the attitudes regarding the content of the exchange between agents are communicated.

Speech Act Theory divides communication into different types of communication messages, called performatives or communicative acts. These different actions include requests, proposals, queries and informs [1, 12, 10, 13]. The main goal of an agent using these performatives is to change the mental state of the agent receiving the message, and thus, the agent sending the message will want to now whether it made it or not. Interaction protocols are often based on

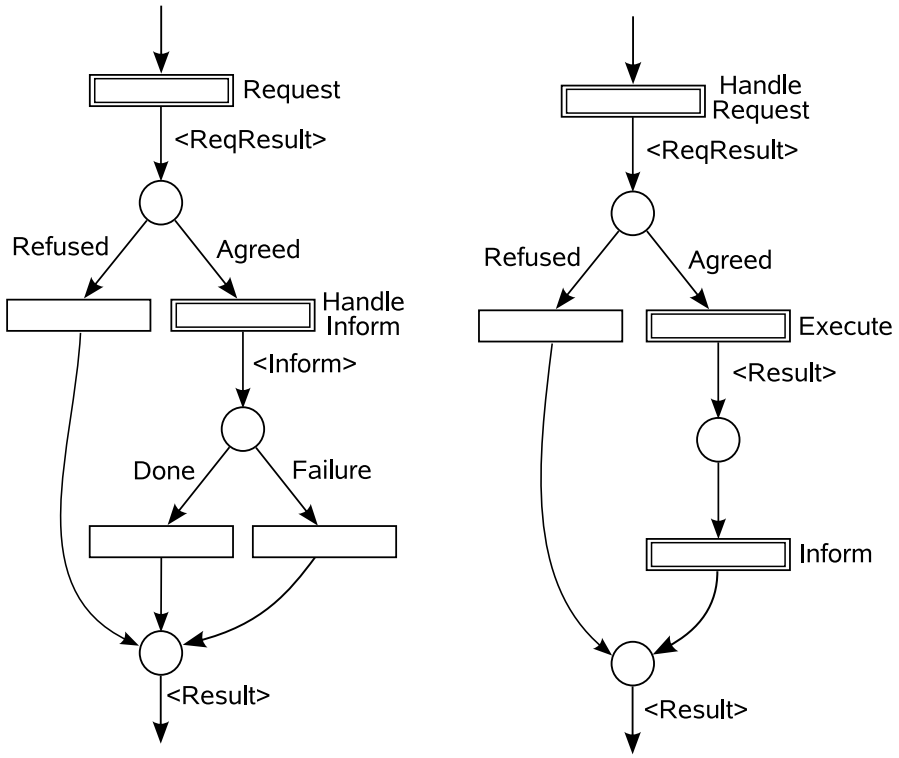
Partially supported by CONICET (PIP 5050), Agencia Nacional de Promoción Científica y Tecnológica (PICT 2002 Nro 13096) and SGCyT Universidad Nacional del Sur (24/ZN11 and 24/N016)

these communicative acts. Some of them are used to transfer information to the hearer, others are used to make the hearer perform a certain action.

Interaction protocols are usually implemented by directly exchanging messages. That is, the implementation of the protocols is based on primitives for sending and receiving messages. The exchanged messages specify the desired 'performative' or 'communicative act' (inform, request, etc.) and they also have a declarative representation of the content of the message. Thus, the developer of the protocol must determine which information it should send in each message and it must also obtain just the messages corresponding to the current interaction. In this work, we will consider only IPs between two agents. IPs concerning more than two agents will be left for future consideration.

Although there exists an almost established standard set of interaction protocols, they are just tested patterns of agent interactions. These IPs need to be adapted to the specific application domain when implementing them in a MAS. Thus, the desired IPs must be completely implemented using primitives for sending and receiving messages, resulting really difficult to reuse already implemented protocols.

In order to avoid these problems when implementing IPs, one alternative is to implement the performatives mentioned above as protocol primitives. In Section 4 we show how to implement them using a set of interaction primitives that extends Prolog (see Appendix A and [6] for details). Implementing them as primitives that return the desired answer allow the developer to use them in order to obtain simple interactions (see Section 5.1) and also to create standard IPs such as FIPA Request (see Figure 1 and Section 5.2) or even more complex IPs.



