UChile1 2004 Technical Report

Javier Ruiz-del-Solar, Paul Vallejos, Juan Cristóbal Zagal, Raúl Lastra, Carlos Gortaris, Iván Sarmiento

Department of Electrical Engineering, Universidad de Chile
http://www.robocup.cl
jruizd@ing.uchile.cl

Abstract. The U-Chile-1 four-legged team is an effort of the Department of Electrical Engineering of the University of Chile in order to foster research in robotics at high level. U-Chile-1 is composed by a group of undergraduate and graduate students of electrical engineering and computer science of our engineering school. This document describes the relevant aspects of the software that was developed from scratch, and is constantly being updated, by our team. The system has shown to be a relatively successful approach; we had an acceptable participation in RoboCup 2003, our first participation in a world soccer robot championship. This year we have improved several aspects of our U-Chile-1 software (mainly localization, actuation and strategy), and we have also ported it to the new ERS 7 robots. We have also developed the first realistic simulator for the four-legged league. All these aspects are here reported.

1. Introduction

1.1. Motivation

As a side effect of research activities, robotics is emerging as a new way of teaching sciences and technology; it allows students to apply a great amount of knowledge to a variety of problems that arise when solving real robotics problems. At our department of electrical engineering a main interest is to foster the research of new wave robotics. Mobile robotics research is intended to be the main subject of research within our robotics laboratory. RoboCup is a great opportunity to share new knowledge in the field as well as to compare the level of the universities by means of their performance during the games. We believe that it is possible to have a good participation in this kind of tournaments even with our very restricted budged, compared to other universities within the league.

For this reason me enter in 2003 to the RoboCup four legged league. We consider that in order to improve further the skills of any of such group of robots, it is necessary to incrementally incorporate learning mechanisms aiding the engineering task. We aim at contributing with this great challenge by proposing novel learning strategies on each one of the main issues of the RoboCup four-legged soccer robot problem. As all of us know, the four-legged soccer is a very constrained problem in terms of hardware (camera, processor, available memory, sensors, etc.) and game field, in which many different

processing approaches have been already proposed. As a new team in this league we want to specialize in the introduction of evolutionary, adaptive and collaborative methods for the perception, control and strategy subsystems.

1.2. System Architecture

Our software system is divided into four task-oriented modules: vision, localization, strategy and motion control, see Figure 1. The vision and motion control modules operate in each robot locally. The localization module is distributed, it operates on each robot and a global estimate of the overall localization is generated in a distributed fashion. The strategy module is also distributed, and allows the sharing of global information among the robots. In our current implementation these four task-oriented modules run, in each robot, in two different computer processes: one for vision and localization that we call the *perceptual process* module, and one for motion control and strategy, that we call the *engine process* module. The reason for having just two processes and not four is the synchronization problems we had in our previous implementation. It is important to remark that our software is fully portable between different AIBO robots. It runs in the ERS 210 and 220 models, and recently was ported to the new ERS 7 model without problems. In the next sections each of the mentioned modules are described.

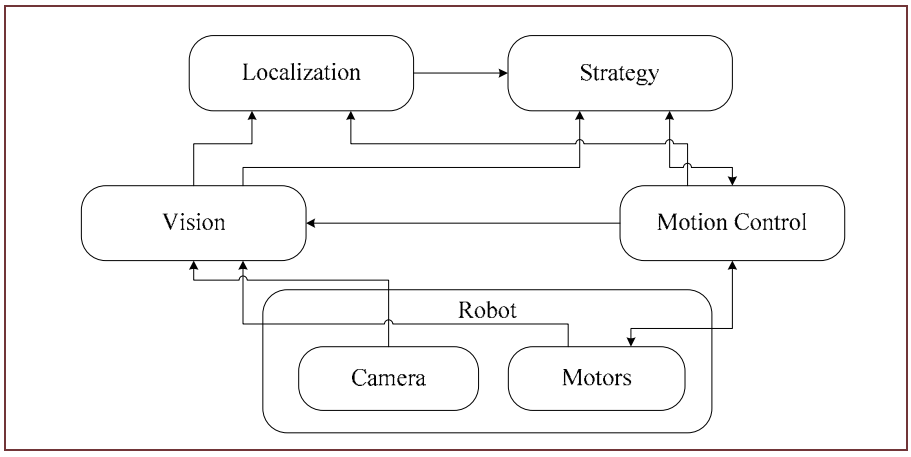

Figure 1. Modular organization of our system. In the bottom the low-level processes of vision and motion control. On top the high level processes of localization and strategy.

2. Vision

The vision module inputs are (i) the raw image data obtained from the robot camera, (ii) the head servo motor state angles obtained from the robot motor controllers and (iii) synchronization data as well as (iv) body inclination estimates obtained from the engine module. Vision module output consists on information of detected game field objects on the image, which is transmitted to the localization and strategy modules. For each detected object, their distance and azimuth relative to the camera and robot body are estimated and projected over the field plane. Confidence degrees are computed for the detection certainty and distance and azimuth estimations. These relevant objects correspond to the red or blue robot players, any one of the four colored beacons, the cyan and yellow goals and the orange ball. This module is divided into six processing sub-modules, which are: (1) color segmentation, (2) run-length encoding of the segmented image, (3) labeling of connected regions, (4) region extraction and characterization, (5) object perception, and finally (6) computation of distances, azimuth and projections of objects over the game field plane.

2.1. Color Segmentation

This sub-module receives as input, from the AIBO camera, the raw color image with their three channels Y, U, and V, and generates as output a single channel image containing identifiers for the background and the set of relevant colors of the game field, which are pink, yellow, green, cyan, blue, red, and orange. We have incorporated in this implementation the detection of white. The segmented image is obtained during operation by extracting identifiers from a 3-D look-up table of 64 levels in each Y, U and V dimensions. The Y, U and V values of each raw image pixel are converted to indexes in the range 0 to 63 for accessing the look-up table identifiers. For generating these look-up table values (training stage), a user watches an online video obtained from the AIBO camera while it is moving on the game field. Image regions that are related to the colors of the game field, as well as examples of non-relevant colors that are used for training the background class, are selected by the user. Training a special class for the non-relevant colors is a fundamental consideration in our implementation. Then, these sample pixels are related to the color identifier that is provided by the user. These samples are directly placed into the look-up table according to their coordinates in the YUV space. This process gives rise to the generation of class related clusters into the YUV space. We have observed that the shape of these clusters is usually oblong, in general difficult to be characterized in terms of rectangles or basic geometrical shapes. Instead of doing so, we just leave these shapes as they are, but we enforce the shape and internal structure of these clusters by applying a median mask over the look-up table values and morphological operators of dilation and erosion over the clusters. Dilation serves for filling in the interior of clusters, while erosion serves for keeping them under their original size. The median

mask is used for cleaning up the interfaces between clusters. In general the training process requires few images of the game field, and it takes about 15 minutes.

2.2. Run-length encoding

This sub-module takes as input the resulting image of the color segmentation sub-module, and generates a run-length codification of the image in the form of an array of 4xN elements, where N is the variable number of segments which are extracted from each frame. A run or segment consists of a set of foreground connected pixels of the same color identifier within the same image row. This codification is generated by scanning the image thought its rows and, for each run or segment, the coordinates x and y of the first pixel are stored in the array as well as the corresponding length of the run. The UNSW team uses this codification [1]; we consider that it is particularly useful for reducing the amount of data as well as for simplifying the subsequent labeling stage.

2.3. Labeling of Connected Regions

A variant of the connected components labeling algorithm is used for assigning a label to each connected segment, and for defining connected regions on the image. This process takes as input the 4xN run-length codification updating the fourth field of each segment with a corresponding label. This is performed by looking, on each row, the upper neighbors of each segment, and then copying their corresponding labels. If a segment has two upper neighbors having different labels, their corresponding labels are replaced going upwards along each branch. If a segment has no label, a new label is generated. In this process we consider only the foreground segments. Similar labeling processes have been proposed by other teams. We consider that this is a good choice in combination with the run-length codification.

2.4. Region Extraction and Characterization

The region extraction sub-module takes as input the run-length encoded and labeled image, and generates a list of regions or blobs which are related to each color. Each region is characterized with a set of region descriptors which are: the coordinates of the region bounding box, the coordinates of the region center of mass, the perimeter of the region, the mass of the region, the color of the region, and other descriptors. These descriptors are afterwards used for the recognition of relevant object into the game field.

