Processing Interaction Protocols in Parallel: a Logic Programming implementation for Robotic Soccer

Mariano Tucat [*]
mt@cs.uns.edu.ar

Alejandro J. García [†]
ajg@cs.uns.edu.ar

Consejo Nacional de Investigaciones Científicas y Técnicas
and
Departamento de Ciencias e Ingeniería de la Computación
Universidad Nacional del Sur
8000 Bahía Blanca, Argentina

ABSTRACT

In this paper we explore different situations in which collaborative agents have to communicate among themselves using standard interaction protocols. We will propose how to process these interactions in parallel without interfering with other agent's activities. Thus, agents will not have to interrupt or delay an activity for handling incoming messages, and in some cases, answers can be created and delivered immediately.

Our proposal will be oriented to a robotic soccer domain with autonomous mobile robots. We will analyze three kinds of situations in which the interaction between agents plays an important role for coordination: *requirements, queries* and *proposals.* Requirements arise when an agent asks another to execute a specific action. A query is used when an agent wants to acquire certain information from another agent. Finally, proposals arise when an agent wants to synchronize with another agent for collaboration.

In a realistic scenario, an agent may interact with several agents and these interactions usually proceed simultaneously with the rest of the activities of the agent. Therefore, our proposal is to process these interactions in parallel with the decision cycle of the agent reducing the overhead imposed by the interaction. The implementation of this approach will be done in an extended logic programming framework developed for implementing multi-agent systems.

Keywords: Interaction Protocols, Parallel Processing, Logic Programming, Multi-Agent Systems

[*]Fellowship of Consejo Nacional de Investigaciones Científicas y Técnicas (CONICET).

[†]Partially supported by CONICET (PIP 5050), Universidad Nacional del Sur (24/ZN09), and Agencia Nacional de Promoción Científica y Tecnológica (PICT 2002 Nro 13096)

1 INTRODUCTION

In this paper we explore different situations in which collaborative agents have to communicate among themselves using standard interaction protocols. We will propose how to process these interactions in parallel without interfering with other agent's activities. Thus, agents will not have to interrupt or delay an activity for handling incoming messages, and in some cases, answers can be created and delivered immediately.

Our proposal will be oriented to a robotic soccer domain with autonomous mobile robots. We will analyze three kinds of situations in which the interaction between agents plays an important role for coordination: *requirements, queries* and *proposals.*

Requirements arise when an agent asks another to execute a specific action. For example, any agent in the field may ask the one who has possession of the ball to pass it to a specific location. Another possible situation of a requirement can be when, based on specific game situations such as their locations in the field, an agent requires to shift its role with another one.

A query is used when an agent wants to acquire certain information from another agent. As an example, an agent may send a message to its teammates, asking them to inform their location in the field. The agent may also want to know the score of the game, and it may query other non-player agents like the coach or referee.

A proposal arises when an agent wants to synchronize with another agent for collaboration. In this case, it may propose the execution of the specific action to its teammate and it should wait for the answer. This kind of interaction may be useful, for example, when an agent with possession of the ball wants to pass it to a teammate and it wants to coordinate explicitly this action.

In order to implement the interactions mentioned above, the agents controlling the robots must be able to exchange messages with each other. For this purpose, we will use a set of interaction primitives we have developed and reported in [7] (a brief description is also included in Appendix B). These interaction primitives were motivated by the implementation of multi-agent systems in dynamic and distributed environments, where intelligent agents communicate and collaborate.

In this work, we will show one way of implementing these different types of interaction in a specific domain of robotic soccer: the E-League [2] (for more details see Appendix A). This league is a competition in which two teams of four mobile robots play soccer and the challenge is centered on the design of the intelligence of the robots. Considering that each robot can be seen as an agent, this domain offers an ideal situation for the development of a multi-agent system (MAS).

In a realistic scenario of a multi-agent environment, an agent may interact with several agents and these interactions usually proceed simultaneously with the rest of the activities of the agent. Therefore, our proposal is to process these interactions in parallel using the mentioned set of primitives that extends logic programming providing communication among agents.

The rest of the paper is organized as follows: in Section 2 we briefly describe the design of a multi-agent system that was implemented for controlling a robotic soccer team. In Section 3 we show how to extend this design in order to allow agent interaction and to process them in parallel with other agent's activities. In section 4 conclusions and future work are discussed. Two appendix are included at the end of the paper in order to provide details about topics related to this approach.

