Cerberus'06 Team Report

H. Levent Akın
Çetin Meriçli
Barış Gökçe
Fuat Geleri
Nuri Taşdemir
Buluç Çelik

Artificial Intelligence Laboratory
Department of Computer Engineering
Boğaziçi University
34342 Bebek, İstanbul, Turkey
{akin, cetin.mericli, sozbilir, nuri.tasdemir, fuat.geleri}@boun.edu.tr
buluc_celik@hotmail.com

January 25, 2007

Contents

Acknowledgements

We gratefully acknowledge the support of our work by the Boğaziçi University Research Fund through projects 01A101, 03A101D, and 05A102D, State Planning Organization through Project 03K120250 and Boğaziçi University Student Fund.

Chapter 1

Introduction

The "Cerberus" team made its debut in RoboCup 2001 competition. This was the first international team participating in the league as a result of the joint research effort of a group of students and their professors from Boğaziçi University (BU), Istanbul, Turkey and Technical University Sofia, Plovdiv branch (TUSP), Plovdiv, Bulgaria. The team competed also in Robocup 2002, Robocup 2003, Robocup 2005 and Robocup 2006. Currently Boğaziçi University is maintaining the team. In 2005, despite the fact that it was the only team competing with ERS-210s (not ERS210As), Cerberus won the first place in the technical challenges. This year, we have carried out our success with old ERS-210s to the more powerful ERS-7s by reaching the quarter finals. We lost only three games to the eventual first, third and fourth place teams.

The software architecture of Cerberus mostly remained the same with the last year. All of our modules are platform and hardware independent and our development framework allows us to transfer from or to the robot any input, output or intermediate data of the modules. This infrastructure enables us to have a considerable speed-up during development and testing.

The organization of the rest of the report is as follows. The software architecture is described in Chapter 2. In Chapter 3, the algorithms behind the vision module are explained. The main localization algorithm is given in Chapter 4. The planning module is described in Chapter 5. The Locomotion module and gait optimization methods used are given in Chapter 6. The results concerning the Soccer and Technical Challenge Competitions of the Four Legged League of Robocup 2006 are discussed in Chapter 7.

Chapter 2

Software Architecture

Software architecture of Cerberus mainly consists of two parts:

- Cerberus Station
- Cerberus Player

In the next subsections we describe these parts.

2.1 Cerberus Station

This is the off-line development platform where we develop and test our algorithms and ideas. The whole system is developed using Microsoft .NET technologies and contains a set of monitors which enable visualization of several phases of image processing, localization, and locomotion information. We have included recording and replay facilities. It is possible to record live images, classified images, regions found, perceived objects and estimated pose on the field in real time to a log file and replay it in different speeds or frame by frame. This allows us to test our implementations without deploying the code on the robot each time. Cerberus Station also contains a locomotion test unit in which all parameters of the motion engine and special actions can be specified and tested remotely. For debugging purposes, a telnet client and an exception monitor log parser are also included in the station. Since each sub-module of the robot code is hardware independent, all modules can be tested and debugged in the station. This hardware and platform independence provides great savings on development time when combined with the advanced raw data logging and playback system. Cerberus Station communicates with the robots via TCP and uses a common serializable message structure for information exchange.

