Layered Reactive Planning in the IALP Team

Antonio Cisternino[1], Maria Simi[1]

[1] Dipartimento di Informatica, Università di Pisa
Corso Italia 40, 56125 Pisa, Italy
{cisterni, simi}@di.unipi.it

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 of the absolute positions of objects. The team is programmed using ECL, a Common Lisp implementation designed for being embeddable within C based applications. 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 [1, 2, 3]. The team is programmed using ECL, a public domain implementation of Common Lisp [4].
RoboCup is a real time domain task where players receive perceptions from the server and have to react within the allowed time. To make things more realistic, the environment is inaccessible (perceptions are restricted to the point of view of the player and are limited by the distance) and non deterministic (the effect of actions is not completely predictable).
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.
The layered approach used in IALP has been inspired by agent architectures for robots, as proposed for example in [5, 6] and, in the context of multi agent applications such as RoboCup, by the idea that complex behaviors can be learned in layers of increasing complexity [7]. In our approach no learning is involved, but the complexity of behavior of the planner is obtained by defining suitable actions for each layer, by means of a language oriented to action definition built on top of Lisp.
For coping with limited perceptions, we have developed a memory model that 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.

We have implemented IALP using this memory model and the planner architecture. IALP uses a model of coordination without communication [8] 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 exploited the Lisp reader and the macro feature. Using the ECL implementation of Common Lisp, designed for being embeddable within C based applications, we wanted to see if such language can compete with C/C++ written teams in a real time domain such as RoboCup.
In this paper we report about the main features of the IALP team. Section 2 and 3 describe the planner architecture and the declarative language used for defining the behavior of layers; section 4 is an account of the memory model; section 5 explains how the planner and the KB have been used to program the players and the coordination model used.

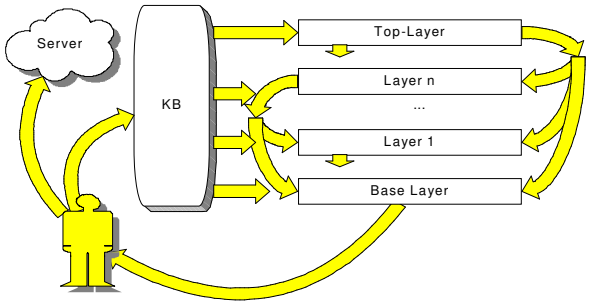


Fig. 1: The planner architecture

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 outputs 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.
The overall architecture of the planner is shown in figure 1.
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. Actions are expressed using the classic functional notation of LISP. A simple example of plan is the following:

```
((dash 100) (turn 90) (dash 100))
```

Another example, involving the **while** iteration construct is the following:

```
((*while* (not (can-kick? Kb))((go-ball)))
 (kick! (enemy-goal kb)))
```

The plan executor sends to the inferior layer the request to repeatedly execute the action `(go-ball)` until the condition `(can-kick? kb)` is verified; at this point the action `(kick! (enemy-goal kb))` will be executed.
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.
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 inferior layer.
The execution of an action by a given layer may simply return a standard plan, good for any situation (in this case the system is an interpreter of plans) or involve a computation of a plan taking into account the knowledge contained in the KB. The top layer must necessarily compute a plan. Thus 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 suitable 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; for example, in order 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. An advantage is that it may be difficult for an opponent team to guess the behavior of players.
The planner executes a standard perception/action cycle but we have extended the base layer to allow it to return a list of basic actions, so that all the available slots for executing actions can be exploited. Thus the basic cycle of the planner is “read a perception from the server, compute the next sequence of actions and execute them”.

3 The language for defining the behavior of the planner

We have developed a simple declarative language based on LISP to define the behavior of the various layers.
Each layer contains the definition of an *update function* that, testing some conditions on the current environment, decides if some of the executing plans must be terminated. The function can return four different values: `nil`, `UP`, `DOWN` and `ALL`. When the value returned is `nil` the planner can continue its execution. `UP` means that the upper layers must abort the execution of the current plans; this capability is useful when the changes in the environment affect only the more abstract plans while lower layers can continue executing their tasks. `DOWN` is the dual of `UP` and aborts the executing plans of inferior layers; in this case it is deemed useful to continue with the overall strategy but some change in the environment make it necessary a re-planning at the lower layers. `ALL` is equivalent to returning both `UP` and `DOWN` and forces the planner to rebuild entirely his plans.
The way to define an update function for a layer is as follows:

```
(defupdate layer
  "Optional documentation"
  body)
```

where *body* is the body of the function. In order to define an empty update function, that is an update function that always returns `nil`:

```
(def-empty-update layer)
```

