Communications for cooperation: the RoboCup 4-legged passing challenge

Carlos E. Agüero Durán, Vicente Matellán, José María Cañas, Francisco Martín
Robotics Lab - GSyC
DITTE - ESCET - URJC
{caguero,vmo,jmplaza,fmartin}@gsyc.escet.urjc.es

Miguel A. Ortuño Pérez
GSyC
DITTE - ESCET - URJC
mortuno@gsyc.escet.urjc.es

***Abstract*— Communications are the basis for the collaborative activities in the TeamChaos 4-legged team. In this paper we present the communications architecture developed both to let teammates communicate, and to easy the debugging of robot behaviors from external computers. Details of its implementation on the aiBo robots are also given. Using this infrastructure we describe a protocol for role exchange named *Switch!* that we have created. We also describe the use of both the communication architecture, and the *Switch!* protocol in the passing challenge of the 2006 edition of the RoboCup.**

I. Introduction

RoboCup [1] is an international joint project to promote AI, robotics, and related field. It is an attempt to foster AI and intelligent robotics research by providing a standard problem where wide range of technologies can be integrated and examined. RoboCup chose to use soccer game as a central topic of research, aiming at innovations to be applied for socially significant problems and industries.

In order for a robot team to actually perform a soccer game, various technologies must be incorporated including: design principles of autonomous agents, multi-agent collaboration, strategy acquisition, real-time reasoning, robotics, and sensor-fusion. RoboCup is a task for a team of multiple fast-moving robots under a dynamic environment.

The competition is organized in various leagues according to the type of robots used, their size, etc. The results showed in this paper have been used in the 4-legged category [2], where only aiBo robots are allowed.

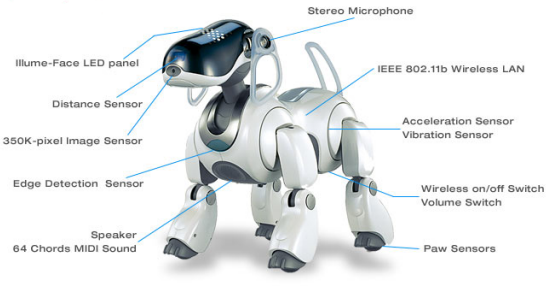

Fig. 1. aiBo robot with its main sensors detailed

[0]1-4244-0537-8/06/$20.00 C2006 IEEE

RoboCup 4-legged league is organized around two main competitions: soccer matches and technical challenges. Soccer teams are composed by four robots and they must play football without the help of any external aid (human or computer). Only communication among the members of the team is allowed.

Technical challenges include one open and two specific challenges. Technical challenges are specifically designed problems to focus research in particular issues for the league evolution. One of the specific challenges created for the last edition of RoboCup is the *passing challenge*, where teams must develop passing and catching skills. We will use this challenge to test our coordination issues.

We have developed a coordinated behavior for this challenge based in two main pillars: communications and roles exchange. Communications can be defined as an explicit method for cooperating. Other methods mentioned in [3] use the environment or make interactions among the robots via sensing as a result of agents sensing one another, but without explicit coordination. Dynamic roles exchange is based in the *divide and conquer* idea. A global task is subdivided in a more simple set of sub-tasks. Using some kind of scheduler, the sub-tasks are assigned to the robots obtaining an increase in the overall goal of the team achieving the original shared task.

In this article, we present the software architecture utilized, including the communications layer developed. The protocol designed for roles allocation is detailed in section IV. Finally, we show as a real application the results obtained in the passing challenge in the RoboCup 2006 edition.

II. TeamChaos architecture

TeamChaos is a multi-university, multinational effort to create a multi-robot team of soccer playing physical robots to enter the RoboCup international competition in the 4-legged league. TeamChaos software development is organized in three main projects: *TeamChaosRobot*, *ChaosDocs*, and *ChaosManager*.

TeamChaosRobot contains all code related to the robot, *ChaosManager* is a suite of tools for calibrating, preparing memory sticks and monitoring different aspects of the robots and the games, and *ChaosDocs* contains all the available documentation, including team reports, team description papers, and RoboCup applications.

For the purpose of this article, we are going to focus in the *TeamChaosRobot* project. The code developed has been written using the API offered by Sony included in the Open-R library [4]. The programming structure of the robot code is based in Open-R objects. Each object is a single thread and the mechanism for communicating objects is solved using a message exchange mechanism.