2 COORDINATING A SOCCER TEAM OF MOBILE ROBOTS

Robotic soccer has drawn much attention in the area of multi-agent system research and development. This complex and challenging application domain is useful in the evaluation of different kinds of developments that have been carried out in the field of mobile robotics. For example, robots playing soccer need to be able to react to the current state of the environment and also try to carry out plans to change the environment to some predicted future state.

One advantage of this particular application domain is that the game of soccer can be seen as a well defined system: the number and type of players, duration of play, allowed behaviors, and penalizations (among other aspects of the game) are governed by a well defined set of rules. Each team is composed of players that should cooperate and also coordinate in order to reach their goal of winning the game.

Coordinating agents [3, 4, 9, 11] in a dynamic environment, such as a mobile robots, is a difficult task. Every agent must be able to contribute to the overall purpose of the multi-agent system efficiently and effectively. The coordination should allow each agent to consider the objectives of the MAS, maintaining its own goals in order to ensure a correct behavior.

As a previous work, we were involved in the developed of a multi-agent system to control a team of robots (called Matebots) for the E-League competition in RoboCup 2004 (see [10] and Appendix A for details). As explained in [6], the implementation of the MAS was carried out following a layered design. The proposed design was based on services that were associated with four main layers (see Figure 1), each of which covers different levels of abstraction of the problem to be solved, providing services to the upper layers. Thus, the implementation of some services could be modified without provoking many changes in other layers using these services.

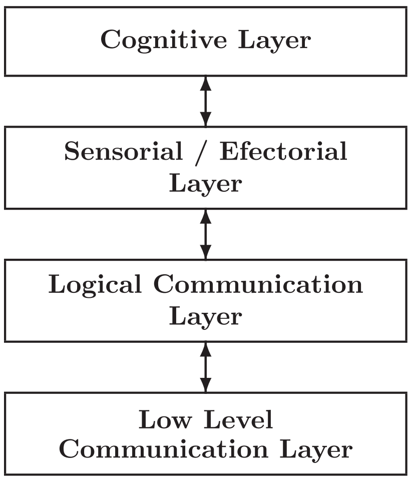
Cognitive Layer
Sensorial / Efectorial Layer
Logical Communication Layer
Low Level Communication Layer

Figure 1: The architecture proposed in [6].

In this work, we will explain how to extend the architecture of Figure 1 in order to implement different interaction protocols that agents may use for coordination. We will also show how an agent may handle the asynchronous arrival of messages from different interaction protocols. In the rest of this section we will briefly explain some details of the mentioned architecture and which part of it will be modified in order to allow the agents to implement interaction protocols (we refer the interested reader to [6] for a complete description of this architecture).

The `Low Level Communication Layer` (see Figure 1) provides access to the basic hardware and software of the league. This includes physical support, such as infrared transmitters, video camera, communication network, and common software.

The `Logical Communication Layer` offers the necessary services that allow communication with the vision and communication servers. In order to provide communication between agents, we must extend this layer to include a set of primitives (for more details see Appendix B) that allow the exchange of messages. Having this primitives, an agent may interact with any other agent sending and receiving messages.

In the `Sensorial/Effectorial Layer`, the visual information is processed and translated into terms that express states of the world. The coordinates and speeds of the robots and ball can be interpreted to express particular situations. Query and action predicates related to particular game situations are defined here.

The `Cognitive Layer` is responsible for the design of the agents that implement the team. Like in any other agent system, the agents perceive the environment and reason about the actions to take. In Figure 2 there is an outline of the implementation of the perception/action loop of each agent. At the beginning of every cycle, the agent will call the predicate `perceive/1` which returns a list (`PercList`) containing all the information the agent can obtain from its environment. This predicate is implemented in the `Sensorial/Efectorial Layer`. Then, the agent will execute the predicate `revise/2`, which is responsible for updating its beliefs and returning them as a list (`RevBeliefs`). With these revised beliefs, the agent will call the predicate `decide/2`, that takes the decision of what action to execute next. These two predicates (`revise/2` and `decide/2`) are implemented in the `Cognitive Layer`. Finally, the agent will call the predicate `do/1` (implemented in the `Sensorial/Efectorial Layer`) which is responsible for executing the chosen action.

```
agent :-
    perceive(PercList),
    revise(PercList,RevBeliefs),
    decide(RevBeliefs, Action),
    do(Action),
    agent.
```

Figure 2: The agent perception/action loop.

