

# Layered Reactive Planning in the IALP Team

## IALP

*Antonio Cisternino, Maria Simi*

---

---

AC, MS: Department of Computer Science, University of Pisa

**Abstract.** *The main ideas behind the implementation of the IALP RoboCup team are discussed: an agent architecture made of a hierarchy of behaviors, which can be combined to obtain different roles; a memory model which relies on the absolute positions of objects. The team is programmed using ECL, a Common Lisp implementation. The research goal that we are pursuing with IALP is twofold: (1) we want to show the flexibility and effectiveness of our agent architecture in the RoboCup domain and (2) we want to test ECL in a real time application.*

## 1 Introduction

IALP (Intelligent Agents Lisp Programmed) is a team for the simulation league of the RoboCup initiative [2]. The team is programmed using ECL, a public domain implementation of Common Lisp [1].

For the basic architecture of IALP we have adopted a reactive planning approach and developed an agent architecture where the global behavior of the planner is structured in layers. The requirements we had in mind for the architecture is that it must be open and offer different levels of abstraction coping with different problems in a modular way; moreover the architecture is meant to be general and flexible enough to allow reuse of code built for the RoboCup initiative in other domains.

For coping with limited perceptions, we have developed a memory model which relies on the absolute positions of objects, and offers a set of predicates allowing players to reason about the game at different levels of abstraction. IALP uses a model of coordination without communication [3] and a concept of role for a player that is built on top of basic abilities, common to all the agents. The layered and modular structure of the planner allows an easy reuse of the basic capabilities of the players and specialization of roles at the higher levels.

Using Common Lisp to implement IALP offers clear advantages from the AI programming point of view; in particular we have taken advantage of the Lisp reader and the macro feature. Using the ECL implementation of Common Lisp, designed for being embeddable within C based applications,

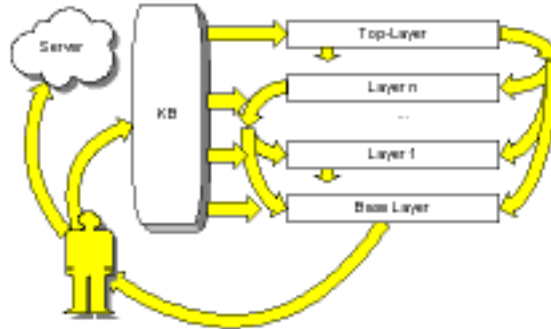


Figure 1:

we wanted to see if such language can compete with C/C++ written teams in a real time domain.

In this paper we report about the main features of the IALP team: the planner architecture and the memory model are described in sections 2 and 3; section 4 explains how the planner and the KB have been used to program the players and the coordination model used.

## 2 The architecture of the planner

The core of IALP is a hierarchically structured reactive planner that computes and executes plans. There is an ordered chain of layers, with a base layer and a top layer. The base layer is devoted to the communication with the RoboCup server: thus the output are commands like (**dash speed**) or (**turn moment**). The top layer defines the overall strategy of a player; it contains the most abstract plans and fully determines the behavior of the agent. The intermediate layers define a hierarchy of actions: each layer decides upon the implementation of an action using the actions offered by lower layers.

A plan built in a layer is a list of actions defined in one of the layers below. A *while* action can be used to repeat a sequence of actions until a specified condition is verified. At each cycle, the interpreter of plans requests an action in executable form to the base layer; if this layer is executing a plan, the next action of the plan is executed. If the layer does not have a plan (has finished executing the previous one), it requests a new plan to the upper layer. This chain of requests may propagate to the top layer, which must always return an appropriate plan. The architecture of the planner is shown in figure 1.

### Planner architecture

Each intermediate layer receives a plan from the upper layer and must execute the actions contained in it. The way an intermediate layer executes an action is by computing a particular function that takes into account a number of parameters and returns a plan to be executed by the lower layer. If the action is unknown to that layer the task of computing the plan is delegated to the layer below.

The planner as a whole implements a hierarchy of actions that are all available to the top layer to solve the task of writing a player for RoboCup in a suitable abstract language. Since the top-level planner determines the behavior of the underlying planners, specific abilities implemented by lower levels may be reused for building different roles. In particular the layered approach is convenient for sharing low level abilities that all players should possess.

Each layer can request to reset the executing plans to upper and/or lower layers. This feature is important to implement reactive behaviors and in particular to react promptly to referee messages.

Another feature of the IALP planner is the possible non-determinism in the execution of actions. It is possible to define several alternative implementations for an action, all of them considered equivalent with respect to the outcome. In this case the interpreter chooses randomly the implementation to be used. With this feature, it is quite easy to introduce a richness of behavior. The result is that it may be difficult for an opponent team to guess the behavior of players.

