Instituto de Sistemas e Robótica

Pólo de Lisboa

Agent-Based Programming Language for Multi-Agent Teams

Rodrigo Ventura   Pedro Aparício   Pedro Lima

March 1999

RT-701-99, RT-401-99

ISR-Torre Norte
Av. Rovisco Pais
1096 Lisboa CODEX
PORTUGAL

Este trabalho foi financiado pela Fundação Calouste Gulbenkian - Serviço de Ciência.

Abstract

This report specifies a programming language for multi-agent teams. The language aims at providing an abstract level approach to the programming of teams composed of either software or hardware agents (e.g., robots), encapsulating the lower level implementation details (e.g., graphical primitives for icon animation, robot motion primitives) and providing an abstraction level appropriate for multi-agent systems.
The overall architecture of the multi-agent team, the language specifications and an example of application to robotic soccer are included.

Keywords: Cooperative Robotics, Multi-Agent Systems, Distributed Artificial Intelligence, Robotic Soccer

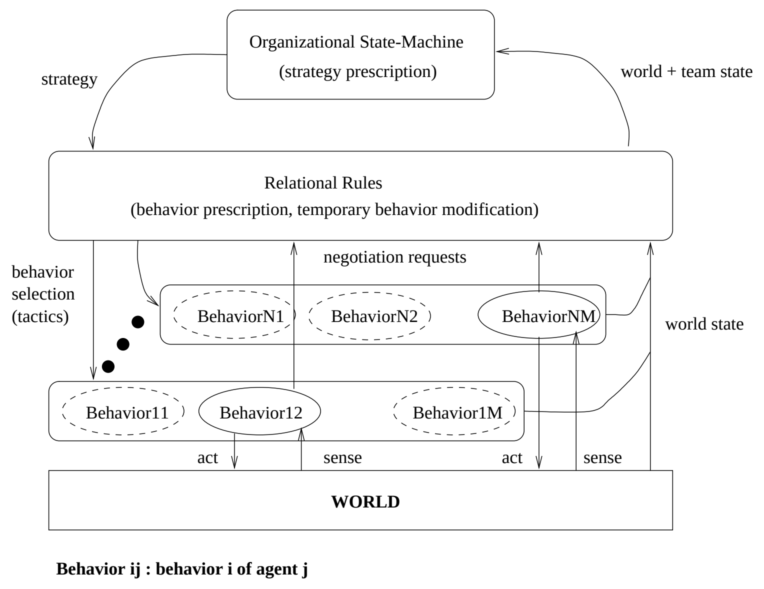

Figure 1: The functional architecture of the ISocRob team.

1 Purpose

The purpose of this report is to specify a programming language for multi-agent teams. The language aims at providing an abstract level approach to the programming of teams composed of either software or hardware agents (e.g., robots), encapsulating the lower level implementation details (e.g., graphical primitives for icon animation, robot motion primitives) and providing an abstraction level appropriate for multi-agent systems.

The language specifications intend to be comprehensive, but they are inspired by hardware agents, namely soccer robot teams.

We begin by defining in Section 2 the main concepts involved in the architecture underlying the language specifications, such as agent behavior, agent state, functional levels, world model. Section 3 provides the detailed specification of the language, including the way the above concepts are actually implemented. Finally, in Section 4 an application to soccer robots illustrates the main concepts and their implementation using the language.

2 Architecture and Main Concepts

The team architecture is depicted in Figure 1, and its mapping onto the actual hardware of the ISocRob team is shown in Figure 2. It is based on a 3-level functional hierarchy [1]:

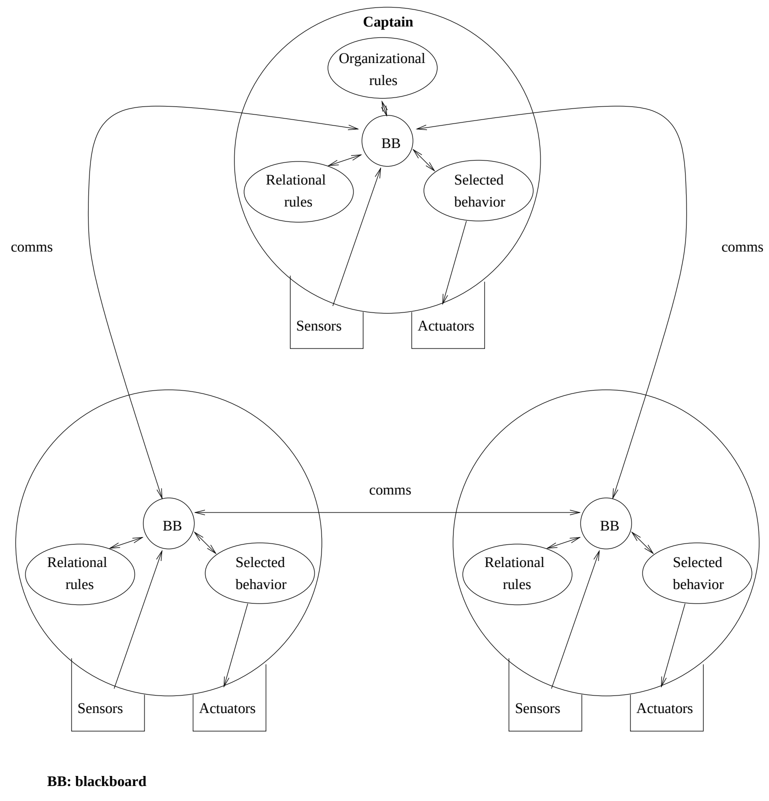

Figure 2: The mapping between the functional architecture and the actual hardware of the ISocRob team (3 homogeneous robots – from the hardware standpoint).

- **Organizational**: establishes the strategy to be followed by the whole team, given the world and the team state (e.g., game plus team state in robotic soccer).

  In the soccer team context, game states can be grouped in two main categories:

  1. game situations reached upon the application of RoboCup tournament rules (e.g., kickoff, end-of-game, penalty-for, penalty-against);
  2. evaluation of current game status, as perceived by the team members (e.g., symbolic, such as losing & close to the end of the game, ball close to our goal, or numerical evaluations);

  Strategies can be divided in, at least, two major categories:

  - pre-programmed scenarios for game situations in game state category 1 above;
  - dynamic strategies (e.g., defend, attack, counter-attack), corresponding to game state category 2 above.

- **Relational**: to accomplish useful cooperation, relationships between agents must exist. This involves an important characteristic of the agent concept: social ability, meaning that one given agent has to be aware of the existence of other agents like him, with whom it has to negotiate. At this level, groups of agents cooperate and eventually come to an agreement about some objective (common or not). Also, based on the strategy prescribed by the organizational level, behaviors are distributed by the different agents.

  For instance, in the robot soccer context, if one robot is willing to pass a ball to one of its teammates, it must find one of them available and sufficiently well positioned. The ball pass is arranged via a negotiation process, and after an agreement is reached, including the knowledge of relative positions and orientations, the pass is performed. Another example is the situation where two teammates actively try to get the ball. In such a case, one of them should signal the other its intention. A negotiation process would follow, where the teammates would determine their distances to the ball to decide which one should pursue it.

  Negotiation among agents at the relational level may result in temporary modifications to the current strategy established by the organizational level (e.g., a player acting as a Forward might temporarily refrain from doing so, in order not to conflict with one of its teammates while pursuing the ball).

- **Individual**: encompasses all the available *behaviors* of each agent. Those include the primitive tasks (e.g., path planning, motion with collision avoidance) and their relations.

A *behavior* corresponds to a set of purposive (i.e., with a goal) primitive tasks sequentially and/or concurrently executed. The primitive tasks consist of *sense-think-act loops* (STA loops), a generalization of a closed loop control system which may include motor, ball tracking or trajectory following control loops.

STA loops are composed of the following key components:

- **goal**: the objective to be accomplished by the primitive task (e.g., a position set-point, an object to be found in the image, localization with respect to an object in the image);

- **sense**: sensor data required to accomplish the goal (e.g., image, infra-red data);

- **think**: the actual algorithm which, using the sensor data, does what is required to accomplish the goal (e.g., motion controller, ball visual servoing);

- **act**: the actions associated with the **think** algorithm (e.g., move the wheel motors).

Notice that some of the components might not be used in some cases (e.g., a path planning primitive task does not need sensors and actuators).

The sequence of primitive tasks is traversed as the logical conditions associated with the connections between them become true. The logical conditions are defined over a predicate set. There are two *predicate* classes:

- predicates which check the value of a given variable (e.g., the variable *goal* in lastseen-left(goal));
- predicates which check the past occurrence of a given event (e.g., the see(ball) predicate checks whether the ball became visible). It is assumed that the opposite event must occur in order to modify the predicate's truth value.