(a) Initiate (b) Participate

Figure 1: FIPA Request Interaction Protocol represented with RCPN

There exists several ways of specifying interaction protocols. Some of these ways are semi-formal, such as AUML, and some others are formal, such as Colored Petri Nets (CPN). In this

work we will use Recursive Colored Petri Nets (RCPN), an extension of CPN taken from [11] (see Section 4.3). In our case, we will split interactions by roles, and, since we will consider only protocols between pairs of agents, we will have two roles, the Initiator and the Participant of the interaction.

2 PROTOCOL PRIMITIVES

As mentioned before, Speech Act Theory divides communication into different types of communication messages. These different actions include requests, proposals, queries and informs:

- An agent I `requests` another agent P the execution of a particular *action*. That is, I wants P to execute the specified action for it.
- An agent I `proposes` another agent P to execute an *action* for it. That is, I offers to execute the specified action for P.
- An agent I `queries` another agent P for some specific information or *data*.
- An agent I `informs` another agent P about certain *data* it knows.

Using these interactions, an agent may exchange any kind of information needed and require or propose the execution of actions. Thus, these primitives allow agents to maintain meaningful conversations and also facilitates the creation of complex interaction protocols. When an agent inquires another agent by using one of these simple interactions, it will obtain a positive or a negative answer. In the latter case, the inquiring agent will also obtain an explanation. This explanation may be that the receiving agent does not understand part of the inquiry or it may be a domain dependent reason. Accordingly, an agent receiving an inquiry should process it deciding whether to agree or refuse it and, in the latter case, it must also provide a reason.

Example: *In a robotic soccer game, the agents controlling the robots of a team may interact in order to coordinate their actions. Thus, an agent* ***`attacker1`*** *may notice that a teammate* ***`defender1`*** *has possesion of the ball and thus,* ***`attacker1`*** *may ask* ***`defender1`*** *to pass the ball to it. Therefore,* ***`attacker1`*** *initiates the interaction* ***`request`*** *and waits for the answer. In the case of a positive answer,* ***`defender1`*** *compromises to pass the ball to* ***`attacker1`*** *, whereas in the case of a negative answer, it refuses to do it. One possible reason for a refusal may be that agent* ***`defender1`*** *prefers to shoot on goal.*

A `request` is used when an agent I needs another agent P to execute an action for it. Thus, a positive answer means that agent P accepts to execute the action asked by agent I. That is, agent P compromises to execute the action required by agent I. In the case of a negative answer, agent P must give the reasons for the refusal.

`Proposals` are used whenever an agent I desires to offer the execution of an action to an agent P. While a positive answer of P assents and compromises I to execute the specified action, a negative answer will mean that P does not want I to execute the proposed action, and also gives the reasons for this decision.

An agent I `queries` another agent P whenever it wants to know specific information that P may know. Thus, a positive answer means that agent P consents to inform I about the truth of the specified information. In the case of a negative answer, agent P will give I reasons for its refusal, which may correspond to the fact that P ignores the information required or it does not desire to inform it to I.

Finally, an agent I may `inform` another agent P about certain information it knows to be true. Thus, agent P may agree the inform, meaning that it accepts the information mentioned

by agent I. However, agent P may refuse the inform, giving the corresponding reasons, which in this case may refer to misunderstanding or discordance. Note that an agent using this kind of interaction may not be interested in the answer of the informed agent. That is, the agent may not expect any result of the interaction.

These four kinds of interactions allow an agent to ask another one either for the execution of an action or for specific information. They also allow an agent to inform something to another agent and to propose the execution of a particular action. Thus, these interactions allow the agents to maintain meaningful conversations.

3 IMPLEMENTING INTERACTION PROTOCOLS

As mentioned before, IPs are usually implemented by the simple exchange of messages, probably using primitives for sending and receiving messages. That is, the agent developer implements the message processing and it is also responsible for the creation of the messages corresponding to specifics agent communication languages. Thus, the developer must determine which information it should send in each message and it must also receive just the incomming messages corresponding to the current interaction.

Note that, by their nature, agents can engage in multiple dialogues, perhaps with different agents, simultaneously. The term `conversation` is used to denote any particular instance of such a dialogue. That is, agents may be concurrently engaged in multiple conversations, with different agents, within different IPs. Thus, receiving the incomming messages of a particular interaction may not be easy.