2.5. Object Perception

2.5.1. Rule-Based Object Perception

The task of the object perception sub-module is to identify image regions that are related to the relevant game field objects. The recognition of objects is performed by evaluating the response of a set of rules that are applied over a set of candidate image regions or combinations of these candidate regions. For example the detection of a landmark is performed by evaluating the response of some rules to the combinations of regions of the colors pink, cyan, yellow, and green. The detection of the other players into the game field is also performed by applying rules over combination of regions. The detection of a ball usually requires that the related blob has the color of the ball, and if this is not the case one might expect to reject this candidate blob. The rules which are used in our team where derived by using our proposed genetic based method for visual rule selection and parameter tuning [2].

2.5.2. Motion Based Object Perception and Tracking

We have incorporated an alternative method for detecting moving objects, which also allows for the fast tracking of them; it is specially designed for the detection of the ball. The method uses motion segmentation, motion correction and Kalman filtering. This method is fully explained in [3].

The motion perception algorithm compensates in software the camera movement using the robot body and head movements' information. This information is used to correctly align the current frame and the background. In this way a stabilized background is obtained, although the camera is always moving. Afterward, two traditional movement analysis algorithms are applied over the stabilized background, background subtraction [4] and motion history [5]. Finally, the motion perception implementation uses Kalman filtering for the robust tracking of the moving objects. This allows to have reliable detections and to deal with common situations such as double detections or no detection in some frames because of lighting conditions.

2.5.3. Computation of Distances, Azimuth and Projections

The distances and the azimuth of the objects are first calculated with respect to the optical axis of the camera. These measures are then projected over the game field plane by simple matrix operations. These transformations are a function of the actual pose, described with three angles, of the robot head state. Finally the distances and azimuth estimates are transformed from camera relative estimates to body relative estimates. A special treatment is given for the computation of the robot relative ball position. The information of the bounding box which is related to the orange ball region is complemented with information of the shape of the orange region. A circle is fitted to this data and an accurate ball position is derived.

3. Localization

Our localization module includes two integrated functionalities: robot self-localization and ball localization (see full description in [6]).

The robot self-localization uses the following algorithms: Extended Kalman Filter (EKF), Monte Carlo Localization (MCL) and Single Landmark Localization (a method completely reactive to the vision information proposed by the German Team [7]). The results of these algorithms are integrated by MCL.

The ball localization is implemented using EKF. For determining the ball position is employed the information obtained locally by the robot (robot pose and robot observations), and also ball information transmitted by teammates (each robot shares its ball localization information with its teammates). In this way the ball localization process is distributed between the robots.

The general block diagram of the localization module is presented in Figure 2.

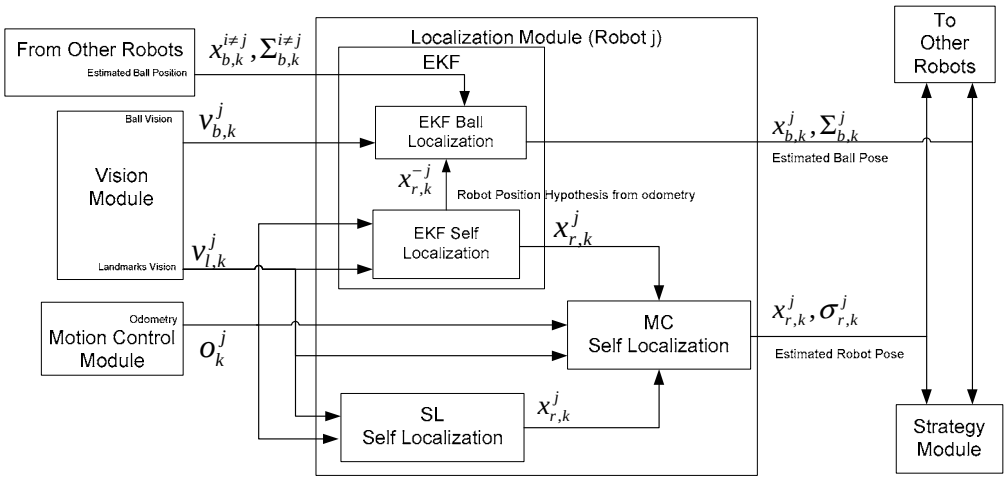

Figure 2. Localization Module Block diagram and its interaction with other modules. EKF – Extended Kalman Filter; MC – Monte Carlo; SL – Single Landmark. $v_{b,k}^{j}$: Ball vision information generated for robot j in time step k. $v_{l,k}^{j}$: Landmarks vision information generated by robot j in time step k. o_k^j: Odometry information generated by robot j in time step k. $x_{r,k}^{-j}$: Robot position hypothesis from odometry information. $x_{r,k}^{j}$: Robot position estimated in time step k. $x_{b,k}^{j}$: Ball position estimated by robot j in time

step k. $\Sigma_{b,k}^{j}$: EKF correlation ball error matrix generated for robot j in time step k. $\sigma_{r,k}^{j}$: MCL individuals' dispersion generated by robot j in time step k.

3.1. Self-Localization

3.1.1. Extended Kalman Filter Localization

The state vector $x \in \Re^3$ corresponds to the robot pose and is defined as

$$x = [x_{robot} \quad y_{robot} \quad \theta_{robot}]^T \tag{1}$$

The robot process is modeled with a non-linear function f , which depends on the past state vector x_{k-1} and the control orders sent to the robot motors u_{k-1}:

$$x_k = f(x_{k-1}, u_{k-1}, w_{k-1}) \tag{2}$$

A new measurement $z_k \in \Re^m$ is related to the past state vector x_{k-1} by a non-linear function h as:

$$z_k = h(x_k, v_k) \tag{3}$$

In (2) and (3) the random variables w_k and v_k represent the process and measurement noise, respectively. They are assumed to be independently (from each other), white and with Gaussian distribution. In each time step k the state vector is estimated by the EKF prediction and correction stages.

EKF Prediction Stage. The updated state vector $x_k^- \in \Re^{3x3}$ and the estimation error covariance matrix $P_k^- \in \Re^{3x3}$ are given by:

$$\begin{aligned} x_k^- &= f(x_{k-1}, u_{k-1}, 0) \\ P_k^- &= A_k P_{k-1} A_k^T + W_k Q_{k-1} W_k^T \end{aligned} \tag{4}$$

$Q_k \in \Re^{3x3}$ represents the error covariance of the process model ($Q_k = E[w_k w_k^T]$) and in our implementation it was estimated based on multiply experiments. Given that this matrix represents the certainly of the predictive model, if we increment it, the method will be more reactive to the vision information, therefore unstable, because in our case the sensorial information is usually very noisy. Conversely, if the Q_k matrix is nearly to zero

the estimation process just updates its estimation from the odometry information, i.e. from the robot model, which in our application is also very noisy. Therefore we set the value of this matrix manually, and we validate it in different robot pose estimation experiments with our robots. The final value of Q_k that we choose is:

$$Q_k = \begin{bmatrix} 0.1 & 0 & 0 \\ 0 & 0.1 & 0 \\ 0 & 0 & 0.05 \end{bmatrix} \quad (5)$$

$A_k \in \Re^{3x3} / W_k \in \Re^{3x3}$ is the Jacobian matrix of partial derivates of f with respect to x / w. In our case both were set to the identity matrix. P_k is an important indicator about estimation certainly, and it needs an initial value. If we were absolutely certain that our initial state estimate is correct we would let $P_0 = 0$, but this is not the case. For the sake of simplicity we initialize P_0 as the identity matrix.

EKF Correction Stage. In this stage the estimated robot posed is corrected using the observational information. First we have to compute the measurement vector z_k that will be of variable dimension because it depends on the number of detected landmarks *m*, which were recognized in time step *k*. z_k contains distances (d_i) and angles (o_i) relative to the observed landmarks:

$$z_k = \begin{bmatrix} d_1 & o_1 & d_2 & o_2 & . & . & d_{m/2} & o_{m/2} \end{bmatrix}^T \quad (6)$$

The measurements have associated a matrix of covariance $R_k \in \Re^{mxm}$, $R_k = E[v_k v_k^T]$. In our implementation this matrix is computed directly from a measurements' error table. For building this table several measurements were taken (different landmarks perceived at different distances and angles). In this way the variances of computed vision distance with respect to the real distances were calculated. Angles error variances do not depend of the object distance. If the object is complete on the image we can estimate the object relative angle in the best way and the error variance is low (2°). However, if the object is not complete on the image we can't estimate the angle correctly and the error variance is large (18°). Taking into account these considerations the error table was filled using the measured distances shown in Table 1. (angle errors are either 2° or 18°).

Table 1. Object distance dispersion (in centimeters).