A *world model* is required to provide the necessary information to the organizational level. Since all computation is supposed to be distributed by the team members, with no external storage available, a distributed world model representation is needed, containing all the relevant variables for negotiation between agents, and in general the result of processing raw data, for primitive tasks usage.

3 Language Specification

3.1 Language Requirements

The language should provide the means to let the team strategist (e.g., the coach, in robotic soccer) program the population in order to achieve the strategic objectives. Those are implicit in the selected behaviors, for each behavior and in the primitive task STA loops.

We propose the following implementation for each of the above concepts:

- the *strategy* is described at the organizational level by a state-machine whose transitions are traversed upon the matching of specific world states, and whose states prescribe the current strategy;
- the *tactics,* including behavior selection, negotiation, and temporary behaviors modification, is implemented by production rules at the relational level;
- a *behavior* consists of a state-machine, where each state corresponds to an *STA loop* and each transition has associated logical conditions defined over the predicate set described in Section 2;

The organizational state machine must run in one of the team members, designated as the *captain.* Whenever the captain does not signal that it is live for more than a timeout period, a new captain should take control of the team.

A *blackboard* implements global shared memory and event[1]-based communication.

[1]Event is here interpreted in the context of a computational model.

3.2 Computational Model

The computational model for the language consists in two classes of objects: *agents,* and *blackboards.* A blackboard forms the only communication medium among agents, either to communicate among themselves, or between them and the external world. This follows the RUBA [4] paradigm of a multi-agent system (MAS). However, the current language specifications propose several improvements over RUBA, such as the extension of the blackboard for a distributed system, efficient blackboard indexing using a hierarchical name-space, and event-driven programming. The goal of these improvements is to tailor the language to a robotic application.

3.3 Distributed Blackboard

This is a conceptually centralized repository of data, but is distributed among the agents. The blackboard is a mapping of symbols hierarchically organized in nested name-spaces, e.g. `robot0.sensors.collision.2`, to variables. This scheme is supposed to abstract a myriad of possible semantics, such as message passing, shared memory, distributed data, local variables, and so on. A blackboard is implemented with an hash table of names to variables. Each variable has the following attributes:

`scope` : global vs local, *i.e.,* whenever its creation is reflected in all blackboard members, or not;

`location` : remote vs local, *i.e.,* whether it is an index to a blackboard in another machine, or it is local to this machine;

`policy` : broadcast vs indexed, *i.e.,* whether variable updates are broadcast to all blackboards or not;

`type` : static vs hook vs closure, *i.e.,* whether it is a variable, a event hook triggered when the variable is changed, or a closure that is called whenever that variable is read. This closure is called in a defined agent context;

`lock` : If the variable is locked, it cannot be modified, until unlocked. This feature must be used carefully to avoid deadlock conditions;

The access semantics for each variable is supported by the primitives:

`read` : reads the value of the variable;

`write` : sets the variable to a value;

`hook` : adds a closure to be called whenever the variable changes — which will be called *event* here and henceforth, under the context of the computational model (not to confuse with events mentioned in the previous section).

`lambda` : defines a closure to be called in order to obtain the variable value, in the context of the defining agent;

3.4 Agent Programming

The agent programming is based on three elements: *rules, states,* and *events.* Each agent has a private set of variables, which usually start with a `/` for clarity. The syntax of the language is based on LISP. Each one of these elements are defined by clauses. All RUBA keywords start with a : sign. The syntax of a rule is

```
(:if expression
 :then clauses1 :go-state state-spec1
 :else clauses2 :else-state state-spec2)
```

being interpreted as usual — if *expression* returns true, *clauses1* is evaluated while *state-spec1* specifies a state transition. Otherwise, *clauses2* and *state-spec2* are taken into account. Any one of these keywords are optional. If *expression* is omitted, it is as if it was true. A state transition specification has the syntax:

```
state | (variable state)
```

The latter form specifies what variable is supposed to hold the state, while the former uses a default variable name (say, variable `/state`).