The *TeamChaosRobot* code is organized around four Open-R objects:

- ORLROBOT is the low level processing object that interacts with the hardware to produce motion or to process images. This object produces a data structure comprising local perceptions and odometry estimations. It accepts motion commands and perceptual "needs" [5], that is, elements (ball, landmarks, etc.) that the higher levels would like to track.
- ORHROBOT is the high level object, it implements the control strategies and maintains local and global representations of the environment.
- ORTCM is the communications managing object. It is in charge of sending and receiving data to and from other robots and/or to and from external computers.
- ORGCTRL There is an additional Open-R object in TeamChaos named ORGCtrl that is the responsible of managing the game all instructions sent by the *GameController*, which is an electronic referee during the match.

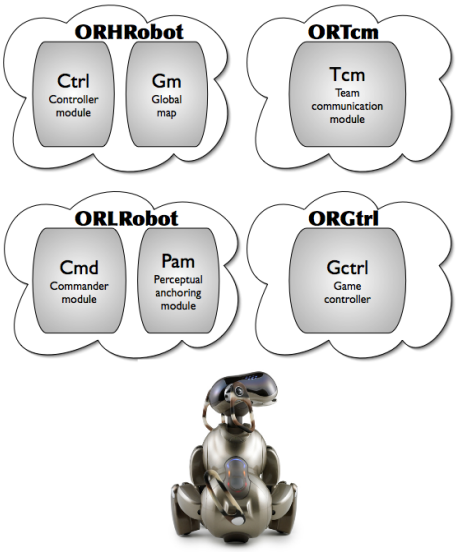

Fig. 2. Modules organization of the *TeamChaos* code

Each robot uses a conceptual layered architecture, which is a variant of the *ThinkingCap* architecture [6]. *ThinkingCap* is a framework for building autonomous robots jointly developed by Örebro University and the University of Murcia that aids developers in the process of implementing complex behaviors. The robot software architecture is divided in three levels or layers: the lower level offers hardware abstraction, the middle layer maintains a model of the environment and the higher level manages the strategy actions.

This theoretical layers have been implemented into the code as software blocks that we have called modules. The lower layer (implemented in the CMD, or Commander Module) provides an abstract interface to the sensori-motoric functionalities of the robot. The middle layer maintains a consistent representation of the space around the robot (through the PAM, or Perceptual Anchoring Module), and implements a set of robust tactical behaviors (in the CTRL, or Controller Module). The higher layer maintains a global map of the field (kept in the GM, or Global Map), and makes real-time strategic decisions based on the current situation (situation assessment and role selection is performed in the HFSM, or Hierarchical Finite State Machine).

As we can see in figure 2, the *ORLRobot* and *ORHRobot* objects manage the three control layers of the architecture. Inside each object we can see the modules involved. Note that there is not a clear vision of the layers in the figure, this is due to the figure represents the architecture from the implementation point of view and not from a design angle.

The communications module contained in the *ORTcm* is briefly explained in next section because is the most interesting module for our cooperation purposes.

The next sections of the article are going to show the more relevant points of the communications infrastructure and the dynamic task allocation mechanism designed. Once we have explained those two main elements, section V gives the details of the proposal we have chosen to try to solve the RoboCup 4-legged passing challenge.

III. COMMUNICATIONS

Communications let us use external tools for making laborious tasks more easily. For instance, they let us receive images and data from the robot to debug its behavior, to refine the camera parameters, to reconfigure the robot camera while the robot is running, or to teleoperate the robot to check kicks or locomotion using communication between robots and *ChaosManager* too. Besides being used as support for these tools, communications among robots are needed for sharing information among them and playing in a coordinated way.

The communication protocol developed is stateless, so there are not any connection establishment process, neither for termination. Also, transmissions are not reliable. Due to real time needs, communication protocol does not use any ACK's or retransmissions. It is better to lost some information than retransmits several times and receive old information.

OPEN-R offers both TCP/IP and UDP support but we have chosen the UDP alternative for make lighter and real time communications. At this moment, TCM opens four UDP sockets, three of them for *ChaosManager* communications, and the other one for communications among robots.

An *ExternalMessage* data structure has been developed as the exchange unit between entities. It has a set of fields describing the data source and destiny, both the source entity

(AIBO or computer), the TeamChaos module that has sent it, data length, data transfered in the message, etc.