Real Distance	50	100	150	200	250	300	350	400	450
Landmark Distance Dispersion	1	1	3	6	20	30	35	40	50
Goal Distance Dispersion	1	2	3	4	8	10	13	17	25
Ball Distance Dispersion	1	3	7	13	19	30	38	50	61

For each estimated robot pose we can compute $h(x_k,0)$ according to our geometric model. Then, we must calculate $H_k \in \Re^{mxm}$, that is the Jacobian matrix of partial derivatives of h with respect to x:

$$H_{[i,j],k} = \frac{\partial h_i}{\partial x_{j,k}^-} = \frac{h_i(x_{j,k}^- + \Delta,0) - h_i(x_{j,k}^- - \Delta,0)}{2\Delta} \quad (7)$$

with $z_{i,k} = h_i\left(x_k^-,0\right)$.

The Kalman gain is then computed as:

$$K_k = P_k^- H_k^T (H_k P_k^- H_k^T + V_k R_k V_k^T)^{-1} \quad (8)$$

with $V_k \in \Re^{3x3}$ the Jacobian matrix of partial derivatives of h with respect to v, set to the identity matrix in our case. Finally, the robot pose estimation and the covariance matrix are computed:

$$x_k = x_k^- + K_k(z_k - h(x_k^-,0))$$

$$P_k = (I - K_k H_k)P_k^- \quad (9)$$

3.1.2. Single-Landmark Localization

This method was implemented based on [7]. The rational behind the method is: when we have the distance to a single landmark, this measure allows us to describe a circle of positions around the landmark, and if we also consider the last position estimation, it is

possible to choose the nearest position in the circle like a “virtual position”. Then we can interpolate these position to obtain a position estimation in this iteration. The robot orientation will be obtained in the same way, interpolating the last orientation with the orientation obtained from the angles measurements to the landmark image. This idea is illustrated in Figure 3.

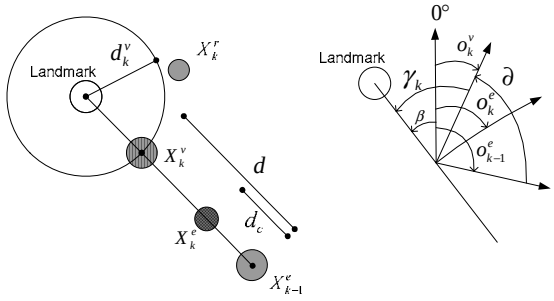

Figure 3. Single landmark localization distance update (left), and angle update (right). x_k^r : Real robot position in time step k. x_k^v : Virtual robot position in time step k. x_k^e : Robot position estimation in time step k. d_k^v : Measured landmark distance in time step k. d_c : d multiplied for the distance correction factor. γ_k : Measured angle to landmark in time step k. β : Angle between straight line from landmark to current position and x axis o_k^v : Virtual robot orientation in time step k. o_k^e : Robot orientation estimation in time step k. ∂ : Difference between last robot orientation estimation and virtual robot orientation in time step k.

The final estimation is obtained by interpolation between the virtual position and the last position estimation. The interpolation is computed using the distance correction factor. The distance correction factor is multiplied by the distance between the last position estimated to current virtual position. Finally, an average point is calculated. The distance correction factor used in our case is 0.5. The same factor is used to estimate the robot orientation.

3.1.3. Monte Carlo Localization

We use the classic MCL method with a population of 20 individuals. The individuals’ pose are actualized using the odometry information in the same way as in the EKF. MCL incorporates a probabilistic component to each movement model:

$$\begin{bmatrix} x_k \\ y_k \\ o_k \end{bmatrix} = \begin{bmatrix} x_{k-1} \\ y_{k-1} \\ o_{k-1} \end{bmatrix} + \begin{bmatrix} \Delta x \\ \Delta y \\ \Delta o \end{bmatrix} + \begin{bmatrix} fx*(1-p_j)*a \\ fy*(1-p_j)*b \\ fo*(1-p_j)*c \end{bmatrix} \quad (10)$$

The value p_j is called the weight of each individual and is a measurement of its importance. The values a, b and c are uniformly distributed random variable with range [-1,1], fx, fy and fo are constant mutation factors.

The next step calculates the weights for each individual using the sensorial information. The weight of individual *j* is calculated as:

$$p_j = \prod_{i=0}^{m-1} e^{-\frac{(h(x)_i^d - d_i)}{sigma_d}} * \prod_{i=0}^{m-1} e^{-\frac{(h(x)_i^o - o_i)}{sigma_o}} \quad (11)$$

with $h(x)_i^d / h(x)_i^o$ the measured distance/angle to the landmark *i*. We limited the weights change as:

$$p_j = \begin{cases} p_j & if \quad (p_j - p_{j-1}) < 0.1 \\ p_{j-1} + 0.1 & if \quad (p_j - p_{j-1}) > 0.1 \\ p_{j-1} - 0.05 & if \quad (p_j - p_{j-1}) < -0.05 \end{cases} \quad (12)$$

In this way we prevent large changes in the weights, produced for instance by vision noise.

For integrating the three described self-localization approaches we use the sensor resetting idea, i.e. SL and EKF give solutions that are introduced into MCL. The SL solution provides one individual, while EKF provides five individuals, one corresponding to the EFK solution, and four corresponding to variations over this solution. The four EKF variations are generated using mutation constants weighted by a random number between 1 and -1, the individuals inherited are placed around its father. SL and EKF population are inserted using a probabilistic algorithm described in Figure 4, where the worse MCL individuals will be more probably replaced.

The weights of the five introduced individuals are normalized, because the meaning of these weights is nearly related with the probability density function and will be essentials to the next step called re-sampling. The normalization process is described in **(13)**. The re-sampling step is a probabilistic way to choose some individuals of the population, e.g, the better individual have a larger probability to survive in the next population. The re-sampling algorithm we implemented is described in Figure 5.

$$\hat{p}_j = \frac{p_j}{\sum_{i=0}^{N-1} p_i} \tag{13}$$

where p_j is the weight not normalized calculated in **(11)** for the individual j, and $\hat{p}_j$ is the weight normalized assigned to the individual j.

```
while(j <= Population_insert_size){
  while(true){
    S = int_Random ;
    if(Individual_insert[j].Weight>= Individual[S].Weight){
      Individual[S]=Individual_insert[j] ;
      j = j+1 ;
      break ;
    }
  }
}
```

Figure 4. Insertion algorithm

```
  while( j < PopulationSize) {
    while ( i < Population Size ){
        S = Random_Value;
        if ( Weight( i ) < S )
          Individual [ i ].KeepNextPopulation [ j ];
          j = j + 1;
          break ;
        }
       i = i+1;
    }
    i = 0;
  }
```

Figure 5. Re-sampling algorithm.

Finally, the estimated pose is calculated as the average of all individuals (we assume that the individuals form a single cluster):

$$\bar{x} = \sum_{i=0}^{P\_Size-1} x_i * p_i \; ; \; \bar{y} = \sum_{i=0}^{P\_Size-1} y_i * p_i \; ; \; \bar{o} = \tan\left(\frac{\sum_{i=0}^{P\_Size-1} \sin(o_i)}{\sum_{i=0}^{P\_Size-1} \cos(o_i)} \right) \tag{14}$$

And the belief in the pose estimation is calculated as the dispersion of the individuals.

$$\sigma = \sqrt{\sum_{i=0}^{P\_Size} \left((\bar{x} - x_i)^2 + (\bar{y} - y_i)^2 \right) * p_i} \tag{15}$$

3.2. Ball-Localization

The implemented EKF allows us to include more variables in the state vector. Considering that, we included the ball position in the state vector, and compute the ball vision like any other landmark. The sensorial information gives us a ball position relative to the robot but, if we know the robot position it is possible to calculate the ball position in the field. In this way the new state vector $x \in \Re^5$ includes the robot pose and the ball position. All the operations needed by the EKF for self-localization are computed in the same way but with the larger state vector. However, in this case it is necessary to manipulate the process to get the robot pose non-dependent of the ball position. For obtaining that, when the H matrix is built, we set to zero the following values

$$\frac{\partial h_d^b}{\partial x_{robot}} = \frac{\partial h_d^b}{\partial y_{robot}} = \frac{\partial h_d^b}{\partial \theta_{robot}} = \frac{\partial h_\theta^b}{\partial x_{robot}} = \frac{\partial h_\theta^b}{\partial y_{robot}} = \frac{\partial h_\theta^b}{\partial \theta_{robot}} = 0$$

Robot communication is an important issue in the ball localization tasks. As mentioned, all robots share the estimated ball position with the teammates. For this reason the shared ball information is computed in the vector z as any other sensor measurement. In this way the z vector is defined by:

$$z_k^j = \left[d_1 \;\; o_1 \ldots \; d_{m/2} \;\; o_{m/2} \;\; d_b \;\; o_b \;\; x_b^{i \neq j} \;\; y_b^{i \neq j} \ldots \; x_b^{i \neq j} \;\; y_b^{i \neq j} \right]^T \tag{16}$$

where, d_i is the distance measured to landmark *i*. o_i is the angle measured to landmark *i*. d_b is the distance measured to ball. o_b is the angle measured to ball. x_b^i is the *x* ball position estimated by the robot i. y_b^i is the *y* ball position estimated by the robot i.