The state clauses have the syntax:

```
(:on-state state-cond [clause]*)
```

which simply takes into account the contained clauses when the state satisfies the *state-cond* condition. This condition has the following syntax:

```
([[:varvariable] [state]*:not[state]*]+)
```

The event concept is new to RUBA, and corresponds to the interactive nature of robotic applications, which *sense-think-act* loops are a much more natural approach than the classical functional recursive decomposition paradigm [3, 2]. The syntax of an event clause is

```
(:on-event event-spec :do clauses1 :scope clauses2)
```

where the contained clauses are taken into account only when an event satisfying the *event-spec* specification occurs. Its syntax is:

```
(type [variable] [clause]*)
```

meaning that when an event of kind TYPE arrives, it is stored in *variable,* and the contained clauses (*clauses1*) are taken into account. The scope of the link between the event and the clauses is limited to *clauses2.*

The execution model is different from that in RUBA, in the sense that the rules are only scanned once, when the agent is created. These clauses may trigger state changes, which makes the agent scan other clauses. This can be understood as an event based model, which is an implicit loop model.

3.5 Low-level Integration

In a real-robot context, the issue of integrating high-level programming and the hardware interface arises. One possible form of doing so consists of using an FFI (foreign function interface). However, there is a cleaner alternative, which avoids the cross function calling overhead — a low-level access to the blackboard. From the MAS point of view, the low-level interface looks just like any other agent(s) transacting information with the blackboard, reading and writing from/to variables in the blackboard. For instance, a C library that provides this interface must exist.

3.6 How to Implement SocRob Functional Architecture

This section describes how the above language can be used to implement the architecture of a robotic soccer team. Each robot has a blackboard and an agent, with the exception of the team captain which will have an additional agent to implement the organizational state machine.

- *Organizational state machine* — this state machine determines the global strategy of the team, stating it in the blackboard. Whenever the strategy changes, the corresponding variable is changed. This change is controlled by the rules, based on changes in certain blackboard variables, relevant to the case. Each robot will have a dormant captain agent, except for one. This allows fault-tolerant behavior: whenever it stops functioning, another robots takes over as captain;

- *Relational rules* — these rules support the negotiation processes in order to cope with all conflicting situations. A list of these situations have to be specified. The negotiation paradigm follows two phases: a statement of interest state, where each agent states its point of view, "no strings attached," and a second phase, where the issue is settled in an unambiguous fashion to all;

- *Behaviors (state machines)* — the behaviors are defined as state machines and rules in each robot agent. Each state means something in terms of sensor-actuators mapping, and a transition is triggered when a specified condition is reached;

- *STA loops* — These loops implement the sensor-actuator loops. Speed is essential here, therefore only low-level programming languages are allowed. In a first phase of the project, these STA's can be monolithic blocks of code. In the future, it is our desire to separate them in smaller inter-linked processes, with small interaction latency. The way these processes relate to each other is previously defined in a declarative language. The switching between STA loops is controlled by blackboard variables;

- *Communication between agents for negotiation* — The blackboard is the sole medium of communication between the agents, supporting the message exchange needed to perform negotiation. This negotiation has to be designed in such a way that it take a minimal number of steps. There is a first phase to assert each agent position, and a second phase to settle the result;

- *Handling the user interface* — In order to monitor the team status, an additional agent can be connected to the blackboard network, from the outside. This agent would have access to all or some of the blackboard variables. Additionally, each agent would add monitoring variables to the blackboard, allowing external agents to sneak into the robot status;

4 Application to a Soccer Robot Team

4.1 The Field Division

Figure 3 presents a functional division of the field in several regions. These are zones where robots try to locate themselves inside the field, according to their assigned behaviors, e.g., defenders should stay inside the *D* zone and Forward players should stay inside the *F* zone. This division helps the assignment of influence areas to players.

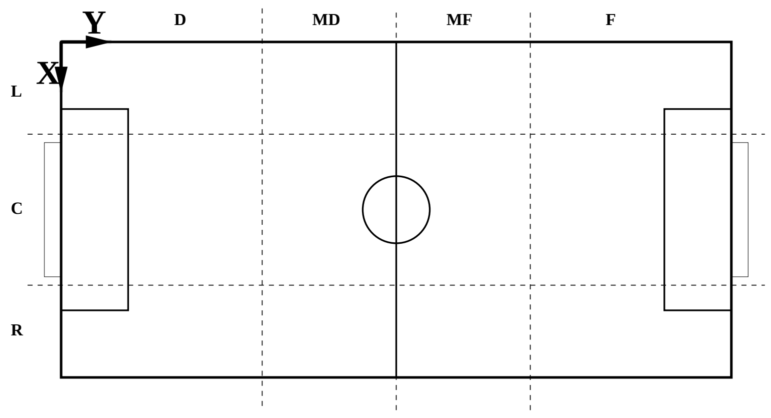

Figure 3: The field division in actuation areas.