Using the primitives for sending and receiving messages has the advantage of allowing the developer to create any particular message structure and also to add any kind of information. Thus, the developer is allowed to use any ACL. That is, he may use a standard ACLs, such as FIPA ACL or KQML, or it may use a specific language defined for the application domain.

However, implementing the interaction protocols this way imposes an extra overhead in the development of the agents. That is, the agent must process every message and deliver it to the corresponding conversation. Having in mind that an agent may have several conversations simultaneously, this message delivery may not be easy, especially in the case of an implementation using threads.

In contrast to this way of implementing the interaction protocols, one alternative is to implement the interactions explained above as protocol primitives. These protocol primitives solve the problem of processing the incoming messages and may also implement any desired ACL. Thus, the agent developer can concentrate in the information or action the agent needs, without worrying in the way it obtains it.

The main advantage of using these protocol primitives is that they provide an abstract way of achieving interaction between agents. These primitives allow agent developers to avoid caring about the exchange of messages needed to interact, and also allows them to focus on the actions and information required. They also ensure that the agent initiating the interaction will obtain a positive or negative answer and, in the case of a negative answer, it will also obtain a reason.

Thus, these interactions handle the exchange of messages needed to achieve the expected behavior, providing a simple way of performing the communication wanted. Besides that, these simple interactions allow the creation of standard IPs, such as those defined by FIPA, and they also facilitate the construction of more complex IPs.

4 SYNTACTIC AND SEMANTIC DETAILS

Our proposed protocol primitives are divided in two groups. The first group implements the initiating part of the interaction and the other group implement the participating part of the interaction. The second group is also divided in two classes. Four primitives handle just one interaction of a specific kind, whereas the other four primitives handle all the interactions of a specific kind.

4.1 Initiating the Interaction

This group of primitives allows an agent to initiate an interaction with another one, asking for the execution of an action or specific information, and also proposing the execution of an action or informing certain data. The syntactic details follow:

- `query (+Agent, +Info, +Context, -Result).`
- `inform (+Agent, +Info, +Context, -Result).`
- `propose(+Agent, +Action, +Context, -Result).`
- `request(+Agent, +Action, +Context, -Result).`

Where, as usual, "`+`" means that the parameter must be instantiated when the predicate is called, "`-`" means that the parameter should normally be uninstantiated and "`?`" means that the parameter may or may not be instantiated. The parameter `Agent` is the name of the agent receiving the interaction, `Info` represents the data being informed or queried, `Context` is a list of parameter-value pairs representing context information of the interaction, such as the ontology being used, `Result` returns the result of the interaction, and `Action` represents the action being requested or proposed.

Semantic Aspects

These primitives initiate the corresponding interaction with the agent specified in the first parameter and with the context included in the third parameter. In the case of `inform` and `query` primitives, the information exchanged represents data known or desired by the agent, respectively. Whereas in the case of `request` and `propose` primitives, the information exchanged represents actions. The last parameter of these primitives returns the result of the interaction. This result may be an agreement, a refusal or a failure. In each case, `Result` will be instantiated with a term having the corresponding result as functor. In the case of an agreement, the result will have as a parameter the action or data agreed. In the case of a refusal or failure, the result will have as a parameter the reason of the refusal or an explanation of the failure, correspondingly.

Note that these primitives not only create the corresponding message using a specific ACL (in our case FIPA ACL) and send it to the specified agent, but they also receive the answer and then instantiate the result with the corresponding value. Moreover, they guarantee that the returned result is 'agreed', 'refused' or 'failure', with the corresponding reasons in the last two cases.

4.2 Participating in the Interactions

The primitives of this group allow an agent to participate in an already initiated interaction. The first four primitives handle just one interaction synchronously, whereas the last four prim-

itives bind the following interactions to the execution of a particular predicate that handles them. The syntactic details follow:

- `handle_query (?Agent, ?Info, +Context, +Process, ?Answer, -Result).`
- `handle_inform (?Agent, ?Info, +Context, +Process, ?Answer, -Result).`
- `handle_request(?Agent, ?Action, +Context, +Process, ?Answer, -Result).`
- `handle_propose(?Agent, ?Action, +Context, +Process, ?Answer, -Result).`
- `bind_query (?Agent, ?Info, +Context, +Process, ?Answer, -Result, +Bind).`
- `bind_inform (?Agent, ?Info, +Context, +Process, ?Answer, -Result, +Bind).`
- `bind_request(?Agent, ?Action, +Context, +Process, ?Answer, -Result, +Bind).`
- `bind_propose(?Agent, ?Action, +Context, +Process, ?Answer, -Result, +Bind).`

The parameters `Agent`, `Info`, `Context`, `Result` and `Action` are the same as explained above. The parameter `Process` represents the predicate to be called in order to determine whether to agree or refuse the interaction, `Answer` is the answer of the interaction, and `Bind` is the predicate to be called whenever the corresponding interaction must be handled.