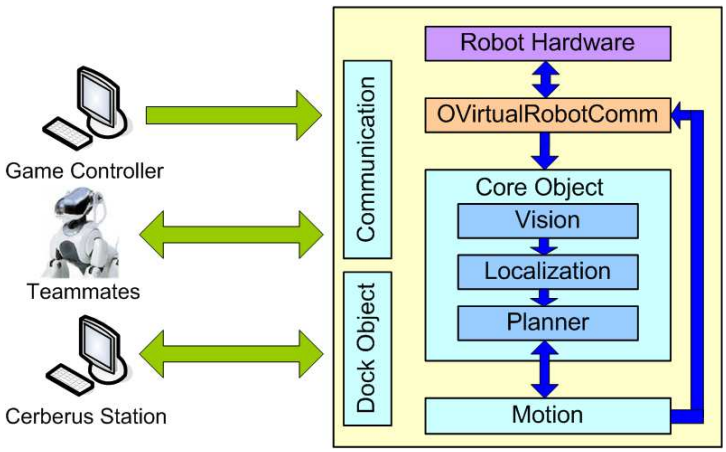

Figure 2.1: Cerberus Software Architecture

2.2 Cerberus Player

Cerberus Player is the part of the project that runs on the robots. Most of the classes in *Cerberus Player* are implemented in a platform independent manner, which means we can cross-compile them in various operating systems like OPEN-R, Windows or Linux. Although, robot dependent parts of the code are planned to run only on the robot, a simulation system for simulating locomotion and sensing is under development. The software architecture of *Cerberus Player* consists of four objects:

- Core Object
- Locomotion
- Communication
- Dock Object

In the following subsections we describe these objects.

2.2.1 Core Object

The main part of the player code is the *Core Object*. This object coordinates communication and synchronization between all the other objects that are connected to it. *Core Object* takes the camera image as its main input and sends the corresponding actuator commands to the locomotion engine. *Core Object* is the container and hardware interface of *Vision*, *Localization* and *Planner* modules. This combination is chosen because of the execution sequence of these modules. All of them are executed for each received camera

frame and there is an input-output dependency and execution sequence that is from *vision* → *localization* → *planner*.

2.2.2 Communication Object

Communication Object is responsible for receiving game data from the game controller and managing robot-robot communication. Both the game controller and robot-robot communication infrastructure use UDP as the communication protocol.

2.2.3 Dock Object

Dock Object is the object which manages the communication between a robot and the *Cerberus Station*. It redirects the received messages to *Core Object* and sends the debug messages to the station. *Dock Object* uses TCP to send and receive serialized messages to and from *Cerberus Station*.

Chapter 3

Vision Module

The Vision module is responsible for information extraction from the received camera frame. Image processing starts with receiving a camera frame and ends with an egocentric world model consisting of a collection of visual percepts as shown in Fig. 3.1.

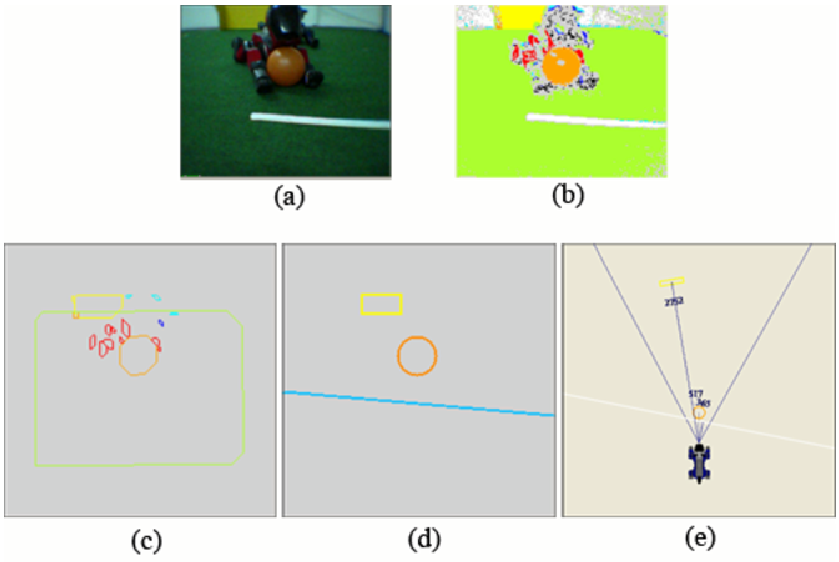

Figure 3.1: Phases of image processing. a) Original image, b) Color classified image, c) Found blobs, d) Perceived objects e) Egocentric view

3.1 Color Classification

Instead of using previously implemented color classification methods like decision trees and nearest neighbor [1], we have implemented a Generalized

Regression Network (GRNN) [2] for color generalization [3]. After labeling a set of images with the proper colors, a GRNN is trained with the labeled data and after the training phase, the network is simulated for the input space to generate a color look-up table for four bits (16 levels) of Y, six bits (64 levels) of U, six bits of V and three bits (eight levels) of the radius. Eight levels for the radius is sufficient, and eventhough it increases the memory requirement of the lookup table from 64KB to 512KB,this is still reasonable. This year, the addition of the radius improved the performance of the color classification table dramatically, as the radial color distortion of the ERS-7 AIBOs is a serious problem for the rest of the vision module.