Besides Defense (*D)*, MidDefense (*MD)*, MidForward (*MF)* and Forward (*F)*, further divisions are introduced to increase the field resolution. Along the field longitudinal axes, the field is divided in Left (*L*), Center (*C*) and Right (*R*) parts. This division is particularly useful when the team has more than one player acting in the same functional area (e.g., two defenders).

4.2 Player Behaviors

Several behaviors must be implemented in the ISocRob soccer team. These include the following:

- GoalKeeper – Defends the goal. To do that, it continuously looks for the ball and, if necessary, leaves the goal area and kicks it away. The influence zone is defined by the goal area lines and is shown in Figure 4.

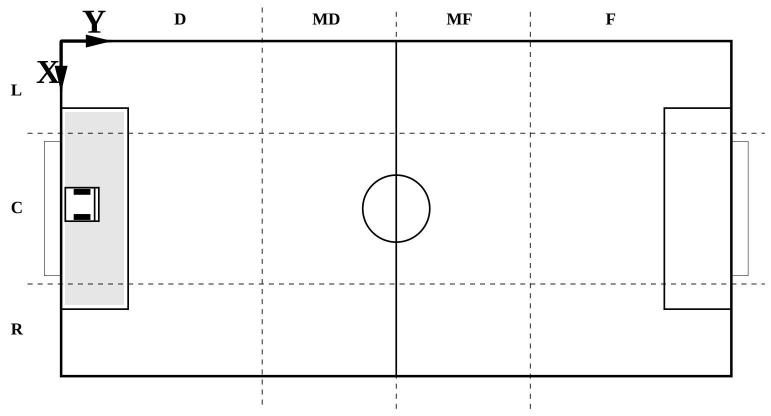

Figure 4: Goal-Keeper influence area.

- Defender – The defender mission is to move the ball from the vicinity of our goal to the opponents field. If possible, it should try to move the ball to the vicinity of the MidField or MidForward player. The idea is to return to its original position (*D*) when the ball is once again in the opponents area (see Figure 5).

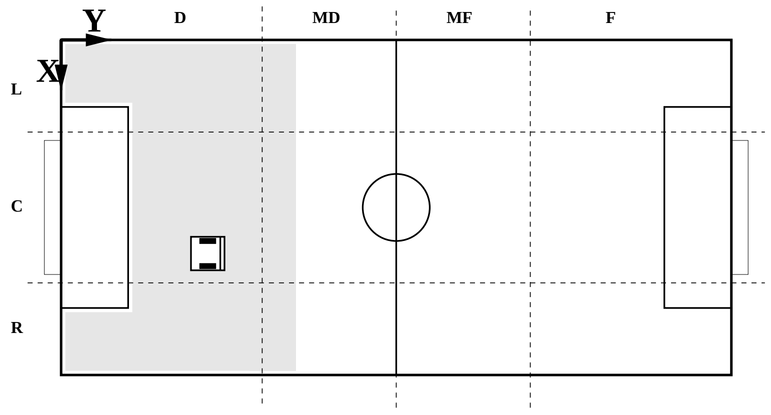

Figure 5: Defender actuation area.

- MidField – Like in real soccer, the Midfield position player is able to play in a variety of positions. Its influence zone lies within the *MD* and *MF* areas. This player natural ability is to receive the ball from its own team field and decide what to do, based on the other players availability. If a Forward is in the near vicinity of the opponents goal (*F* area), the MF should try to pass the ball to it.

- MidForward – This player moves between our field and the opponents field. Although its natural influence area is *MF*, it is allowed to follow the ball into our

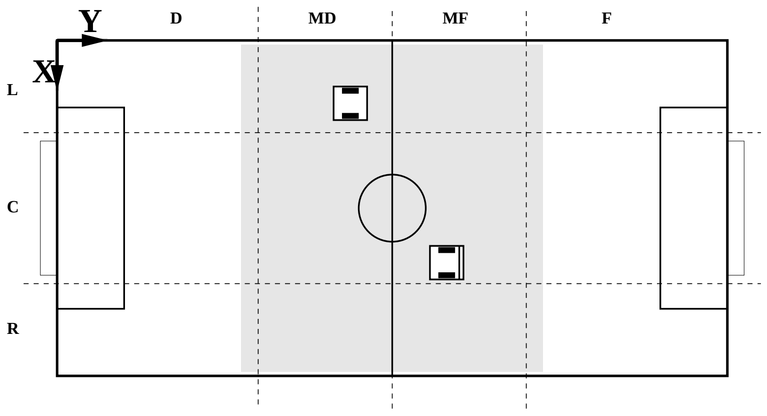

Figure 6: MidField players influence zone.

field (see Figure 7). If a Forward player is in a good position, is should try to pass the ball to it (if possible).