Semantic Aspects

The parameter `Agent`, when instantiated, determines which agent should initiate the interaction. Otherwise, this parameter will be instantiated with the name of the agent initiating the interaction. The same happens with the second parameter: when instantiated, it determines which is the information or action to be handled; otherwise, the corresponding value will be instantiated. The parameter `Context` determines some properties that the interaction should have. The parameter `Process` indicates the predicate to be called in order to determine the answer to the interaction, and the parameter `Answer` determines the answer to the interaction. This answer must be an agreement or a refusal, and, in the latter case, it must also contain a reason. The parameter `Result`, in the first group of primitives, returns the result of the interaction. This result may be an agremente, a refusal with the corresponding reason, or a failure with some information explaining the reasons. In the last four primitives, the parameter `Bind` determines the predicate to be called in order to handle the corresponding interaction.

As mentioned above, the first four primitives handle only one interaction. That is, when executed, the primitives block the thread of execution of the agent until the corresponding counterpart is executed. However, in the case of the last four primitives, their execution just bind the corresponding interaction to the execution of a particular predicate. That is, from that moment, whenever the counterpart of the interaction is executed, it is automatically handled and the associated predicate is called with the result of this interaction.

4.3 Implementation

In Figure 2 we show a representation of both roles of the request interaction, that is, the Initiator (a) and the Participant (b), using RCPN. In these RCPN, places represent states or decision making points in the interactions. A transition can be either elementary or abstract. Elementary transitions represent the execution of an action or method, while abstract transitions represent a RCPN itself. Arcs from places to transitions, when labeled, work as filters by allowing the firing of a specific transition. Colored tokens means that they have structured information, and a transition may add and modify information in these tokens.

In this interaction, the initiator sends the request and then it waits for the positive or negative answer. In the case of the participant, it receives the request, process it and depending

on the result of this processing it sends the corresponding message. Note that the processing transition corresponds to an abstract transition, that is, a RCPN itself. This transition is domain dependent, and thus, the agent participating in the request must provide it. This transition is restricted to return an agree or a refuse as its result.