Each robot has an unique IP and it is associated to his robot identifier in a configuration file. When someone sends a message to the identifier of a robot, the consequence will be an unicast message towards the IP joined to this identifier. We also allow broadcast transmissions. This feature is useful for sending debug information that is analyzed and showed by our external debugging tools (*ChaosManager*) and for sharing information among the members of the team.

The communications module developed is explained in more detail in [7] and [8]. In this article, we only want to mention the highlights of the communication library, embedded in our aiBo framework, relevant for the cooperation challenge faced (those highlights are described in the following sub-sections).

A. TcmClient: Open-R abstraction

The use of Open-R objects is a bit arduous due to the message passing mechanism used in Open-R. Developers that want to use the primitives offered by the communication module need to know the Open-R exchange system to fulfill their jobs. We have designed Open-R wrapper called TcmClient. TcmClient offers a simple interface to every object for sending/receiving data hiding all Open-R details.

B. Asynchronous sent

Each Open-R object is a single-thread process, however the *ORTcm* object must listen messages from several independent robots. Bottlenecks may happen in those objects if they have to wait its turn in the *Team Communication module*. This is the reason for developing a non-blocking scheme for sending data. We have developed a message queue, common to all objects. The communication library manages all operations that involves this queue.

C. Independent protocol data length

Open-R offers TCP support but we have selected the UDP alternative for making lighter and real time communications due to the lack of retransmissions, connection establishments, etc.

We have chosen UDP as the protocol for the team communications. There are not any connection establishment and termination. Also, transmissions are not reliable and the communication protocol does not use any acknowledgments or retransmissions.

UDP protocol establishes a limit for data length. We have developed a cut/compose phase into the Tcm object that allows to exceed the UDP limit in a transparent way.

D. Asynchronous reception

Normally, objects decide when are they sending data, but they do not know when they are going to receive it. The option of waiting until data arrives is clearly discarded due to the mentioned non-blocking requirements of all objects. We have implemented a callback mechanism to solve this situation.

It is necessary to register a method associated to a specific type of data. The communication layer associates the module, data type and callback method and automatically invokes the suitable method when data are received.

The concept could be summarized into the next phrase: *I am the **Global Module** and I am interested in **localization_data**. When some data localization arrives, please invokes the method **dispatchLocalizationData**.*

E. Independent type messages

Data must be serialized before it can be sent. This is a simple task when we know what kind of data we are handling. The problems arise when we are designing an independent communication object that does not know which kind of data (pointers, objects, scalar values, etc.) is going to manipulate. We have created an interface class called *ISerializable* with two empty methods that must be implemented for each object that wants to travel using the network. Those methods are *serialize()* and *unserialize()* and every object will need to inherit from *ISerializable* to be sent.

Along this section we have talked about the key points of the TeamChaos communication infrastructure that allow developers to design cooperative behaviors. Next section introduces the dynamic task allocation problem, which we have found interesting for coordination issues, and details some aspects of our proposal to this problem.

IV. Dynamic task allocation: *Switch!*

Dynamic task allocation is a requirement for a multi-robot system that wants to fulfill a common task. Splitting the global task of the team in several subtasks is a way of cooperating among the members of the group. In very static environments it is possible to assign fixed roles to each robot. However, in dynamic scenarios we need dynamic task allocation to adapt the team to the changes of the environment. In [9] is detailed in depth the problem of dynamic task allocation.

We have designed and implemented a simple dynamic task allocation mechanism called *Switch!* [10]. The main purpose of *Switch!* is to offer a software layer to applications, that manages all issues related to role assignment. This system provides a simple interface for knowing which is the most appropriate task for a particular robot given the actual "state" and a set of available "roles". Inside *Switch!*, there is an explicit communication protocol among robots based in local perceptions and global localization of each robot.

Our role allocation algorithm is based in heuristic functions. Those functions evaluates some parameters (distance to relevant objects, localization, etc.) and obtains a value for each available role. We will call this value *utility*. *Utilities* will be calculated periodically and roles will be assigned to robots in a priorized way according to the values obtained.

A general definition for *utility* is "value to estimate the cost of executing an action". *Utilities* has been used in several environments, they have been applied to game theory, operations research, economics and multi-robot coordination, as we can see in [11]. In our approach, utilities are employed

to evaluate the degree of adaptation of one role to one robot, in a particular moment, and in a specific environment.