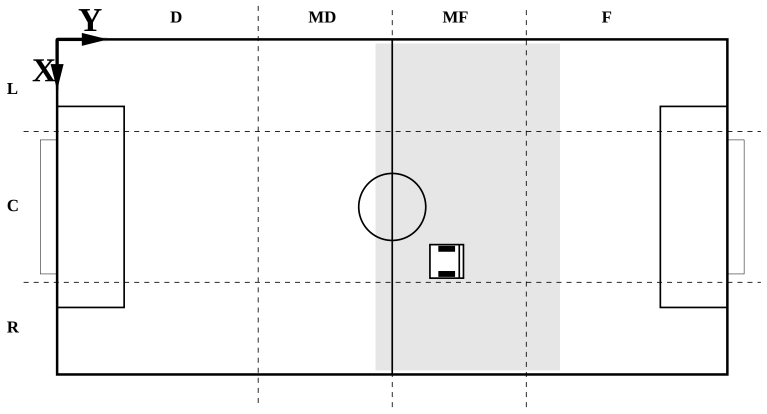

Figure 7: MidForward actuation zone.

- Forward – The Forward behavior induces the player to be in the *F* zone (see Figure 8). If the ball goes into our field, the F players mission is to keep track of the ball, although it should not move out of its zone by own initiative. When the ball moves into the F zone, it must try to take control over it and kick it into the opponents goal. An alternative consists of letting the Forward players move up and down the field, not across it.

Some general considerations:

- During the game the players must stay inside their influence zones. If another teammate is assigned to the same area, the players should be able to negotiate a position, so that each of them has a well defined operation area. This should not prevent players from entering the other area if necessary.

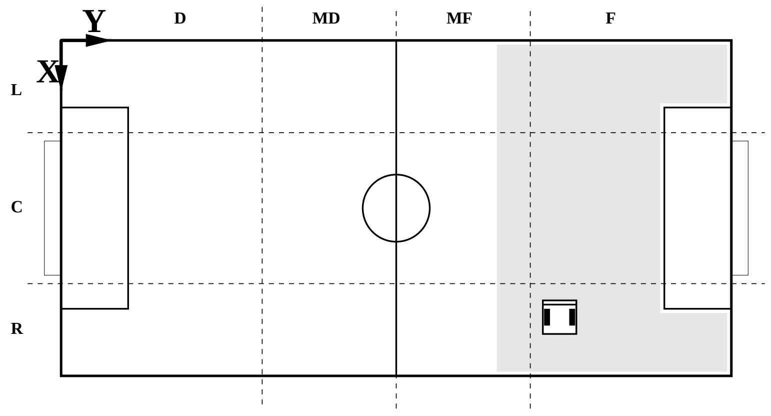

Figure 8: Forward actuation zone.

- When a pass is not aimed at a player, the corresponding kick should be oriented towards the *emptiest* area, i.e., it should try to keep the ball away from the opponent players.

4.3 Game State

The *game state* refers to either situations reached as a result of the application of RoboCup rules or to an evaluation of the current game status. State changes are induced by the time flow and teams actions during the game.

Examples of game states are as follows:

Game situations

- game-start – This happens in the beginning of the game, after a goal or when the game restarts after a break.
- penalty-for, penalty-against – There are several situations where a penalty should be awarded to one of the teams (check the rules).
- game-end – This is signaled by an external event (e.g., two whistle blows).

Symbolic evaluation of game status

- ball-our-ofield – The ball is in our possession, i.e., one of our players has ball possession. The ball is in our field.
- ball-nour-ofield – The ball is not in our possession, i.e., none of our players has ball possession. The ball is in our field.
- ball-our-tfield – The ball is in our possession, i.e., one of our players has ball possession. The ball is in the other team field.

- ball-nour-tfield – The ball is in our possession, i.e., one of our players has ball possession. The ball is not in our field.
- losing & close to the end of the game.
- ball close to our goal.