The possibility of aborting executing plans instantaneously is important in real time domains such as RoboCup where the environment is highly dynamic. An example is the *referee* message that changes the state of the game: each player must suddenly change his behavior to adjust to the new state. The update method that deals with *referee* messages is located in the base layer and has the following definition:

```
(defupdate basic-layer
  "Handles referee messages"
  (if (and
        (eql
          (last-percept-type kb)
          'REFEREE)
            (not (last-message-read kb)))
      (progn
             (message-read kb)
           'UP)
    nil))
```

The update function aborts all executing plans if the last perception is of type *referee* message. In this case, because the base layer is the bottommost, `UP` is equivalent to `ALL`.
In addition to the update function, a layer must define a set of actions. For each action at least one implementation must be provided. If there are multiple implementations of a given action the interpreter chooses among them with a given *policy*. For the moment the *policy* consists in choosing randomly among the implementations. In our language the list of actions defined in a layer is specified by means of the following construct:

```
(defactions layer
 (action-list action-name)
 (action-list action-name imp1 ... impn)
  ...)
```

If the `action-list` statement is followed only by the `action-name` an implementation is assumed with the same name of the action. If imp_1 `...` imp_n are specified, the action definitions with these names are associated to *action-name*.
A definition of an action is similar to the definition of a function but uses the `defaction` keyword:

```
(defaction name (params)
  "Documentation."
  body)
```

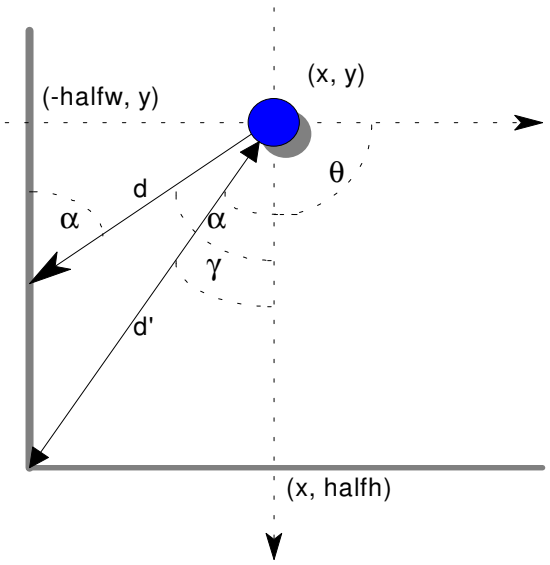
The `name` of the action must be one of those declared within the `defactions` construct. The parameter list allows passing some parameter to the action, for example the `go-ball` action must receive as a parameter the speed that must be used. The `defaction` must return a plan.
As an example of action definition we include a possible implementation for the `run-with-ball` action:

```
(defaction run-with-ball (speed dir k)
  "The power of kick is speed * k."
  (if (can-kick? kb)
      `((turn ,(ball-dir kb))
            (kick ,(* k speed) ,dir)
            (turn ,dir)
            (dash ,speed))
    '((sleep))))
```

This action checks if the player can kick (the ball is close enough) and, when this is the case, returns a plan that prescribes: “turn towards the ball, kick in the direction requested, turn and dash”. If the player cannot kick, the plan `((sleep))` is returned and the player does not do anything because the action requested cannot be executed”. This plan causes the immediate termination of the action and the request for a new action.

4 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 of a



$$\theta = \frac{\pi}{2} + \alpha$$

$$x = -halfw + d \cdot \sin\alpha$$

$$\frac{d \cdot \sin\alpha}{d'} = \sin\gamma \Rightarrow \gamma = \sin^{-1}\left(\frac{d \cdot \sin\alpha}{d'}\right)$$

$$y = halfh - d' \cdot \cos\gamma$$

Fig. 2: Absolute position

player, or KB, 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.
When the perception is received it is parsed using the `read-from-string` function provided by LISP. This is very convenient because the perceptions sent by the server are strings containing S-expressions and the LISP reader knows how to deal with them.

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 (0, 0) position corresponds to the center of the field, the *x* direction is towards the enemy goal and the *y* direction is on the right of a player looking at the opponent's goal.
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. Figure 2 shows how the absolute coordinates of the player are computed from the perception of a line (i.e. its distance *d* and its relative direction α) and a corner flag (i.e. its distance *d'* and its relative direction γ).
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.
Differently from what has been proposed by other researchers [9] we have decided to maintain absolute positions for the following reasons. The absolute positions kept for all moving objects can be exploited when an object is not in the current *see* perception. For example if the player sees the ball at simulation cycle *t* and another player covers the ball at time $t + k$, we can assume that the ball is near to the last position recorded into memory. This assumption is reasonable only if the elapsed time *k* is reasonably small. Moreover, if the player changes its direction, the information stored in the memory is not affected and it is not necessary to update object positions. Also the distance among objects can be easily computed from their absolute positions.
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, we have defined functions and predicates and derived more abstract properties of the environment useful for defining player behaviors. Some of these predicates are used to make qualitative statements about the environment: for example the predicate `can-kick?` is true when the ball is near enough to the player, which in terms of lower levels means within 2 meters.
Two very important functions are `distance` and `dir-x-y`. The function `distance` computes the distance between the player and another point (*x*, *y*) and is fundamental for evaluating distances from objects during the game. This function is used instead of the relative distance provided in the perceptions, because, by referring to its memory, an agent is able to estimate the distance of an object even when the object is not currently perceived. This function is also exploited to evaluate the distance from a given point situated in a zone of the pitch; this is useful to implement a zone-based strategy.
The function `dir-x-y` allows a player to know the moment of which to turn to see a point (*x*, *y*). This function also exploits the fact that the player can estimate