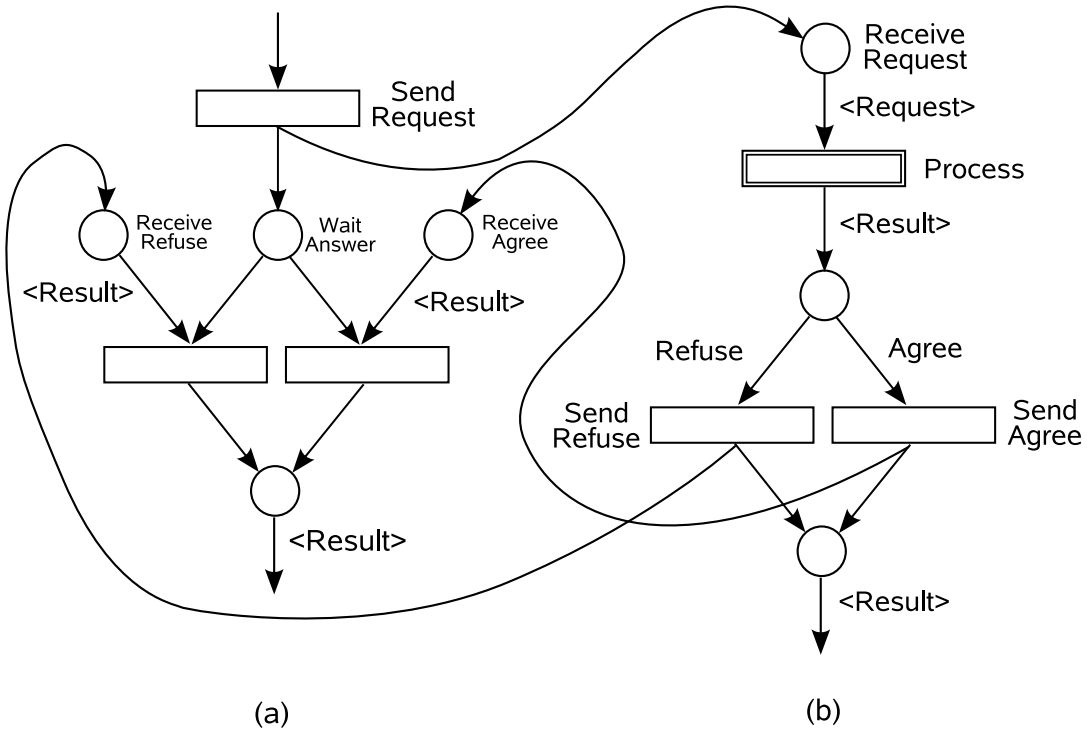


Figure 2: Request Interaction Primitive

We will use two abstract transitions to represent this interaction (see Figure 3), one labeled as `request` representing the initiator role and the other one labeled as `handle request` representing the participant role. The arcs representing exchange of messages between the agents, that is, those arcs that start in a role and end in the other role, will be omitted.

Figure 3: Abstract Transitions of the Request Interaction Primitive

In the case of the others protocol primitives, the abstract transitions used to represent them are equivalent. This representation of the protocol primitives as abstract transitions facilitate the creation of complex interaction protocols using RCPN. It is important to mention that, although the roles of each protocol are represented in different RCPN, whenever a role uses an abstract transition of a protocol primitive, the other role must use the counterpart of this transition.

5 APPLICATIONS

In this section we include two examples in order to show the application of the protocol primitives introduced in this work. In the first one we will show a common situation in which these protocol primitives allow a simple implementation of the interaction needed by two agents that want to synchronize the execution of an action. In the second example we will show how these protocol primitives can be useful to create complex IPs, in particular, we show how to create the FIPA Request IP.

5.1 Robotic Soccer Domain

One possible situation in which a group of agents may use these protocol primitives is in a robotic soccer game. The agents controlling the robots of a team may interact in order to coordinate their actions. As an example, an agent *attacker*1 may notice that a teammate $defender1$ has possesion of the ball and thus, *attacker*1 may ask $defender1$ to pass the ball to it.

In order to obtain the ball, agent *attacker*1 initiates the interaction `request` and waits for the answer. In the case of a positive answer, agent $defender1$ compromises to pass the ball to *attacker*1, whereas in the case of a negative answer, it refuses to do it. Note that in the case of a positive answer, agent *attacker*1 will determine whether $defender1$ is passing the ball to it by its trajectory.

In the case of the agent initiating the interaction, agent *attacker*1, it may execute:

```
request(defender1,pass_ball(X,Y),[ontology(robotic_soccer)],Result).
```

This means that the agent executing it is requesting agent $defender1$ the execution of the action `pass_ball(X,Y)`. The ontology of this request is `robotic_soccer` and the result will be instantiated in `Result`.

In the case of the agent participating in the interaction, agent $defender1$, it may execute:

```
handle_request(attacker1,pass_ball(X,Y),[ontology(robotic_soccer)],
        decide_pass(attacker1,X,Y,Ans),Ans,Result).
```

Thus, agent $defender1$ will be blocked waiting for agent *attacker*1 to request it to pass the ball. In that moment, the predicate `decide_pass` is called and the answer will be the parameter `Ans`. Finally, `Result` will be instantiated with the result of the request, the possible result are "agreed", "refused" or "failure".

This alternative will handle only one request from agent *attacker*1 and will also block the thread of execution of agent $defender1$ until *attacker*1 initiates the request. Another alternative is to bind these specific requests to the execution of a particular predicate. In order to do this, agent $defender1$ may execute:

```
bind_request(attacker1,pass_ball(X,Y),[ontology(robotic_soccer)],
decide_pass(attacker1,X,Y,Ans),Ans,agreed(_),assert(action(pass_ball(X,Y))))
```

In this case, any request made by agent *attacker*1 to pass the ball to it from now on will be handled, the predicate `decide_pass` will determine the answer of the request, and only the *agreed* interactions will execute the `assert`.