4.4 Scenarios for Game Situations

Pre-defined scenarios are usually associated with the game states corresponding to game situations (see above). Examples are:

- In a game-start situation the players must move to their pre-determined start positions (see Figure 9). Precise positioning of players must be accomplished at this state as they must be correctly positioned prior to the start of the game. After positioning, the players will wait for the external kickoff signal (e.g., a whistle blow) that signals the start of the game.
- In the penalty-for situation, the player in charge of penalties must move to the penalty-shoot position and wait for the start signal, after which it will trigger a pre-defined penalty-shooting behavior.
- When game-end is signaled, all the players must immediately stop.

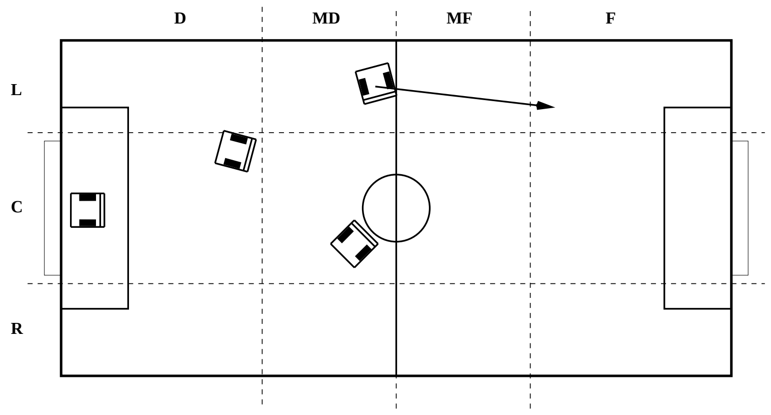

Figure 9: Players position at game start-up.

4.5 Dynamic Strategies and Tactics

During the game the ball moves in and out of our field. Depending on the ball position and movement direction, the current game state, the result, the number of available players and their behaviors, the opponents positions and the elapsed time, a strategy is defined. This is inspired by real soccer. The possible strategies are the following:

- defense – The ball must be prevented from entering our field. If it enters, it must be moved into the opponents field and ultimately, into its goal. Several defense tactics exist. The one to use is chosen based on some criteria, dependent on the game state.

  – *Strong Defense* (SD) – This strategy points to creating a continuous, physical barrier between the ball and our goal. It is aimed at avoiding opponent players from moving towards our goal; The captain moves all players into defense positions, i.e., Mid-fielders behavior is changed to Defender behavior and Forward players are changed to Midfield behavior. Defenders are still defenders, although some re-positioning may be necessary. When re-positioning, the Defender players should try to avoid to occlude the Goal-Keeper visibility of the field, i.e., the DC zone should be free of players (see Figure 10).

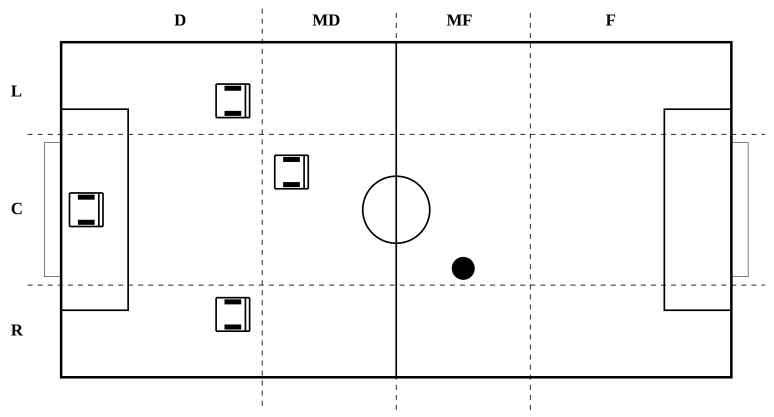

Figure 10: Strong defense positions.

  – *Medium Defense* (MD) – Points towards a strong defense and a good recovery mechanism, essential for counter-attack. The concept is illustrated in Figure 11. The difference between the SD and the MD is that in MD not all players are moved into our field. This gives the possibility of making the transition to Counter-Attack easier, as one of the players stays in the opponent field.

- counter-attack – A counter-attack happens if the team is positioned to move the ball quickly into the opponent field and score a goal. It requires a Defender, to prevent a possible intersection of the ball by an opponent, a Medium, to pass the ball into the Forward area, and a Forward player to kick the ball into the opponent goal. If the movement is performed without major opposition (i.e., it does not start moving towards our field), it should be changed to an attack movement. The counter-attack scheme is presented in Figure 12.