A language for defining plans has been developed exploiting the macro feature of LISP. This language allows defining the update function, responsible for resetting the executing plan, and the actions implementation for a layer.

### 3 The memory model

A memory model is used in IALP to record basic properties of the environment used to decide which actions should be sent to the server. The memory keeps track of objects seen recently and is responsible for computing the absolute positions of any object and of the other players. The memory also stores the messages heard and the physical status of the player.

The IALP player executes a standard cycle: receives a perception from the server, updates the memory, computes a new set of actions and sends them to the server. In deciding the next actions the planner uses higher level predicates implemented from the information contained in the memory.

If the received perception is *see* the memory tries to update the absolute position of the player. The coordinates are the same used by the server. The absolute position of the player is computed using a borderline and a flag. When a borderline is visible the player can easily compute his distance from the line, and thus one coordinate, which is x or y depending on the line and the direction in the coordinate system chosen. If a flag is also perceived the player can compute the second coordinate. This method has a good precision and is fast to compute. The basic assumption is that the player movements are continuous and if the player at a given time cannot compute one or both coordinates he can assume the previous ones, without making a significant error.

Once the position of the player has been computed, the absolute coordinates for each dynamic object present in the *see* perception (players and ball) are also computed using standard trigonometric calculus.

The choice of recording absolute coordinates (see [4] for a different choice) allows us to focus memory updating on the moving objects because static

objects are recorded in a stable form using their position.

The *hear* and *sense body* perceptions are treated similarly: they are parsed and all the information stored in appropriate structures in the memory of the agent. The *referee* messages are stored separately from other messages since they contain the status of the game and it is necessary to make sure that they are acted upon.

Given this memory model based on absolute positions of objects, we have defined functions and predicates and derived more abstract properties of the environment useful for defining player behaviors; an example is a function for computing the distance between two objects used by the `goto-x-y` action.

## 4 The implementation of IALP

Using ECL has been our bet. RoboCup is a real-time domain task where system level languages like C/C++ seem to be much more effective than traditional AI languages like LISP or PROLOG. On the other hand, LISP provides advantages: no need for a parser of the messages sent by the server, automatic garbage collection, macros, closures and other high level language features traditional in AI programming were all available so that we were able to concentrate on higher level programming tasks. Preliminary experiments have shown that LISP processes are capable of maintaining the synchronization among server and clients.

IALP is built on top of the architecture described in previous sections: we have implemented the functions and predicates required in the RoboCup domain and defined a number of layers describing the capabilities of the different players.

Since most of the abilities are common to all the agents, players in different roles tend to have a great number of shared layers. Right now, they share all the layers but the topmost. More specifically, three layers are shared by all players: the *basic* layer, bottom layer of the planner whose output are actions sent to the server, the *player-base*, defining a first level of abstraction, with actions like `go-ball`, and the *individual* layer defining the individual behavior of a player. Moreover each player has a different *top-layer* situated above the *individual* layer, that distinguishes the behavior according to the role strategy. The goalie has also a *goalie-base* layer, situated between the *individual* layer and the goalie's top layer, to implement capabilities that are specific of this role.

This homogeneity among players is justified by the definition of role that we have assumed: a role is a *prevalence of a behavior*. This implies that the basic capabilities of the various players are the same, and only the overall strategy of the team and the environment account for differences in behavior. When all the team members are forced in a situation of defense, for example, we would like the attackers be able to behave like defenders.

The overall strategy of the team emerges from role definitions. A role is substantially defined by the zone of the field assigned to a player when he is not engaged in the current action. The player is responsible for the ball and opponents in his zone.

The ball flows from the defense zone to the attack zone as a consequence of the decision function used by each player. When the player has the ball, he checks whether he can pass the ball or shoot into the enemy goal; if not, he tries to move forward with the ball until a pass becomes possible or he can shoot. For deciding whether to pass the ball or proceed, each player, depending on his role, has a number for each team mate, used for assigning a preference to the candidates for a pass. Thus defenders prefer to pass the ball to middle players and are not happy to pass the ball to the goalie. The evaluation function also considers, for each possible target of the pass, the *gain* in case of success and the *risk* that the pass will be intercepted. The most promising target is thus chosen and its value compared with the gain and risk of advancing with the ball.

### References

- [1] G. Attardi, The Embeddable Common Lisp, ACM Lisp Pointers, 8(1), 30-41, 1995.
- [2] Franklin, S., "Coordination without Communication", <http://www.msci.memphis.edu/franklin/coord.html>
- [3] Kitano, H., Asada, M., Osawa, E., Noda, I., Kuniyoshi, Y., Matsubara, H., "RoboCup: A Challenge Problem for AI", AI Magazine, Vol. 18, No. 1, 1997.