Having the radius as a paramter can be viewed as having eight separate color classification tables, but providing the radius to the network as input also allows the training of each one of the ring segments affect the others.

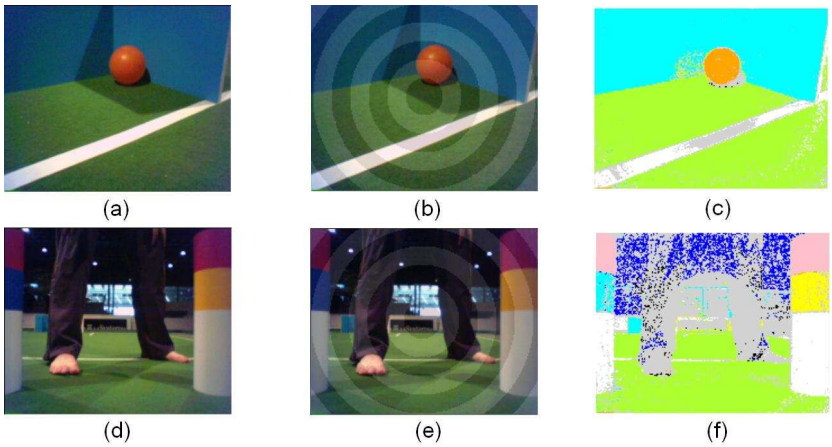

Figure 3.2: The color classification. (a) and (d) Origional Images, (b) and (e) Radius levels, (c) and (e) Classified images

The resultant color lookup table is very robust to luminance changes and allows our vision system to work without using any kind of extra lights other than the standard ceiling fluorescents. Moreover, since no extra operation is required at runtime, no performance losses arise due to radial distortion correction. Fig. 3.2

3.2 Finding Regions

This sub-module is responsible for processing a labeled image and extracting the potentially significant regions for the perception sub-modules. Here we use a different approach. Instead of using run length encoding (RLE), we use an optimized region growing algorithm that performs both connected component finding and region building operations at the same time. This

algorithm works nearly two times faster than the well known RLE-Find connected components-build regions approach.

The approach which uses RLE first runs RLE on the image, connects the runs to find the connected components, filtering out potentially insignificant components, and finally rotating the regions. Fig. 3.3 shows an RLE sample.

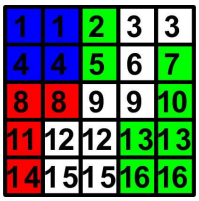

Figure 3.3: An RLE sample

Our approach, on the other hand, uses a region growing algorithm directly on the raw image. Starting from leftmost top pixel, each pixel is processed in the following manner:

- If it was not labeled, it receives a new label
- It is compared with its consecutive pixel on the right. If they are of the same color and the consecutive pixel is labeled with a different label from the pixel which is being processed, then the labels of both pixels are noted to be united. If they are of the same color and the consecutive pixel has no label, then the consecutive pixel is labeled with the same label with the pixel which is being processed.
- It is compared with its consecutive pixel at the bottom. If they are of the same color, then the consecutive pixel is labeled with the same label with the pixel which is being processed.

At this point, the sub-regions and notes indicating the subregions to be combined are prepared in just one pass on the image. Next, the sub regions are combined according to the prepared notes. Fig. 3.4 shows a sample with the new approach.

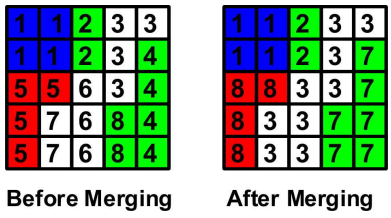

Figure 3.4: A sample with the new approach

After obtaining the combined sub-regions, they are filtered for significant regions and rotated as in other approaches.