For computing the communication information in the same way to any other measure the, function $h(x)$ includes directly the ball position estimation.

4. Motion Control

This module is in charge of controlling the robot motors according to instructions provided by the strategy module and it is also in charge of generating odometric estimations of the robot displacements.

The module, according to , receives low-level inputs from the robot motor controllers, and exchanges information with the localization and strategy modules. The motion control module generates outputs for the motor controllers, and also provides an estimation of the robot inclination for the vision module. It provides odometric information to the localization and strategy modules. It receives instructions from the strategy module.

The motion of the head and body limbs are controlled independently. There are some instructions for the head, another set of instructions for the body limbs, and there are also some instructions for the overall robot. A head instruction consists on the specification of the three head angles. A body instruction consists on the specification of an infinitesimal change in the robot pose.

Each instruction is internally decomposed into a set of microinstructions. A microinstruction consists of a vector of length 15 in which the angular state of each robot joint is specified, as a reference. It also contains odometric information and timestamps for ensuring the synchrony of motions. A microinstruction is divided along the time into frames[1], each one having a duration of 8 milliseconds. For each frame, a motor reference value is computed as a linear interpolation among the references of the previous and current microinstructions. The number of frames generated from a microinstruction is given by the instruction, or it can be calculated based on the desired speed.

The decomposition of instructions into microinstructions is performed in two different ways. On the one hand, if the instruction consists on a set of static matrices specifying a motion itinerary, then the decomposition is performed by interpolating between the different reference points in each joint. On the other hand, if the instruction is part of a parametric walking, then the microinstructions are computed internally depending on the desired displacement. It is important to notice that in our implementation the robot is free to move in an omni directional fashion. Each joint variable is calculated in real-time in order to account for an arbitrary body pose change. This is performed by calculating the inverse kinematics of each joint and the displacement of each limb extreme, which is necessary for producing an arbitrary robot displacement.

The gait developed by our team has 20 parameters which define its locus, speed and neutral position of the legs. From the 20 parameters, nine define the shape of the locus:

1. The length of the locus;
2. The height of the lifting stage for the front locus;

[1] Frames are the robot's basic time unit.

3. The height of the air stage for the front locus;
4. The height of the descending stage for the front locus;
5. The height of the lifting stage for the rear locus;
6. The height of the air stage for the rear locus;
7. The height of the descending stage for the rear locus;
8. The shape factor (it set how elliptical is the air stage); and
9. The lift factor (it set a degree of diamond like into the lift stage).

There are another seven parameters that rule the operation point, these are the x, y and z coordinate of the front and rear feet, and an additional parameter that governs the turning rate of the AIBO, is used to determine the skew of all four loci in the x-y plane (see Figure 6), a technique introduced by the UNSW team [8].

Other three parameters define the timing of the loci:
1. The speed of the foot while in the ground stage;
2. The lift time; and
3. The time on ground factor.

Finally one parameter of processor charge is also used, it corresponds the number of points of the air stage whose inverse kinematics is calculated.

Since the AIBO is roughly symmetric, the same parameters can be used to describe loci on both the left and right side of the body. To ensure a relatively straight gait, the length of the locus is the same for all four feet. Separate values for the front stage height, air stage height, rear stage height and position of the foot are used for the front and back legs.

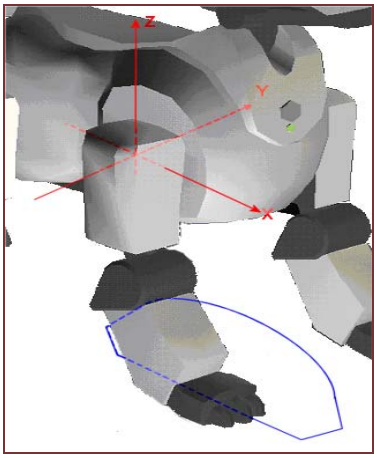

Figure 6. Close view of the AIBO model. In red is shown the employed reference system. In blue the general loci (from a leg).

To summarize, the following set of 20 parameters define the AIBO's gait:

- The locus shape (3 parameters: length, shape factor and lift factor.)
- The front locus shape modifier (3 parameters: lift height, air height and descending height)
- The rear locus shape modifier (3 parameters: lift height, air height and descending height)
- The front locus operation point (3 parameters: x, y and z)
- The rear locus operation point (3 parameters: x, y and z)
- Locus skew multiplier in the x-y plane (for turning)
- The speed of the feet while in the ground
- The fraction of time each foot spends on the air
- The time spent on the lift stage (its equal to the descending time)
- The number of points in the air stage whose inverse kinematics is calculated

Having this parameterization give us a low constrained gait. Over a very large space of gait parameters we have found a quite satisfactory arrangement for a fast gait using Back to Reality [9]. Now our 210 robots are running at the speed of 24.5 cm/s. We are extending this method for generating fast gaits for our new ERS-7 robots. We are also working on the automatic generation of ball kick behaviors [10]. For the next year we expect to use a continuum of kick behaviors that were automatically generated using the UCHILSIM simulator and the Back to Reality approach. The gaits and the ball kicks are obtained at the start from a file in the Memory Stick, which allows add, remove or change kicks and gait in a short time.

5. Strategy

5.1. Strategy Overview

The implemented strategy, as in human football, is based on communication and cooperation between players with different roles. Each robot takes decisions based on its perceptions and the information it receives from teammates. In addition, each robot communicates its internal strategy state to the teammates. In this way, every robot of the team knows the current strategy state of all its partners [11].

A role is defined as a state machine with transitions triggered by a combination of sensorial information, internal strategy state, and messages received from other robots. The role assignment is dynamic and the necessary conditions for a role transition are of the same kind of those for state transitions. States have a set of conditions defining the behavior to be executed. The behavior output is an action executed by the *Motion Control* module; typical actions are walk to specific locations with some orientation, and different kinds of kicks.

The strategy philosophy is common to all the field players, and the goalie is the only one with a special strategy.

5.1.1. General philosophy of the goalie's strategy:

The main task of the goalie is to protect the goal, that is, it must position itself between the ball and the goal. Two states define this behavior, which are executed slightly different depending on whether the goalie sees the ball or not. Obviously, this task can only be performed when at least one player knows the ball position. When none of the teammates knows where the ball is, the goalie stays in the middle of the goal and looks for it. If the ball comes close, and this action seems appropriate, the goalie goes to get the ball and kicks it away. After any attempt to kick the ball, the goalie, as all players do, looks towards the kick target place.

5.1.2. General philosophy of the field players' strategy

The field players are focused on going to the ball, one player at the time. The closest robot to the ball (the distance measurement takes into account not only the absolute distance but also the angle, in order to favor players that are already looking towards the ball), goes to it and attempts to kick it towards the opposing goal. The other players take up supporting positions. After any player kicks the ball, everyone looks towards the kick target place. At any time during the game, except while attempting to kick the ball, the players attempt to locate themselves if their position uncertainty becomes too high.

5.2. Potential Fields

Path planning is an important issue in robot soccer; it is very important to choose an adequate path while approaching to the ball, which allows the robot facing the rival goal on the ball location. A known methodology for path planning is the potential fields approach. This approach also allow the robots to place themselves covering the entire field, repelling each others for a correct positioning and keeping them away from the field edges. Potential fields methodology is based on creating virtual potential fields, attractive for desired objectives and repulsive for obstacles or undesired positions. In this way, the virtual force applied over the robot is calculated as the derivate of the potential field. The proposed methodology uses a force focus, this is, given a set of objects, generating repulsive and attractive potential fields, a punctual force can be calculated just on the interest point. When a decision of path must be taken, the force on the initial point is calculated, the calculated force vector direction is used to give a step, when that step is complete another force for that point is calculated, and so on. This is a fast method to choose a path. The equation (17) shows the calculated virtual force VF in a point x for an object in the position o_i with a factor w_i. If the factor w_i is positive, then the virtual force is repulsive and if the factor is negative, then the virtual force is attractive.

$$VF(x) = \frac{W_i \cdot (x - o_i)}{|x - o_i|^2} \quad (17)$$

Virtual forces of lines and rectangular areas are calculated as the 1D and 2D integrals of the equation respectively.