In our proposal *utilities* will be individually computed by each robot as the weighted sum of several factors. In order to describe those factors, we will use the formalism described in [10]:

- Let $I_1, ..., I_n$ be the set of n robots.
- Let $J_1, ..., J_n$ be the set of n priorized roles and $w_1, ..., w_n$ their relative weights.
- Let U_{ij} the nonnegative utility of robot I_i for role J_j, $1 \leq i, j \leq n$.

Every robot I_i always must have an associated role, and every role J_j should be allocated to one robot in a priorized way. If during some period of time communications are perturbed, some roles will not be assigned (due to all robots will choose the more priorized role). When the communications will be restored, role assignment will start selecting among all roles again.

The steps for using *Switch!* are the following:

1) Choose an specific **set of roles**. For example: {*Goal-keeper, defender, striker and supporter*} in the soccer domain. The roles in robots are going to change according to the variations in environment and the position of each robot.
2) Establish the **order of assignation among the roles**. Order means priority of assignment.
3) Specify one ***utility* function for each role** based in a given heuristic. A *role exchange cost* has to be included in all heuristic functions to prevent excessive roles exchanges if only small changes have happened in the environment. This factor provides some kind of hysteresis to the system.
4) Create the **information exchange unit (IEU)**. This is the entity that robots communicate among them. The information contained inside an IEU will be received by each robot. *Switch!* will update its data structures with this information, and the heuristic functions will use it for generating *utilities*.

Once the previous items are defined, the dynamic task allocation mechanism starts its job. *Switch!* computes a matrix with all combinations among robots and roles according to the heuristics configured. Then, it selects the more suitable robot for the roles available in the specified order. Note that *Switch!* is a non centralized method, so each robot will be running all steps we are describing here.

Communication among robots is transparently managed by *Switch!*. It sends and receives the information exchange units and also monitorizes the state of the teammates. If it detects some failure of one member of the group, the less important role is discarded in the process of role assignment (as we can see in figure 3). This guarantees that the roles with higher priority are always assigned.

V. Cooperation in the 4-legged passing challenge

Last RoboCup edition was the first time for the *passing technical challenge*. This challenge has been created to promote coordination, passing, and catching skills among teams participating. In this challenge each team is required to provide three robots. Each robot is placed on the field inside a circle. The center of each circle is known by the robots. Initially the robots are in a previous state called *set* for 15 seconds, this enables them to localize. The robots are then placed into the *playing* state and given two minutes to pass the ball around.

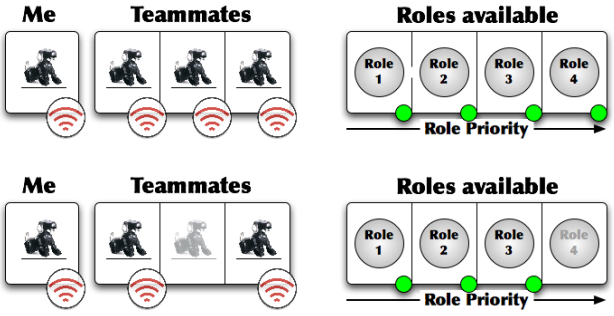

Fig. 3. *Switch!* monitorizing teammates and selecting available roles

Fig. 4. Picture of an instant of the passing challenge in our laboratory

We have used our communication architecture and *Switch!* as the main ingredients for developing the behavior to participate in the challenge. So, we have a solid base for implementing the cooperation needed but we also have some weak points that set some limitations in our behaviors. The main restriction is that we are not able to walk with the ball catched. This constraint forces us to maintain the ball controlled inside the circles. If the ball leaves one of the circles, we will not be capable to return it to the circle and try a pass. Rules say that pass must be started inside a circle and catched inside another.

As we explained in section IV, our dynamic role allocation system needs that some parameters will be adjusted. Next we are going to describe the instance of *Switch!* used to face the challenge.