the absolute coordinates of every object. Together with the `distance` function, `dir-x-y` is very useful to implement a `goto-x-y` action.
As an example of the flexibility of our memory model we show the implementation of the *outside* predicate:

```
(defun outside? (kb)
 (dolist p (enemies kb)
   (when (and
          (not (is-goalie? p))
          (> (pos-x kb)
             (obj-info-x p kb)))
      (return nil)))
  T))
```

5 The implementation of IALP

The Embeddable Common Lisp is an implementation of Common Lisp designed for being embeddable within C based applications [4]. ECL uses standard C calling conventions for Lisp compiled functions, which allows C programs to easily call Lisp functions and vice versa. No foreign function interface is required: data can be exchanged between C and Lisp with no need for conversion. ECL is based on a Common Runtime Support (CRS) which provides basic facilities for memory management, dynamic loading and dumping of binary images, support for multiple threads of execution. The CRS is built into a library that can be linked with the code of the application. ECL is modular: main modules are the program development tools (top level, debugger, trace, stepper), the compiler, and CLOS. A native implementation of CLOS is available in ECL: one can configure ECL with or without CLOS. A runtime version of ECL can be built with just the modules required by the application.
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 a lot of advantages: no need for a parser of the messages sent by the server, automatic garbage collection, macros and closures and other high level language features traditional in AI programming were all available, so that we were able to concentrate on high level programming tasks since the beginning. Preliminary experiments have shown that LISP processes, implementing IALP players, are capable of maintaining the synchronization between 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. So far we have defined a preliminary hierarchy of layers that we intend to evolve and adjust by gaining more feedback from actual matches. Figure 3 shows the structure of the layers for the team members of the current IALP implementation.
Since most of the abilities are common to all the agents, players in different roles tend to have a great number of shared layers. In fact right now they share all the

layers but the topmost. The goalie has an additional layer to implement capabilities that are specific of this role.
This homogeneity among players is justified by the definition of role that we have

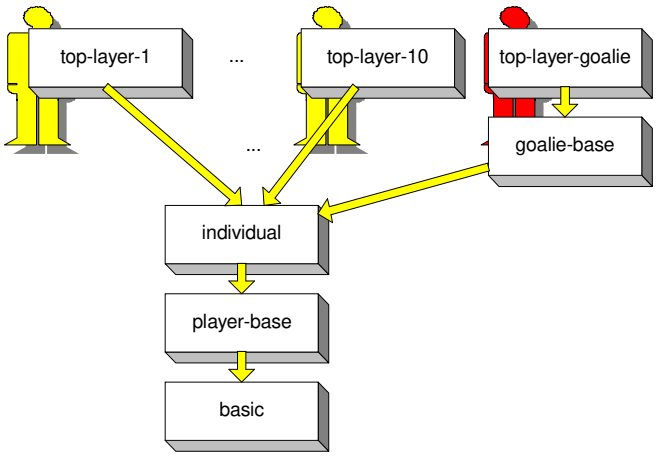


Fig. 3: Layers used in IALP

assumed: a role is *a prevalence of a behavior*. This definition of role implies that the basic capabilities of the various players must be the same and only the overall strategy of the team and the environment determine the effective behavior of a player. To understand why we have chosen this definition, consider a situation where a defender finds himself in an attack position for some reason: we want the defender to behave like an attacker for the period that he is involved in the action. An analogous situation is when all the team is forced in a situation of defence and the attackers must behave like defenders. In the real soccer it is impossible to find players able to perform top level in any role; players usually specialize in a set of tasks. Simulated soccer is different also in that it is not a problem to replicate abilities: why shouldn't we give to all the players the best capabilities in kicking and controlling the ball that we were able to develop? The only exception is the goalie that must have capabilities of his own inapplicable to the other players.
The *basic* layer is the bottom layer of the planner and its outputs are actions that are sent to the server. The actions defined at this level are basic actions such us `turn`, `dash`, `kick`, `say`, `catch`, `move`, and low level actions such as `sleep`, `ndash` and `turn-ball`. The `sleep` action is a no-action command telling the interpreter that no commands must be sent to the server until a new perception is received. The `ndash` action tells the interpreter that *n* `dash` commands must be sent to the server in sequence, without waiting for a new perception. The `turn-ball` sends a sequence of actions to the server to turn the player and the ball of a given angle.
A little preprocessing of the arguments may be done also for the basic actions: for example a turn action with a moment less than 1 is not sent to the server because it is not relevant.