In Figure 7 is shown an example of potential field. In this case, two points are used, one attractive and the other repulsive, as well as one repulsive line. The chosen path from any point will follow the corresponding arrows (each arrow is a force evaluation).

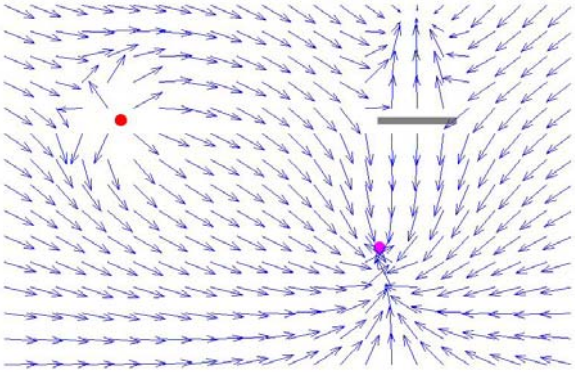

Figure 7. Virtual forces generated by an attractive point (violet), a repulsive point (red) and a repulsive line (gray).

5.3. Behaviors

Behaviors are aimed to select a given action to be executed by the robot. In our implementation, there are three kinds of behaviors: low-level behaviors, complex behaviors and high-level behaviors.

5.3.1. Low-Level Behaviors

Low-level behaviors are the simplest behaviors, they do not use sensorial information and they have a direct connection with the action module. Implemented low-level behaviors are:

- **Stand Still**. This behavior just keeps the robot in a stable position with its four legs on the floor.

- **Turn Continuously**. This behavior makes the robot turn continuously clockwise facing the head with the movement; this behavior does not involve any displacement.
- **Walk**. This behavior generates a walk with a given displacement.
- **Kick**. This behavior selects the best kick to approach the ball to a given objective.
- **Switch Motion State**. This behavior switches between the normal actions and the actions executed when dribbling the ball. The dribbling the ball state main function is to provide movement, even displacement or kicks, while the ball is held with legs and head. Obviously, the held ball constriction makes the dribbling ball state to have a lower motion performance than normal state.
- **Move the Head**. This behavior just move the head to given angles.
- **Look an Objective**. This behavior calculates the necessary head joints angles to look an objective on a given relative position and move the head to them.
- **Ellipse**. This behavior moves the head on an elliptical way.
- **Modified Ellipse**. This Behavior moves the head following the trajectory resulting from the merge of two ellipses: one vertical and the other horizontal, generating a movement that in the higher zone is like an ellipse, but in the lower part it looks to the ground.

5.3.2. Complex Behaviors

Complex behaviors are basically decision trees, which, considering the inner robot state, sensorial information and the information received from teammates, choose the low-level behavior needed to accomplish the desired complex behavior. The complex behaviors implemented are:

- **Localization Help**. This behavior measures the confidence value of the self-localization module and chooses between two low-level behaviors: “Look an Objective”, in which the objective is one of the landmarks present on the field, and “Ellipse” to search landmarks when the confidence value is too low.
- **Go to an Objective**. This behavior chooses a path to achieve a specified objective and chooses the appropriated walking parameters (step length and walking speed) in each moment. This behavior can use potential fields in order to choose the path. Potential fields used by this behavior are flexible and may contain a list of any finite length of objects that produce potential fields; currently the allowed objects are points, rectangles and lines in any direction. This behavior can trigger the low-level behaviors “Stand Still” and “Walk”.
- **Ball Tracking**. This behavior chooses the appropriated low-level behaviors to look at the estimated ball position. It can choose between “Look an Objective”, “Move the Head” and “Stand Still” low-level behaviors.
- **Go to the Ball and Kick it to an Objective**. This behavior is on the limit of being a high-level behavior because of its complexity. Its objective is to select

the corresponding low-level and complex behavior to tracking the ball, walk to the ball and once the ball is accessible, kick it to a specified objective, typically the rival goal. This behavior can trigger the “Ellipse”, “Modified Ellipse”, “Kick”, “Switch Motion State”, “Look an Objective” and “Stand Still” low-level behaviors and the “Go to an Objective” and “Ball Tracking” complex behaviors.

5.3.3. High-Level Behaviors

High-level behaviors are directly related to a specific state. Thus, each robot state has a corresponding high-level behavior. High-level behaviors are decision trees used for choosing the best complex or low-level behavior to be executed. The implemented high-level behaviors are:

- **High-Level Go to Ball and Kick**. This behavior always use the complex behavior “Go to the Ball and Kick it to an Objective”, changing its objective between the rival goal, a teammate (for passing) and the furthermost point from the own goal (for defending the own goal).
- **Look Kick Target**. This behavior always use the low-level behavior “Look an Objective”, setting the objective as the same place as the objective of the last kick of any teammate.
- **Position in the Field**. This behavior uses the potential fields to place the robot on the field. The potential fields used are repulsive for the teammates, attractive for the ball, attractive for a positioning object (defensive line for defender player, player balancer for supporting player and ball-goal alignment for attackers) and repulsive for the field border. This behavior can trigger the complex behaviors “Go to an Objective” and “Ball Tracking”.
- **Goalie Position**. This behavior uses potential fields to place the robot in the goal. Potential fields used include a strong attractive point at the goal and several repulsive lines corresponding to the field border. This behavior can trigger the complex behaviors “Go to an Objective” and “Ball Tracking”.
- **Field Search Ball**. This behavior performs a progressive search for the ball, first it looks around the place turning 360 degrees, then it switch between two given points of the field, at each of them it looks around the place turning 360 degrees. This behavior can trigger the complex behavior “Go to an Objective” and the low-level behaviors “Turn Continuously”, “Modified Ellipse” and “Stand Still”.
- **Goalie Search Ball**. This behavior just positions the robot in the goal and search for the ball using only the head. This behavior can trigger the complex behavior “Go to an Objective” and the low-level behavior “Modified Ellipse”.

5.4. Roles

The possible roles of the players are Attacker, Supporting Attacker, Defender, and Goalie. Only the goalie has a fixed role. The others players may change their roles depending on their position on the field mainly. Employed rules for changing roles are the following:

- The defender cannot become directly an attacker, or the reverse; they must first take up the intermediate position of being a supporting attacker.
- The defender becomes a supporting attacker if it is no longer the last player, i.e. the nearest to his own goal.
- The supporting attacker becomes a defender if it finds itself located nearest to its own goal.
- The supporting attacker becomes an attacker if it goes for the ball while being the nearest to the opposing team's goal.
- The attacker becomes a supporting attacker if another player is going for the ball, and that other player took the position of an attacker.

This system should not allow two players to take up the same position (for any lapse of time), but if they would, it would not affect the decisions regarding state-changing, as the three field players share the same state-system. In Figure 8 is shown the role transition diagram.

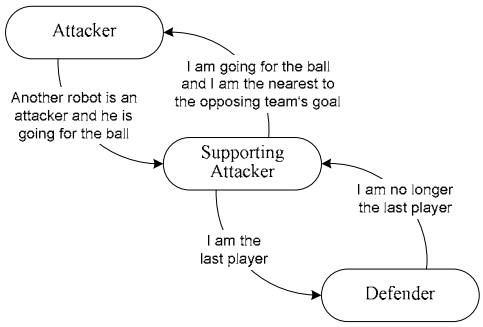

Figure 8. Role transitions allowed for the field players.

5.5. States

Each role is a state-machine with several states and transitions, described in the following paragraph.

5.5.1. Goalie Strategy

The goalie strategy is composed by five states (see Figure 9):

- **Searching the ball**. The goalie walks to the middle of the goal, and searches for the ball moving its head. Once the goalie has reached the middle of the goal, it stops moving the body and only moves the head. This state executes the high-level behavior "Goalie Search Ball".
- **Defending goal, ball seen**. The goalie walks, placing itself on the line between the goal and the ball, 70cm from the goal. This state executes the high-level behavior "Goalie Position".
- **Defending goal, ball not seen**. The goalie places itself along that same line, but nearer the goal, covering the case when the ball position estimation is inaccurate. This state executes the high-level behavior "Goalie Position".
- **Going to ball**. The goalie goes to the ball, using its own visual information and the information received from its teammates. When it gets close enough to the ball, it uses its own visual information only, and tries to hit the ball. It always tries to reach the ball from the closest side to its goal, in order to avoid blunders leading to home goals. When kicking the ball, the goalie aims directly ahead of where the ball is. This state executes the high-level behavior "High-Level Go to Ball and Kick".
- **Looking to objective**. If the goalie kicks the ball, it looks in the direction where the ball should be going, and then, after a short time (about two seconds), it goes to the search ball state. This state executes the high-level behavior "Look Kick Target".