1) **Set of roles:** *striker, catcher1, and catcher2* are the roles defined. *Striker* should catch the ball, point out to a

teammate, and then kick the ball towards some *catcher*. *Catchers* should be looking at the ball and catch the ball when it will be kicked.

$$J_1 = Striker,\ J_2 = Catcher1,\ J_3 = Catcher2$$

2) **Order of role assignation:** In this case the order is first allocate the *striker* role, and then the others. This is to favor that the robot best situated to catch the ball always was selected first. The weight of the *Striker* role is higher than the others. The roles with more weight (more priority) will be assigned before the others.

$$w_1 = 0.50,\ w_2 = 0.25,\ w_3 = 0.25$$

3) ***Utility* functions:** As the numbers of robots is very low, and given that there are only two different roles available, *catcher1* and *catcher2* will run the same behavior. We have only declared the utility function for the *striker* robot. If *Switch!* evaluates that the robot is not the *striker* (due to another robot has better *Utility* for this role), it will run the *catcher* behavior.
Switch! will evaluate the degree of adaptation to all robots to the *striker* role. The better positioned robot will be the *striker* and the rest will run the *catcher* behavior. The heuristic selected for this role is based in the local estimation of the distance to the ball ($D_{I_i,Ball}$). Note that less *utility* is better. *Role_Exchange_Cost* is a penalty added to the utility function when the robot changes from one role to another different. This extra cost of changing the role avoids very quickly role changes adding some hysteresis to the system.

$$U_{i,Striker} = D_{I_i,Ball} + Role_Exchange_Cost$$

4) **Information exchange unit:** As the heuristic selected only uses local ball estimation, this is the only data shared among the robots. We have tested other method that uses a global estimation of the ball but this approach has a clear problem: a good localization is needed to fix a global ball position. The uncertainty in localization adds bigger errors than the alternative chosen that only shares local estimations.

Once we have configured *Switch!*, using the *ChaosManager* we have created the behaviors for the robots. One of the tools of the *ChaosManager* is the HFSM (Hierarchical Finite State Machine) that allows to integrate low level behaviors into a more complex one. Using this tool we have designed a common behavior for all robots with the block structure described in figure 5:

- **Initialize localization:** In this state the robots rotate and look for landmarks in order to stabilize localization. The duration of this state is fixed and ends when expires a timeout.
- **Localize in circle:** The robot computes the distance to its own position towards the center of the three circles. Note that the center of the circles are known but the robots do not know in which of those circles are located (they always start inside a circle). [1]

[1]In 2006 RoboCup edition, organization relaxed this rule allowing robots to know in which circle they started

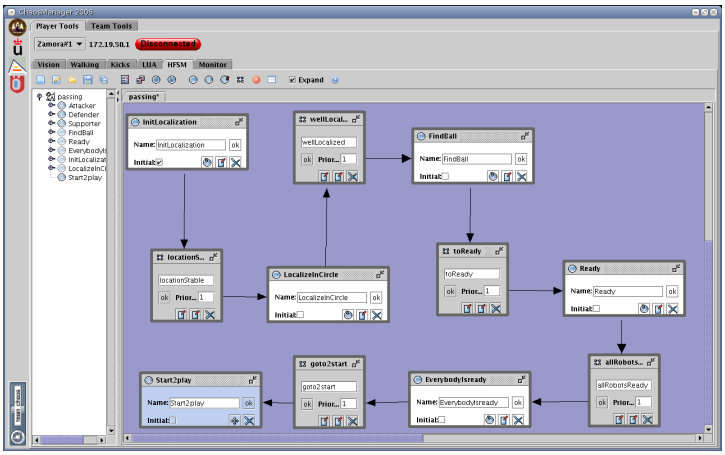

Fig. 5. HFSM tool with the behaviors diagram using states and transitions

Once the first robot estimates the closest distance to some circle, it supposes that it is inside this circle. This robot composes a message for informing to its teammates that this circle has been allocated minimizing the options of the teammates. If the localization of some robot has some error, using this simple protocol the probability to choose an erroneous circle is reduced. We have also implemented acknowledgements to guarantee that all robots receive those messages.

- **Find ball:** When a robot has a stable localization and it knows in which circle is located, it starts to rotate in order to find the ball. We use our own scan algorithm for quickly find the orange ball. Once the robot is correctly faced towards the ball, the robot is ready to play.
- **Ready:** This step is designed to synchronize all team members before playing. If robots are not synchronized, maybe the robot located closer to the ball, would be localizing inside its circle even, while other robot starts to approach to the ball (with the *striker* role). We need that all robots will be ready in order to start in the best conditions for guarantee that the *striker* role will be assigned to the appropriate robot.
In this state the robot broadcasts a *ready message* to its teammates. This ready message must be confirmed with an *ack* by every robot. We have included retransmissions in this synchronization protocol.
- **Everybody is ready:** When a robot has sent its *ready message*, it has received the confirmation from its teammates, and it has received the *ready message* from the other members of the group, it assumes that all robots are ready. Now, the robot starts to play.
- **Start to play:** This is a state always guided by *Switch!* that selects the correct behavior for the robot. If the robot has been allocated the *striker* role, it will run a low level behavior for catching the ball. Now, the robot will point out to the center of some of the circles using its own position and the known location of the destination centers. At this moment, it will broadcast an *imminent passing*

message for warning the teammates and will kick the ball.