The perception/action loop in Figure 2 is what we will modify in order to allow the agents to process the different interaction protocols in parallel with their activities.

3 PROCESSING INTERACTION PROTOCOLS IN PARALLEL

As stated above, we will explore different situations in which the interaction between agents is needed for coordination, in particular those that can be seen as requirements, queries, and proposals. In order to implement these interactions the agents will use the primitives provided by the Logical Communication Layer with the modifications described in the previous section. Thus, an agent may initiate any of the interaction protocol (e.g., request, propose, query) sending the corresponding message to the correct agent and then, it will follow the protocol consequently.

An agent has to be able to receive at any time a message from any other agent, with the purpose of starting a conversation. Therefore, agents should have the capability of handling interaction protocols as soon as possible. Its important to note that messages are received in an asynchronous way, concurrently with the perception/action loop in which the agent decide it actions, and the Logical Communication Layer automatically queues these messages.

There are several alternatives for handling these interaction protocols. One alternative is to process all the received messages from other agents inside the agent's perception/action loop (Figure 2). This can be done by adding a list of received messages to the decision predicate. With this information, the agent can handle every received messages in its decision loop and it can act in consequence.

Although this alternative is simple, it has some disadvantages. The agent perception/action loop must process every received message, and this overhead impose to this cycle delays the agent response to the environment. Also note that this message may probably be received while the perception/action loop is processing another message or deciding what action the robot should execute. Thus, this messages will not be processed immediately, and this means that the reply (when needed) will also be delayed.

A better alternative can be obtained if the interaction between agents is separated in two types:

(a) When the interaction requires the execution of an action by the agent receiving the message.

(b) When the interaction does not require the execution of an action, such as information queries or some specific information updates.

Observe that, in type (a) the agent should handle this kind of interaction inside its perception/action loop, with a particular priority in its reasoning. While in type (b), these interactions could be considered concurrently with the decision cycle of the agent.

In order to do this, every agent may have a main thread executing the perception/action loop and another thread processing the incoming messages. In this message processing thread, those messages of type (b) will be answered immediately whereas the messages of type (a) will be queued in order to be processed during the decision cycle.

This approach will be implemented using the primitive `bind_all/2` (see appendix B). This primitive allows the association of a particular predicate to the arrival of messages. Therefore, when a message is received, the associated predicate is executed immediately, processing the interaction concurrently with the decision cycle of the agent. Thus, those messages of type (b) that do not require the execution of an action will be answered immediately. The rest will be queued in order to be considered in the agent perception/action loop, and priorities among them and possible actions of the agent can be applied.

Figure 3 shows an outline of the implementation of this proposal. There, the predicate `bind_all/2` is executed just once, and it associates the arrival of messages to the execution of the predicate `processMessages/2` which will be in charge of processing the received messages. Observe that this predicate is incomplete. Below we will explain three types of interaction (queries, requirements and proposals) and we will show how to complete this predicate. Also note

```
:- bind_all(processMessages,_).
   % all incoming messages will be processed
   % by processMessages/2

processMessages(_,Msg):-
    % answers immediately whenever possible

processMessages(_,Msg):-
    % otherwise, queues the received Message

agent :-
    perceive(PercList),
    revise(PercList,RevBeliefs),
    retrieveMsgs(MsgList),
    decide(RevBeliefs, MsgList, Action),
    do(Action),
    agent.
```

Figure 3: Outline of our proposal.

that the perception/action loop has been modified and now has an invocation to the predicate `retrieveMsgs/1` that obtains and removes from the queue all the messages that have been received concurrently. These messages, that will all be of type (a), will be considered by the predicate `decide/3` as new possible options for execution. In order to decide which action to execute next, the agent can define priorities among requirements of other agents and actions that it can do by proper initiative.