The transitions between the states of the goalie are specified by the next conditions:

- **Ball lost**. The ball is "lost" when none of the team players knows where the ball is.
- **Ball found**. This means a player (any except the player itself) sees the ball, which position is received through the communication network.
- **Ball kicked**. This condition is true immediately after the goalie has kicked the ball.
- **Ball seen**. In this case, it means the goalie sees the ball.
- **After short time**. There is an automatic state change after about 1 second.
- **Ball reachable**. This condition is a complex one. The goalie has a lined area around the goal where no defender can enter. If the goalie sees the ball, and estimates its position within the area or just outside (within 10cm), then the condition is true and the goalie goes for the ball. We also consider that the goalie can leave its area, but only in special circumstances, and not too far away. It will consider that the ball is reachable and it will go for it if it is at least 50cm closer to the ball than any other player, and if the ball is within

130cm from the goal. This is the second way that the condition we mark here as “ball reachable” becomes true.

- **Ball unreachable**. This actually means that the previous condition becomes false. The condition, however, cannot interrupt the goalie if it is attempting to hit the ball.

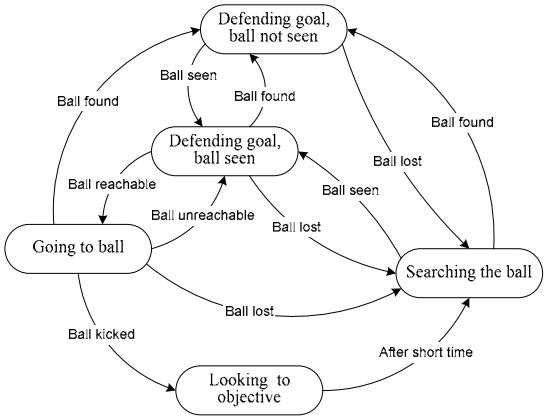

Figure 9. Transition-state map for the goalie.

5.5.2. Field players strategy

The Field players strategy is composed by four states (see Figure 10):

- **Searching ball.** The players look around to see if they can find the ball, and then rotate, looking all around, and if they still do not find the ball, they will go to search in two predefined locations on the field. First, they walk to the first of those two positions, and then they go to the other, and move back and forth until they find the ball. Of course, with all four players searching, the search rarely lasts long enough for the players to use those predefined positions. This state executes the high-level behavior “Field Search Ball”.
- **Positioning.** This state is the one for playing without the ball. The idea is to spread the team on the field, and for this purpose, virtual potential fields are used. They repel the players from the edges of the field, from the defense zone, and from the other players. In Figure 11 is shown the virtual forces present on the attacker while positioning. An additional virtual potential field attracts the defender towards the line between the goal and the ball. This state executes the high-level behavior “Position in the Field”.
- **Going to ball.** Same as for the goalkeeper, except that the kicks are always aimed at the opposite goal. This state uses potential fields to choose the path for approaching to the ball with an appropriated alignment. In Figure 12 is

shown the virtual forces present while going to ball. This state executes the high-level behavior “High Level Go to Ball and Kick”.

- **Looking to objective.** If any robot kicks the ball, then the robot looks at the direction where the ball should be going, and then after a short time (about two seconds) goes to the search ball state. This state executes the high-level behavior “Look Kick Target”.

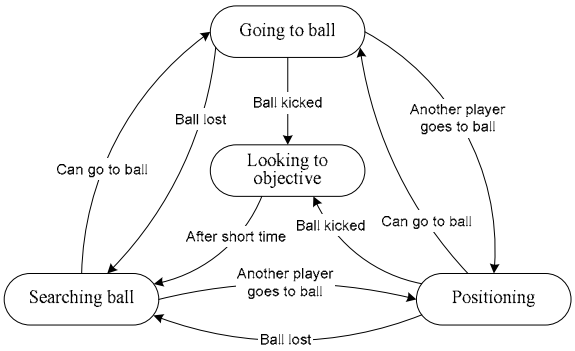

Figure 10. Transition-state map for a field player.

The field players transitions are equal to those of the goalie, except the conditions “Can go to the ball”, and “ball kicked”. The condition “ball kicked”, in this case, includes kicks by other players. The condition “Can go to the ball” almost only depends on who is the closest robot to the ball. If the player estimates that its position is at least 50cm closer to the ball than the next player position, then it goes. If it is the closest, but only slightly, he checks that nobody else is going to the ball in order to go. This does not include the goalie, meaning that another player will go to the ball at the same time as the goalie.

The condition of “another player goes to ball” is actually the opposite of the previous. Another player is closer to the ball and going to play it.

The transitions between the field players states are determined by the next conditions:

- **Ball lost**. The ball is “lost” when none of the team players knows where the ball is.
- **Ball kicked**. This condition is true immediately after any player has kicked the ball.
- **After short time**. There is an automatic state change after about 1 second.
- **Can go to ball**. If the player estimates that its position is at least 50cm closer to the ball than the next player position, then it goes. If it is the closest robot to the ball, but only slightly, it checks that no other field player is going to the ball in order to go.

- **Another player goes to ball**. This actually means that the previous condition becomes false. The condition, however, cannot interrupt the player if it is attempting to hit the ball.

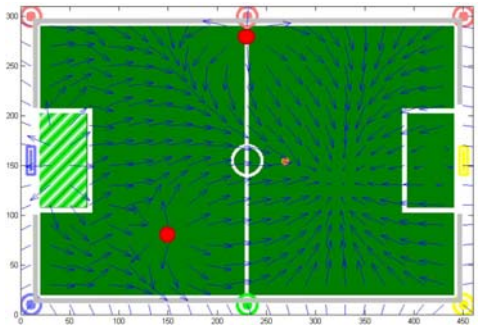

Figure 11. Virtual forces calculated for the attacker while it is positioning. Gray lines represent repulsive lines for repelling from the field edges, red circles represent two teammates generating repulsive fields, an orange circle represents the ball generating an attractive field and a light green square represents the defending area generating a repulsive field.

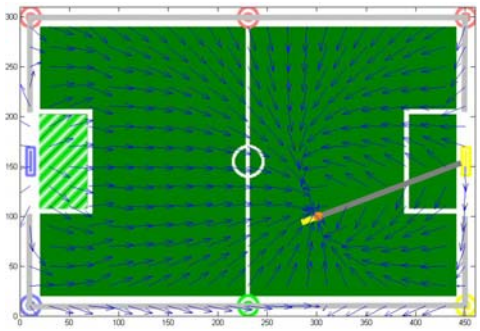

Figure 12. Virtual forces calculated while the player is going to ball. An orange circle represents the ball generating an attractive field, gray lines represent repulsive lines (six for repelling from the field edges and one for repelling the player from the line between the ball and the rival goal). A light green square represents the defending area generating a repulsive field for preventing the player to enter the forbidden area and a yellow line generating an attractive field to align the player on their approach to the ball.

6. UCHILSIM

UCHILSIM is a robotic simulator designed for the RoboCup four-legged league. It was presented on the RoboCup2004 symposium with a big success, obtaining the "*Engineering Challenge Award*" [12]. UCHILSIM is built on top of two processing engines: one in charge of reproducing the dynamics of articulated rigid bodies and the other in charge of generating accurate graphic representations of the objects and the soccer setting. The simulator also includes a variety of interfacing functions which allow the core system to be connected with all the UChile1 controlling modules and sub-systems, as well as to some learning mechanisms. The simulator includes a complete graphic user interface. Figure 13 shows a screenshot of the use of UCHILSIM.

The main design goal of UCHILSIM is to allow robots to acquire behaviors learnt in a representative virtual environment. The long-term goal of the simulator is to allow the generation of complex behaviors for RoboCup team players by efficiently combining the acquisition of knowledge in reality as well as simulations [9].

Most of the rigid body dynamics in UCHILSIM are modeled and computed using the Open Dynamics Engine (ODE) library [13], which is an industrial quality library for simulating articulated rigid body dynamics in virtual environments. ODE is fast, flexible and robust; it allows the definition of a variety of advanced joints and contact points under friction. ODE solves equations of motion by means of a Lagrange multiplier velocity model. It uses a Coulomb contact and friction model which is solved by means of a Dantzig LCP solver method. In UCHILSIM special attention has been provided for the modeling of servo motors and ball.