5.2 Implementing the FIPA-Request Interaction Protocol

In this example we will show how to implement the FIPA Request IP (see Figure 1). We will show how to implement the initiating and the participating roles of the interaction. In order to do this, these predicates will add context information to the interaction, such as the name of the

protocol and a unique conversation id. In the case of the participating role of the interaction, we will show how to implement the predicate that handles one IP and also the one that handles all FIPA Requests.

In Figure 4 is detailed the Prolog code of the predicate `fipa_request/3` that initiates this IP. This predicate generates a unique conversation id and initiates the specified `Request` to `Agent`. Then, the answer of this request is processed. In the case of a refusal, the predicate returns it in `Result` with the corresponding reason. In the other case, the predicate waits for the inform that specifies the result of the execution of the action, which is instantiated in `Result`.

```
fipa_request(Agent, Request, Result) :-
  generate_id(Conv_ID),
  request(Agent, Request, [protocol(fipa_request),conversation_id(Conv_ID)],Answer),
  result_fipa_request(Answer,Agent,Request,Conv_ID,Result).

result_fipa_request(refused(Reason),_,_,_,refused(Reason)).
result_fipa_request(agreed(_),Agent,_,Conv_ID,Result):-
  handle_inform(Agent, result(Result), [protocol(fipa_request),
                conversation_id(Conv_ID)], true, agree, _).
```

Figure 4: Prolog code for initiating the FIPA Request IP

In Figure 5 is specified the Prolog code of the predicate `handle_fipa_request` that handles one FIPA Request IP. This predicate handles the corresponding request and answers it. In the case of a positive answer, it also executes the corresponding action and inform the result of this execution. In the case of a negative answer, the predicate returns the refusal with the specified reason.

```
handle_fipa_request(Agent, Request, Process, Answer, Result) :-
  handle_request(Agent, Request, [protocol(fipa_request),conversation_id(Conv_ID)],
                 Process, Answer, Ans),
  result_handle_fipa_request(Ans,Agent,Conv_ID,Result).

result_handle_fipa_request(refused(Reason),_,_,refused(Reason)).
result_handle_fipa_request(agreed(Request),Agent,Conv_ID,Result):-
  execute(Request,Result),
  inform(Agent,result(Result),[protocol(fipa_request),conversation_id(Conv_ID)],_).
```

Figure 5: Prolog code for handling one FIPA Request IP

Figure 6 shows the predicate that handles all the FIPA Request IPs by associating the result of the IP to a specified predicate `Bind`.

6 CONCLUSIONS AND FUTURE WORK

In this paper we have proposed a set of protocol primitives based on the performatives most frequently used in the creation of interaction protocols. These primitives provide an abstract way of achieving interaction between agents, facilitating the creation of standard and also domain dependent IPs. They also allow the agent developer to focus on the actions and

```
bind_fipa_request(Agent, Request, Process, Answer, Result, Bind) :-
  bind_request(Agent, Request, [protocol(fipa_request),conversation_id(Conv_ID)],
         Process, Answer, Result, process_fipa_request(Agent,Conv_ID,Result,Bind)).

process_fipa_request(Agent, Conv_ID, Answer, Result, Bind) :-
  result_handle_fipa_request(Answer, Agent, Conv_ID, Result),
  call(Bind).
```

Figure 6: Prolog code for handling all the FIPA Request IP

information needed, automatically handling the exchange of messages needed to achieve the expected behavior of the interactions. Thus, the developer avoids the burden of implementing the message processing and also the creation of the messages corresponding to specifics ACLs. In this paper we have proposed a specific way of implementing the mentioned protocol primitives in Prolog. We have classified the primitives in two groups, the first one allows an agent to initiate an interaction and the other allows an agent to participate in a specific interaction. We have explained their characteristics and advantages, and we have also described some specifics applications in which these primitives are useful and facilitate the development of the agents. We have shown that the proposed protocol primitives facilitate the creation of standard IPs between pairs of agents, such as the FIPA Request IP. Since this is a preliminary research, further work is required to study the applicability of these primitives to the construction of IPs among groups of agents.