Catchers will be running a behavior called *face ball* that will do robots always stay looking at the ball. If they receive an *imminent passing message* or the distance to the ball gets small, they will execute a special kick to stop the ball.

Then, *Switch!* will do a role exchange in some robots, *striker/catcher* roles will be swapped and the game will continue.

VI. Conclusion

Cooperation is a requirement for improving multi-robot teams. In this work we have experimented with one method to generate cooperation using explicit communications among the members of the group. We have observed that the use of coordination aids to solve some problems, as the RoboCup passing challenge.

We also used the passing challenge to validate our communication infrastructure, developing some protocols for the challenge easily. It means that the effort for hiding Open-R details and for improving the previous communication architecture has been useful. In this scenario we have checked its correct operation and also the value of a good dynamic task allocator. In every robot *Switch!* always have managed the roles, keeping the *catchers* for going to the ball and contributing to deploy an organized common behavior.

In RoboCup 2006 at Bremen, only four teams achieved to complete some pass. We were one of the teams that reached the objective finishing in the third position of this technical challenge[2].

We also used *Switch!* in the typical soccer matches and the classic set of roles (defender, stricker and supporter) in *RoboCup* 4-legged competition worked satisfactory. According to the ball and robot positions, playing roles were selected among the members of the team. The impact of using our dynamic role allocation mechanism in the game resulted in a better positioning strategy and better times catching the ball.

Acknowledgments

Authors would like to thank the TeamChaos members for their support. This work has been partially sponsored by grants *DPI2004-07993-C03-01* and *S-0505/DPI/0176* corresponding to *ACRACE* and *Robocity 2030* projects respectively.

References

[1] H. Kitano, M. Asada, Y. Kuniyoshi, I. Noda, and E. Osawa, "Robocup: The robot world cup initiative," in *ICJAI-95 - Workshop on Entertainment and AI/ALIFE*, 1995.

[2] Legged Robocup Federation, "Robocup Four-Legged League," http://www.tzi.de/4legged/bin/view/Website/WebHome, 2006.

[3] Y. U. Cao, A. S. Fukunaga, and A. B. Kahng, "Cooperative mobile robotics: Antecedents and directions," *Autonomous Robots*, vol. 4, no. 1, pp. 7–23, March 1997.

[4] Sony, "Sony AIBO SDE and Open-R SDK," http://openr.aibo.com, 2006.

[5] A. Saffiotti and K. LeBlanc, "Active perceptual anchoring of robot behavior in a dynamic environment," in *Proc. of the IEEE Int. Conf. on Robotics and Automation (ICRA)*, San Francisco, CA, 2000, pp. 3796–3802, online at http://www.aass.oru.se/~asaffio/.

[6] H. Martínez and A. Saffiotti, "Thinkingcap-II architecture," online at http://ants.dif.um.es/~humberto/robots/tc2/.

[7] C. E. Agüero, F. Martín, H. Martínez, and V. Matellán, "Communications and basic coordination of robots in TeamChaos," in *Actas VII Workshop de Agentes Físicos*, 2006, pp. 3–9.

[8] TeamChaos, "Team report 2005," 2006, online at www.teamchaos.es.

[9] K. Lerman, C. V. Jones, A. Galstyan, and M. J. Mataric, "Analysis of dynamic task allocation in multi-robot systems," *International Journal of Robotics Research*, vol. 25, no. 4, pp. 225–242, 2006.

[10] C. E. Agüero, V. Matellán, V. Gómez, and J. Cañas, "Switch! Dynamic roles exchange among cooperative robots," in *Proceedings of the 2nd International Workshop on Multi-Agent Robotic Systems - MARS 2006*. INSTICC Press, 2006, pp. 99–105.

[11] B. Gerkey and M. Mataric, "On role allocation in RoboCup," in *RoboCup 2003: Robot Soccer World Cup VII, 2004*. Springer-Verlag, 2004, pp. 43–53.

[2] 2006 results are available at http://www.tzi.de/4legged/bin/view/Website/Results2006