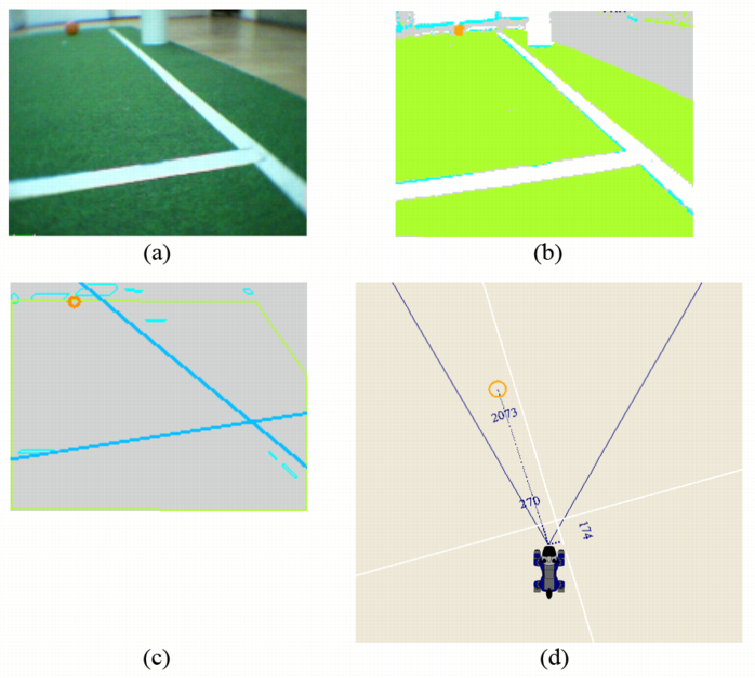

Figure 3.5: Phases of Line Detection. a) Original image, b) Color classified image, c) Perceived lines e) Egocentric view

3.3 Line Perception

Line perception process is an important part of the vision module, since it provides important information for the localization module. The sample images from line perception process are shown in Fig. 3.5. The proposed approach is as follows:

- Hough transform is applied on the white pixels which are close enough to green pixels using Robert's Cross on their Y band as the first operation.
- Two thresholds are used to check each entry in the table prepared in Hough transform. The first threshold is the minimum acceptable value for the line's entry, whereas the second one is minimum acceptable value for the sum of entries of the line and its neighbors. For a line entry to be accepted, it should also be a local maximum.
- For a chosen line, to decrease the quantization error, the weighted average of its angle and the perpendicular distance are taken, where weights are the values in the Hough transform table.

- Now, we have a more or less fine tuned line, but it is still the border of the field line. Especially in images where field lines are close to the camera, the lines occupy a thick region. The selected line is shifted along its normal vector orientation. The amount and the direction of the shift are calculated by following the normal line at different intervals.
- The lines are rotated according to the pan and the tilt of the camera.
- Then, the lines are mapped to the real 3D field using geometrical transformations.
- Once the lines are mapped to the field, they are still relative to the camera. As they need to be relative to the chest of the robot, they are transformed accordingly.
- Finally, extra copies of the same line, which is a rare but possible situation, are eliminated.

3.4 Object detection

The classified image is processed in order to obtain blobs. Here, we use the new approach mentioned in Section 3.2.

3.4.1 Object rotation

Another novel approach used is the concept of a *bounding octagon of a region*. Since the robot must turn its head in order to expand its field of view, it is necessary to rotate the obtained image according to the actual position of the head. However, since rotation is a very expensive operation, it is not wise to rotate the entire image. For this reason typically only the identified regions are rotated. Since octagons are more suitable for rotation than rectangular boxes, using octagons instead of boxes to represent regions reduces the information loss due to rotation.