In the rest of this section we will show in detail how to implement this better alternative following the standard interaction protocols proposed by FIPA [5]. In order to exchange messages, we will use the FIPA Agent Communication Language (ACL) adapted to logic programming syntax. Thus, in this approach, a FIPA message will be a list of terms representing the communicative act and its parameters (see Figure 4). Also note that the agents will use a common ontology, called “robotic_soccer”.

```
[ communicative_act(_), sender(_),
  receiver(_), protocol(_),
  ontology(_), content(_)]
```

Figure 4: Proposed FIPA message syntax

Next, we will explore three situations in which the interaction between agents is needed for coordination: *queries, requirements* and *proposals*. We will also explain how to complete the predicate `processMessages/2` of Figure 3 for handling three standard interaction protocols.

3.1 Querying for information

Here we will show how an agent may ask another one for some specific information. Since this interaction is of type (b), that is, it does not require the execution of an action by the agent that receives the message, then, it should not interrupt its perception/action loop. The FIPA interaction protocol that will be use in this case is the *FIPA Query Protocol*.

An agent may need, in its decision cycle, the information of the location of another robot in the field. Therefore, it may initiate the *FIPA Query Protocol* with the agent controlling that robot, and then, it should wait for the answer. An example of a possible message to initiate this protocol may be:

```
[ communicative_act(query), sender(Me),
  receiver(teammate1), protocol(fipa_query),
  ontology(robotic_soccer),
  content(location(teammate1,X,Y)]
```

Is important to note that the participating agent may not answer immediately, or even, it may not answer at all. Thus, the agent initiating the protocol should consider this alternative in order not to remain blocked waiting for a long time. For this, the agent may use the primitive `receive/4` included in the Logical Communication Layer that allows it to wait for a message for an specified amount of time.

This type of interaction does not require an execution of an action by the receiver, and the agent may answer it concurrently to its processing activities in its perception/action loop

In Figure 3 we have shown an outline of the implementation of this proposal, and when a message is received, the predicate `processMessages/2` is executed automatically. Now, in Figure 5, we show how to complete the implementation of `processMessages/2` in order to process in parallel an interaction of type (b) like this *FIPA query protocol*.

Observe that for each protocol of type (b) that the agent has to handle, (*i. e.*, there is no associated action required by the robot), a rule like the one in Figure 5 has to be added to the agent code.

We will explore next, two situations where the interaction is of type (a), that is, it requires the execution of a specific action by the agent that receives the requirement or the proposal. In this case, the protocol should be handled in the perception/action cycle of this agent.

```
processMessages(_,Msg):-
  member(Msg,protocol(fipa_query)),
  member(Msg,communicative_act(query)),
  member(Msg,content(location(teammate1,X,Y))),
  member(Msg,sender(Sender)),
  my_pos(X,Y),
  my_name(Me),
  send(Sender,[sender(Me), receiver(Sender),
        communicative_act(inform),
        protocol(fipa_query),
        ontology(robotic_soccer),
        content(location(Me,X,Y))]).
```

Figure 5: An example showing how to handle incoming messages of a particular query.

3.2 Requiring the execution of an action

We will consider here a situation in which an agent detects a possible pass to it when another teammate has the ball. Thus, it may send a message asking him to pass the ball and then, it should wait for the answer. The FIPA interaction protocol more suitable for this case is the *FIPA Request Protocol*. An example of a possible message initiating this kind of interaction may be:

```
[ communicative_act(request),sender(Me),
  receiver(teammate1),protocol(fipa_request),
  ontology(robotic_soccer),
  content(pass_ball(Me,X,Y)) ]
```

As in the previous type of interaction, the receiver may not answer immediately and the initiator should be aware of this situation.

As stated above, our proposal for handling this type of interaction is to queue the messages in order to be considered in the next cycle of the agent action loop. The predicate that processes the messages, shown in Figure 3, may have one or more rules for each specific interaction protocol managed, specially those that do not require the execution of an action by the robot. Thus, in order to queue the messages that do not belong to one of these interaction protocols, we need only to add this rule: `processMessages(_,Msg):-queue(Msg).`

The way this predicate queues the messages is a design choice. One alternative is to add the messages to a list, while another one is to assert the messages as facts in the knowledge base of the agent. This choice affects the way in which the agent obtains the messages in the decision cycle (see `retrieveMsgs/1` in Figure 3).