APPENDIX A: INTERACTION PRIMITIVES

We include here some details of the set of primitives we have developed and reported in [6]. These interaction primitives were motivated by the implementation of multi-agent systems for dynamic and distributed environments, where intelligent agents communicate and collaborate. These primitives provide a transparent way for programming agent interaction; this can be done, for example, by using the agents' logical names without considering low level elements like the actual location of an agent, IP addresses or machine names. The primitives allow the implementation of standard Agent Communication Languages like FIPA ACL [4] and KQML [7], and provide tools for developing standard Agent Interaction Protocols.

Primitive	Group	Brief description
connect	conn.	Connects the agent to a particular MAS
disconnect	conn.	Deletes the agent from the specified MAS
my_name	conn.	Returns the agent's name on the specified MAS
which_agents	conn.	Returns the list of the participants of the specified MAS
which_MAS	conn.	Returns the list of existing MAS
send	msg.	Sends a message to one agent
receive/2	msg.	Waits for a specific message
receive/3	msg.	Waits for a specific message for a given period of time
bind	msg.	Binds the arrival of specific messages to the call of a predicate
unbind	msg.	Unbinds the association already made

Figure 7: Brief description of the set of proposed primitives for agent interaction

The resulting framework has the following features:

1. The implemented primitives allow the creation of several independent multi-agent systems inside a LAN. Once an agent runs the initialization predicate (`connect`) it obtains a list of the agents present in the system.
2. An agent can communicate with the other participants using the primitives `send/receive` just knowing their names, regardless of which machine they are actually executing.
3. There are primitives (`bind`) for associating the arrival of a message with the automatic execution of a Prolog predicate, thus allowing event-based programming. Different associations can be made for different messages.

REFERENCES

[1] Philip R. Cohen and Hector J. Levesque. Communicative Actions for Artificial Agents. In Victor Lesser and Les Gasser, editors, *Proceedings of the First International Conference on Multi-Agent Systems (ICMAS'95)*, pages 65–72, San Francisco, CA, USA, 1995. The MIT Press: Cambridge, MA, USA.

[2] T. Finin, R. Fritzson, D. McKay, and R. McEntire. KQML as an Agent Communication Language. In N. Adam, B. Bhargava, and Y. Yesha, editors, *Proceedings of the 3rd International Conference on Information and Knowledge Management (CIKM'94)*, pages 456–463, Gaithersburg, MD, USA, 1994. ACM Press.

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

[5] Foundation for Intelligent Physical Agents. Fipa 97 specification. Part 2, Agent Communication Language. 1997.

[6] Alejandro J. García, Mariano Tucat, and Guillermo R. Simari. Interaction Primitives for Implementing Multi-agent Systems. In *VII Argentine Symposium on Artificial Intelligence*, Rosario, Argentina, August 2005.

[7] KQML. Knowledge Query and Manipulation Language. Oficial Web Page: http://www.cs.umbc.edu/kse/kqml.

[8] Y. Labrou and T. Finin. Towards a standard for an Agent Communication Language. In *American Association for Artificial Intelligence, Fall Symposium on Communicative Actions in Humans and Machines*, 1997.

[9] Yannis Labrou. Standarizing agent communication. pages 74–97, 2001.

[10] Yannis Labrou and Tim Finin. Semantics and Conversations for an Agent Communication Language. In Martha E. Pollack, editor, *Proceedings of the Fifteenth International Joint Conference on Artifical Intelligence (IJCAI-97)*, pages 584–591, Nagoya, Japan, 1997. Morgan Kaufmann publishers Inc.: San Mateo, CA, USA.

[11] Hamza Mazouzi, Amal El Fallah Seghrouchni, and Serge Haddad. Open protocol design for complex interactions in multi-agent systems. In *AAMAS '02: Proceedings of the first international joint conference on Autonomous agents and multiagent systems*, pages 517–526, New York, USA, 2002. ACM Press.

[12] Kumar S., Huber M. J., Cohen P. R., and McGee D. R. Toward a Formalism for Conversation Protocols Using Joint Intention Theory. *Computational Intelligence, Ottawa*, 18:174–228, 2002.

[13] M. Wooldridge and S. Parsons. Languages for Negotiation. In W. Horn, editor, *Proceedings of the Fourteenth European Conference on Artificial Intelligence (ECAI-2000)*. John Wiley & Sons, 2000.

[14] Michael Wooldridge. Semantic Issues in the Verification of Agent Communication Languages. *Autonomous Agents and Multi-Agent Systems*, 3(1):9–31, 2000.