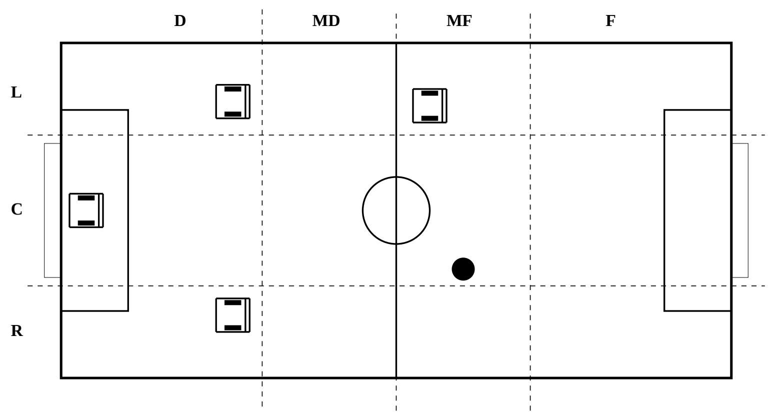

Figure 11: Medium defense positions.

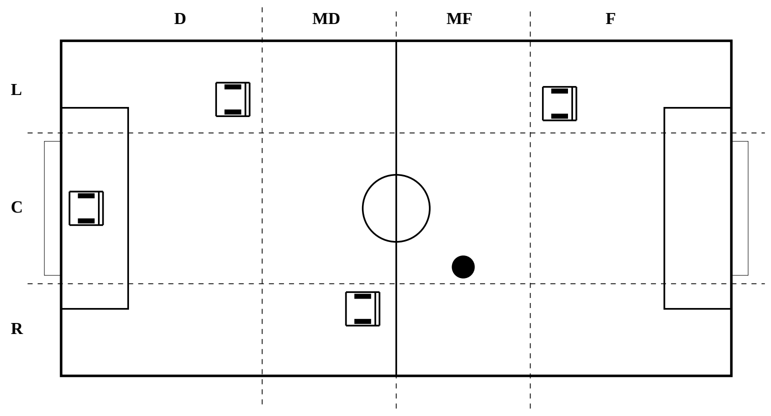

Figure 12: Counter attack and lose defense position.

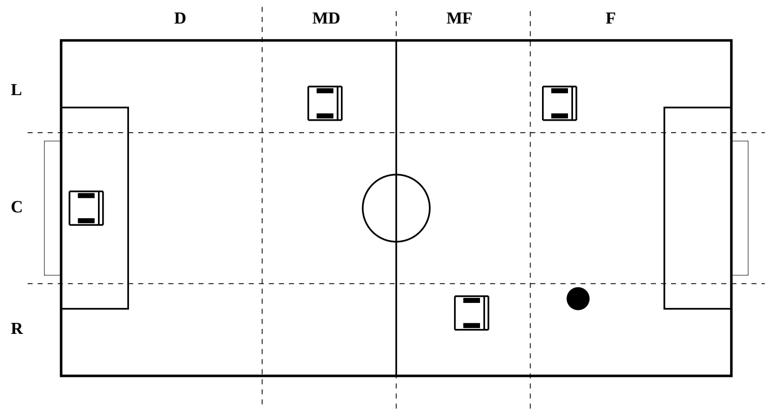

Figure 13: Close attack on the opponents goal.

- attack – In an attack movement all team is moved to the front. Besides the GK, there is only one player in our field (see Figure 13). This movement requires two Midfield players (one in the MD zone and another in MF zone), and one Forward. The idea is to have the ball passed from the M zone into the F zone, where a Forward player is to pick it up and kick at the goal.

References

[1] Alex Drogoul and C. Dubreuil. A distributed approach to n-puzzle solving. In *Proceedings of the Distributed Artificial Intelligence Workshop*, 1993.

[2] Lynn Andrea Stein. Preaching what we practice: How ai is changing the concept of computation. AAAI-97 Invited Presentation, July 1997.

[3] Lynn Andrea Stein. *Interactive Programming In Java.* Morgan Kaufmann, 2001. (to appear).

[4] Rodrigo M. M. Ventura and Carlos A. Pinto-Ferreira. Problem solving without search. In Robert Trappl, editor, *Cybernetics and Systems '98*, pages 743–748. Austrian Society for Cybernetic Studies, 1998. Proceedings of EMCSR-98, Vienna, Austria.