The graphic representation of the objects in UCHILSIM is obtained by using the Open Graphic Library (OpenGL) [14]. The CAD model of the AIBO robots was generated starting from standard data provided by Sony [15]. The changes made to this model were the incorporation of player's red and blue jackets, the renewal of the robot colors, and the modifications in the original model in order to achieve greater accuracy.

The following is a description of the UCHILSIM main components, such as the basic architecture of the system, the dynamic and graphic engines, the user interface, the learning capabilities of the system and the object loader.

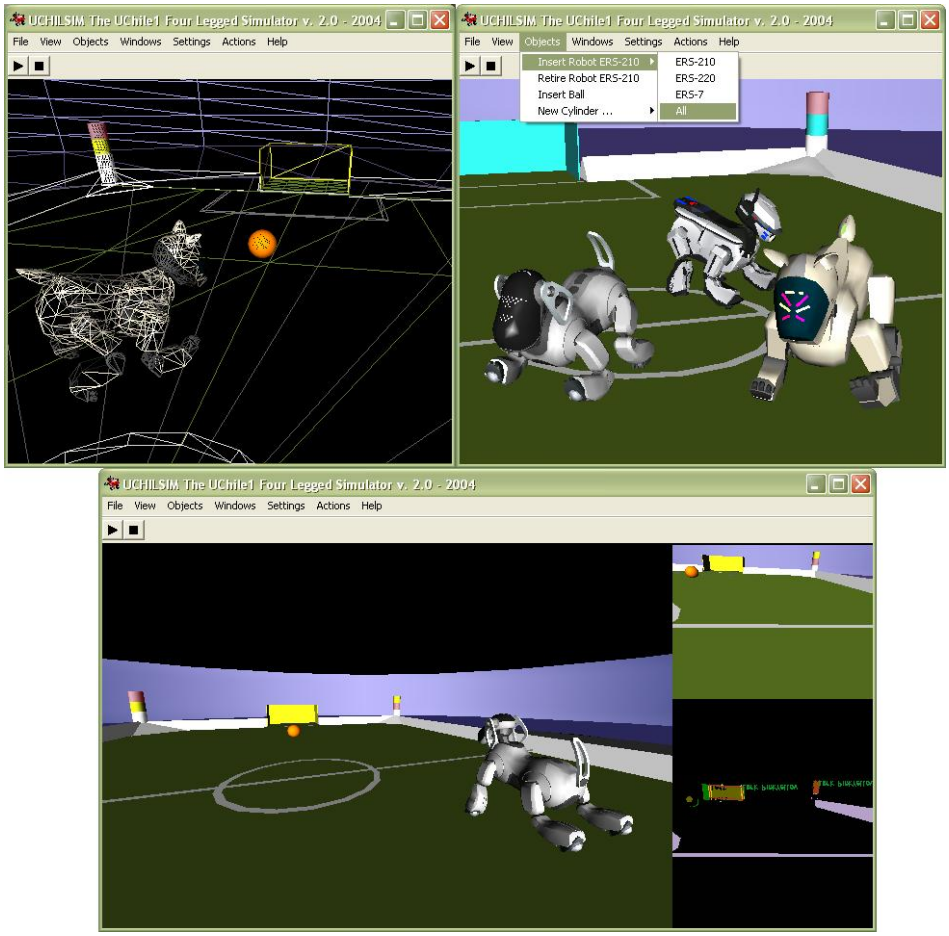

Figure 13. Illustration of the UCHILSIM software and its user interface. It is possible to observe the display of various viewing options.

6.1. Basic Architecture

Figure 14 illustrates the basic architecture of UCHILSIM. It is possible to observe how the core functions of the simulator are interfaced with a variety of applications. In the core of the simulator the dynamic engine closely cooperates with the graphic engine to generate a representation of each object. By means of a learning interface, a set of parameters is interchanged with a learning algorithm which runs independently of the

simulator. These parameters are as a rule for either: defining the simulator variables or the robot controller variables which are being adapted during the learning process. The user interface allows changing several variables of the simulation, such as the way objects are being rendered as well as the external manipulation of objects within the game field. An application interface allows running the overall UChile1 package [16] on the simulation. We are currently working on an Open-r application interface which will allow the compilation of any open-r code under our simulator. We believe that such kind of tool will be quite relevant for the development of the league since it will allow, for example, to realistically simulate a game against any team in the league. Another recently incorporated component is the VRML Object Loader which allows to quickly incorporate new robot models into the simulated environment by following a fast and reliable procedure.

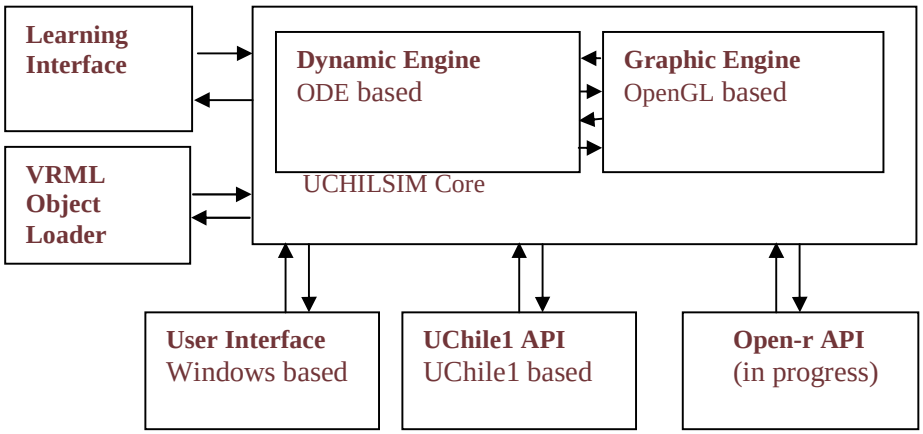

Figure 14. Illustration of the UCHILSIM architecture. It is possible to observe how the overall system is organized as a set of interfaces around a core subsystem which contains the dynamic and graphic engines.

6.2. Dynamic Engine

In UCHILSIM all the AIBO body elements and the soccer field objects are modeled as articulated rigid bodies such as: parallelepipeds and spheres, which are connected to each other using different types of joints. A joint is a dynamic constraint enforced between two bodies, in order that they can only hold certain positions and orientations in relation to each other. We use universal joint models for defining the relationship among the AIBO torso and its thighs, i.e. rotation is constrained to just two degrees of freedom. Simple hinge joints are used for defining the relation among thighs and taps, i.e. constraining rotation to only one degree of freedom. Fixed joint models are used for defining the

relation among the taps and hoofs; in this case one direction of deformation is allowed, and as a result the model accurately represents the rotation of the hoofs. In addition small spheres are attached to the base of each leg by means of fixed joints intended to mimic the effect of the leg tops.

The mass distribution of each rigid body is specified in the form of inertia tensors. For this implementation we assume a uniform distribution of mass for each rigid body. Mass estimation was carried out for each rigid body using a weight measuring device.

Collision detection is performed either with a simple model of spheres and parallelepipeds or by using a simplified version of the graphic grid which is attached to each rigid body. The latter approach is slightly time consuming although more accurate. We have obtained good results in all our experiments by using just the parallelepiped-sphere model. Figure 15 illustrate a diagram of the geometries which are alternatively attached to each rigid body for performing collision detection, it also shows the placement of the servomotors which are included in our model. They apply torque over joints according to the output of a PID controller which receives angular references given by the actuation module of the UChile1 software package [16].

On each simulation step the equation dynamics are integrated and a new state is computed for each rigid body (velocities, accelerations, angular speeds, etc). Then collision detection is carried out, and the resulting forces are applied to the corresponding bodies transmitting the effect of collisions along the entire body. Thus, the friction parameters deserve to be given special attention since they are used for computing the reaction forces between the robot limbs and the carpet. These parameters are under automatic adaptation on each performed experiment.