The *player-base* defines a first layer of abstraction. A subset of the actions defined in this layer is the following: `go-ball`, `see-ball`, `pass-ball`, `run-with-ball` and `goto`. The `go-ball` action causes the player to find and reach the ball. The `see-ball` action makes the player turn until he is able to see the ball. The `pass-ball` action passes the ball to a given player; in this case the assumption is made that an upper layer has checked that the player can receive the ball. Finally the `run-with-ball` allows the player to run with the ball using the `kick`, `dash` and `turn` actions required to produce this complex behavior.
The *individual* layer defines individual behaviors of a player like `stay-in-zone` that situates the player in a given zone of the field. Another action is `handle-with-ball` that manages the ball and tries to move the player with the ball towards the enemy goal. The `free-kick` action is devoted to the execution of a free kick.
The three layers described above are shared by all the players because they correspond to abilities that all players should possess. Each member of the team has its specific top layer that distinguishes the behavior according to the role strategy. The top layer code for all players is similar and changes only in those aspects accounting for the prevalence of behavior.
The bigger differences between the goalie and other players are reflected in an additional layer, the *goalie-base*. This layer defines actions like `free-kick`, used to follow a far away action or `catch-ball` used to catch the ball.
The model of coordination used to pass the ball does not involve communication. The player possessing the ball evaluates the possible candidates for a pass and the risk of loosing the ball; if it decides to make a pass to a certain player he does so. The coordination is in the fact that a player near the action is typically interested in the ball and thus able to recognize the pass.
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. When the player has the ball he checks whether he can pass the ball or shoot into the enemy goal; if he can't, he tries to move in the direction of the opponent's goal until a pass becomes possible or he can shoot.
The flow of the ball from the defense zone to the attack zone is a consequence of the decision function used by the player to establish whether to pass the ball or proceed. 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.

6 Conclusions and future work

In this paper we have described the basic ideas behind the implementation of the IALP team. We have adopted an agent architecture based on a layered reactive

planner. Experience gained in past competitions showed that the low level (the communication layer responsible for handling communication with the server) was too slow; moreover the memory model, implemented in LISP, proved to be too heavy. This suggested rewriting in C both the communication level and the memory model for better performance.
For the future we want to experiment with different arrangements and implementations of layers and different means of coordination, besides testing the validity of the definition of role as a prevalence of behavior.
We are also interested in investigating an emotional approach to define the behavior of players [10]. Finally we want to observe the emerging behavior due to the introduction of non-determinism.

Acknowledgments

We want to thank Silvia Coradeschi for getting us interested in RoboCup and all the students of the course "Artificial Intelligence: Laboratory" in Pisa, during the a.y. 1997/98 and 1998/99, for taking up with so much enthusiasm the RoboCup challenge and for allowing us to learn and gain experience during the IAL-Cup 99.

References

1. Kitano, H. and Asada, M. and Kuniyoshi, Y. and Noda, I. and Osawa, E., "RoboCup: The Robot World Cup Initiative", IJCAI-95 Workshop on Entertainment and AI/Alife, 1995
2. Kitano, H., Asada, M., Osawa, E., Noda, I., Kuniyoshi, Y., Matsubara, H., "RoboCup: The Robot World Cup Initiative", Proc. of the First International Conference on Autonomous Agent (Agent-97), 1997.
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.
4. G. Attardi, The Embeddable Common Lisp, ACM Lisp Pointers, 8(1), 30-41, 1995.
5. Brooks, R. A., "A Robust Layered Control System for a Mobile Robot, *IEEE Journal of Robotics and Automation,* RA-2(1), March 1986.
6. Firby, R. J., "Task Networks for Controlling Continuous Processes", *Proceedings of the Second International Conference on AI Planning Systems*, Chicago IL, June 1994.
7. Stone, P., Veloso, M., "A Layered Approach to Learning Client Behaviors in the RoboCup Soccer Server", in *Applied Artificial Intelligence,* 12, 1998.
8. Franklin, S., "Coordination without Communication", http://www.msci.memphis.edu/~franklin/ coord.html
9. Bowling, M., Stone, P., Veloso, M., "Predictive Memory for an Inaccessible Environment", In *Proceedings of the IROS-96 Workshop on RoboCup*", November 1996.
10. Franklin, S., McCauley, T. L., "An Architecture for Emotion", AAAI 1998 Fall Symposium "Emotional and Intelligent: The Tangled Knot of Cognition".