3.3 Proposing collaboration

Finally, we will show a situation in which an agent wants to synchronize with another agent for collaboration. In this case, the agent may propose to its party the execution of a specific action. As in the previous situation, this interaction may require the execution of an action by the participant, and thus, it should be processed in the decision loop. This kind of interaction will be modeled by the *FIPA Propose Protocol.*

As an example, the agent with possession of the ball may offer to pass it to a teammate. This agent may decide the execution of this action in its percepction/action loop, and thus, it should send its teammate a message proposing the synchronization of the action. An example of a message initiating this kind of interaction may be:

```
[ communicative_act(propose),sender(Me),
  receiver(teammate1),protocol(fipa_propose),
  ontology(robotic_soccer),
  content(pass_ball(teammate1,X,Y)) ]
```

The agent receiving this proposal has to process the offer in it decision cycle. Therefore the processing should be done in the same way as in the requirement explained above.

4 CONCLUSION

In this paper, we have analyzed different forms of processing in parallel agent interaction protocols for a robotic soccer domain. We have explored three different situations in which the interaction between agents plays an important role: queries, requirements, and proposals. As a design choice, our implementation tries to minimize the communication between the agents.

Our approach proposes the processing of these interactions in parallel using a set of primitives that extends logic programming providing communication among agents. This alternative reduces the overhead imposed by the processing of the protocols and also allows the agent to answer immediately.

APPENDIX A: E-LEAGUE

The E-League [2] is an initiative in which two teams of four mobile robots play soccer. The league provides common basic services to all of the participants, such as vision and communication. Each team is composed of four robots. One of the robots can act as goalkeeper, but this is not strictly necessary. There are restrictions over the size of the robots, their shape, and the components used in their construction.

The main goal is to concentrate on the development and study of Artificial Intelligence techniques such as planning, multi-agent development, communication, and learning. The league's most important feature is its simple and modular structure. There are only three basic components that must be available to obtain a functional team:

- A vision module that works as the robot's perception component,
- A communication module that allows actions to be communicated to the robots, and
- A control module that is implemented as a MAS that control the robots.

Each team may have one or more auxiliary computers in which the agents are executed. These agents communicate with the vision component in order to obtain information about what happens on the field, and send messages to the robots by means of a communication module (see Figure 6). Even though the league does not define a standard platform for the construction of the robots, it does impose restrictions over the processing and memory capacity. This allows the use of low cost robotic kits, many of which fall under these restrictions. The system we developed was implemented using Lego Mindstorms kits [1], which are within the rules of the league.

APPENDIX B: INTERACTION PRIMITIVES

We include here some details of the set of primitives we have developed and reported in [7]. 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

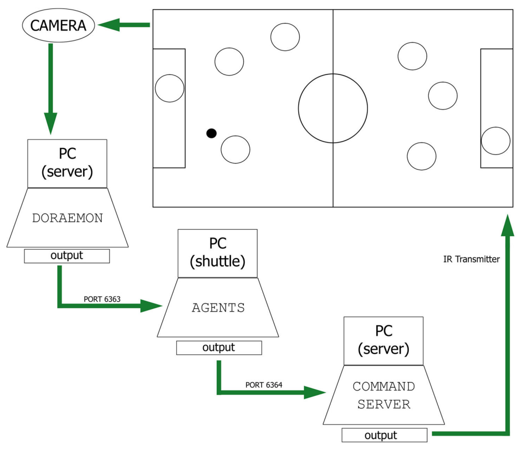

Figure 6: E-League setup (taken from [6])

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 [5] and KQML [8], and provide tools for developing standard Agent Interaction Protocols. 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/4`) 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` and `bind_all`) 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 agents.

References

[1] Lego Mindstorms robots and RCX controllers. http://www.legomindstorms.com.

[2] Official E-League webpage. http://agents.cs.columbia.edu/eleague/.

[3] N. Carriero and D. Gelernter. Coordination Languages and Their Significance. *Communications of the ACM*, 35(2):97–107, February 1992.

[4] 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.

[5] FIPA. Foundation for intelligent physical agents. http://www.fipa.org.

[6] Alejandro J. García, Gerardo I. Simari, and Telma Delladio. Designing an agent system for controlling a robotic soccer team. In *Proceedings of X Argentine Congress of Computer Science*, page 227, 2004. http://cs.uns.edu.ar/~ajg/papers/index.htm.

[7] 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.

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

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

[10] RoboCup. Lisbon, Portugal. 2004. http://www.robocup2004.pt.

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