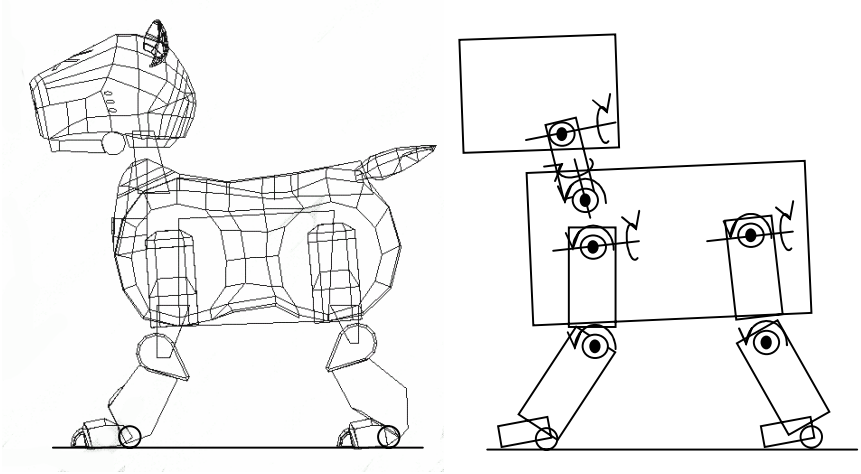

Figure 15. Collision detection models which are alternatively used in UCHILSIM. The figure on the left corresponds to the graphic grid model used for collision detection. The figure on the right corresponds to the model generated with a set of parallelepipeds and

spheres. By using this model we are able to perform accurate dynamic experiments while keeping the simulation at real time speed. The figure also shows the position of the servomotors which are included in the robot model.

6.3. Graphic Engine

The dynamic engine computes the corresponding positions and rotation matrixes of each body in the simulation space at the end of each simulation step. Then, for each rigid body, the corresponding graphic object is rendered. This is carried out by efficiently calling the corresponding graphic data. The graphic engine is also in charge of producing the image acquired from the AIBO's cameras. This is quite relevant for producing experiments with vision based systems. Using this system we will specifically intend to produce an extension of the work presented in [2]. We haven't concentrated our efforts on producing extremely realistic images yet, but we estimate that this process will be simple. We will incorporate some of the transformations which were proposed in [17] such as: the camera distortion and CMOS filters. Using a CAD modeler software we gave the AIBO models blue and red jackets, which were originally provided by Sony, we also constructed the corresponding soccer scenario. The process of importing the graphic data into C++ code was quite time consuming before using the object loader. Currently the graphic data is directly obtained from the object loader module.

6.4. User Interface

The user interface of the simulator currently provides the following set of functions:

1. Loading of arbitrary AIBO models in modified VRML format.
2. Placement, at any moment, of different objects within the simulation, such as: robots, balls and other objects.
3. Arbitrary movement of objects while in motion, this is particularly useful while interactively generating games with the robots.
4. On line Modification of several parameters of the UChile1 controller.
5. Modification of several rendering options and viewing conditions, such as: wire frame representation, bounding box representation, point representation of objects, etc.
6. Management of the images captured by the AIBO's camera. These images can be exported into files or automatically transmitted to some learning software.
7. Loading and saving of a variety of configurations which define the game conditions.
8. Efficient management of several windows on the screen.

6.5. Learning Interface

UCHILSIM is powered with a fast and efficient method for updating its parameters during running time. It is designed to communicate with other programs by means of a TCP/IP network. We have considered this, given that the simulator needs to adapt itself in order to perform experiments with the *Back to Reality* approach proposed in [9]. Using this approach be have carried out experiments for evolve the robot's gait [9] and for learn to kick the ball [10].

6.6. UChile1 and Open-r API

The entire UChile1 software package, which allows a team of fully autonomous AIBO robots to play soccer, can be compiled for UCHILSIM. We had to carry out several modifications on our code in order to make this possible. However, the simulator is a great tool given that, besides its learning capabilities, it is very useful for debugging any piece of code of our system. We are currently working towards generating an Open-r API for the simulator; the idea is to be capable of compiling any Open-r code for UCHILSIM. We believe that there are several applications for such kind of tool, for example, it might speed up the progress of the league by allowing people around the world to develop and test software without the need of having a real AIBO. Thus, incoming research groups might collaborate with the league by testing their code in a simulated environment. For those who already have a real AIBO it might be interesting to test their systems during long evaluation periods, letting teams play against each other during days. This will allow the accurate analysis of the differences among teams and of course the possibility of learning from these experiences.

6.7. Object Loader

The VRML language allows defining objects into a tree like structure of nodes, each one containing graphic as well as structural information of body elements, such as: mass, articulation points, etc. Although graphic data of AIBO models available to the public does not currently contain dynamic information, we have modified them by incorporating mass and motor data into the VRML files. On earlier versions of our simulator the robot models were hard wired into the simulator and the process of incorporating new models was quite time consuming. On the other hand updating a VRML file is considerably less time consuming. Using this technique we have added the ERS-220 and the new ERS-7 AIBO robot models into our simulator.

7. Pointers to relevant publications

Our relevant publications, technical reports, as well as videos and pictures are available in our official website: http://www.robocup.cl [18].

8. References

[1] S. Chen, M. Siu, T. Vogelgesang, T. Fai Yik, B. Hengst, S. Bao Pham, C. Sammut. The UNSW RoboCup 2001 Sony Legged Robot League Team. In: Birk, A., Coradeschi, S., Tadokoro, S, (eds.): *RoboCup 2001: Robot Soccer World Cup V*. Springer-Verlag Berlin, Germany (2002) 39-48.

[2] J.C. Zagal, , J. Ruiz-del-Solar, P. Guerrero and R. Palma, Evolving Visual Object Recognition for Legged Robots, In: *LNAI Proceedings of RoboCup 2003: Robot Soccer World Cup VII*, Springer, (2003).

[3] J. Ruiz-del-Solar, and P. Vallejos, Motion Detection and Tracking for an AIBO Robot Using Motion Compensation and Kalman Filtering. *RoboCup 2004 Symposium*, LNAI (in press).

[4] R. Cucchiara, C. Grana and A. Prati, Detecting Moving Objects and their Shadows - An evaluation with the PETS2002 Dataset, *Proc. of the 3rd IEEE Int. Workshop on PETS*, 18- 25, Copenhagen, June 2002.

[5] J. Piater and J. Crowley, Multi-Modal Tracking of Interacting Targets using Gaussian Approximations, *Proc. of the 2nd IEEE Int. Workshop on PETS*, Hawaii, USA, Dec. 2001.

[6] R. Lastra, P. Vallejos and J. Ruiz del Solar, Integrated Self-Localization and Ball Tracking in the Four Legged Robot Soccer League, *Proc. First IEEE Latin American Robotics Symposium*, 27-29 October 2004, Mexico City, Mexico.

[7] German Team 2003 Technical Report http://www.robocup.de/germanteam/GT2003.pdf.

[8] C. Sammut, W. Uther and B. Hengst, A Description of the rUNSWift 2003 Legged Robot Soccer Team. *Proc. of the RoboCup 2003 Symposium*, July 9 - 11, Padova, Italy.

[9] J.C. Zagal, J. Ruiz-del-Solar, P. Vallejos, Back to Reality: Crossing the Reality Gap in Evolutionary Robotics. *Proceedings of the IAV 2004, 5th IFAC Symposium on Intelligent Autonomous Vehicles*, July 2004, Lisbon, Portugal.

[10] J.C. Zagal, J. Ruiz-del-Solar, Learning to Kick the Ball Using Back to Reality. *Proceedings of RoboCup 2004 Symposium*, LNAI (in press).

[11] P.Vallejos, J. Ruiz del Solar and Alan Duvost, Cooperative Strategy using Dynamic Role Assignment and Potential Fields Path Planning *Proc. First IEEE Latin American Robotics Symposium*, 27-29 October 2004, Mexico City, Mexico.

[12] J. C. Zagal and J. Ruiz del Solar, UCHILSIM: A dinamically and Visually Realistic Simulator for the RoboCup four Legged League, *RoboCup 2004 Symposium*, LNAI (in press).

[13] Smith, Open Dynamics Engine Library, ODE web site available at http://opende.sourceforge.net (2003).

[14] The Open Graphics Library. Available at http://www.opengl.org (2004).

[15] The Open-r Software Development Kit. Available at http://www.openr.org (2003).

[16] J. Ruiz del Solar, J.C. Zagal, P. Guerrero and P. Vallejos, Middleton, C., Olivares, X.: UChile1 Team Description Paper, In: *Proceedings of the 2003 RoboCup International Symposium*, Springer, (2003).

[17] K. Asanuma, K. Umeda, R. Ueda, T. Arai, Development of a Simulator of Environment and Measurement for Autonomous Mobile Robots Considering Camera Characteristics. *Proc. of Robot Soccer World Cup VII*, Springer (2003).
[18] U-Chile-1 Main Publications: http://www.robocup.cl/four/publications.htm