3.4.2 Ball identification

Our vision module employs a very efficient partial circle fit algorithm for detecting partially occluded balls and the balls which are on the borders of the image as shown in Fig. 3.6. Since accuracy in the estimation of ball distance and orientation is needed mostly in cases where the ball is very close, and often the ball can only be seen partially in such cases, having a cheap and accurate ball perception algorithm is a must. The equation for estimating the ball radius is

$$r = \frac{h}{2} + \frac{s \times s}{8 \times h} \tag{3.1}$$

where r is the radius, h is the height of the ball region and s is the width of the ball region. Although this estimation can be used for different types of ball segments (i.e ball segments in different parts of the captured image), the ball center estimation requires separate handling of different partial image conditions.

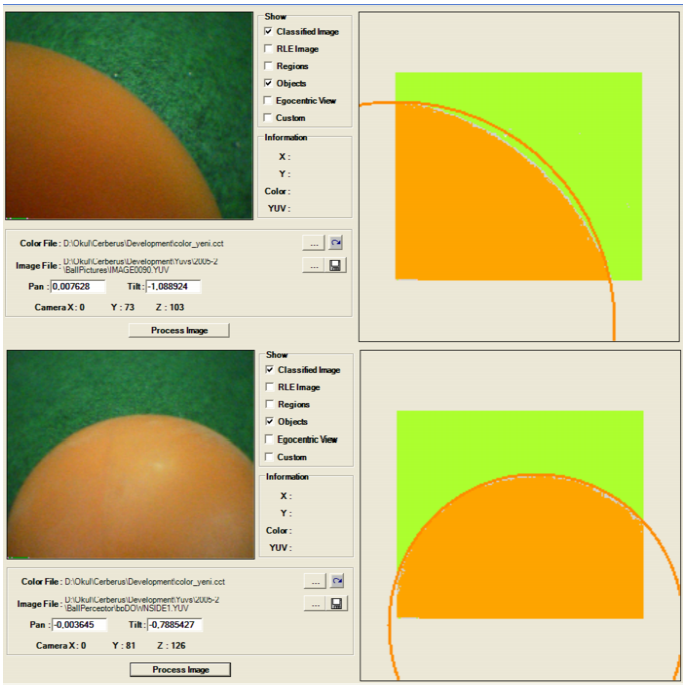

Figure 3.6: Two examples of detecting partial balls via circle fit

3.4.3 Sanity checks

The recognition of objects on the field is based on objects' colors and sanity checks performed on the candidate regions. The sanity check process has three phases.

- **Phase 1.** The candidate region should satisfy object specific precondition checks. For example, the lower edge of the goal or the lowest point of the ball should not be outside of the field region. Of course this control requires a field region (i.e. a merged green region classified as the field) and a threshold value which can be modified for each object type.

- **Phase 2.** A probability value is assigned for the candidate field. The probability calculation is based on the properties of the object. For

example, distance between regions, with respect to the region sizes, has an important effect on probability of being a pair of beacon regions.

- **Phase 3.** Some postconditions checks are performed on the surviving candidate regions. The first postcondition check is usually the probability thresholds which prun

The final checks for each object is well-documented in the source codes which can be accessed from our team's web page.

The vision module is one of the fastest vision systems having the features described above developed on AIBOs. On our previous robots (ERS-210 with 200 MHz processor) a frame was processed in approximately 50 ms which provides a 20 frames per second speed.

Chapter 4

Localization

Localization is one of the main research interests of our research group. We have a number of different localization engines [4, 5, 6, 7, 8]. This year, again we have used *S-LOC*, which was our main localization engine in 2005 [3]. In this section, *S-LOC* and history based egocentric world modeling approach called *My Environment* are presented.

4.1 My Environment

My Environment was initially designed as a part of the localization module and aimed to increase the performance of localization. It was then decided to be a separate module such that not only the localization module, but the other modules could also benefit from its output.

For a human to predict his/her pose, i.e. his/her coordinates and the orientation, vision is the primary input. By estimating the distance and the orientation with respect to known static objects, one can calculate his/her pose. These estimates are valid for not only when they are seen, but also for a period of time after they were perceived, with having the estimates' confidence decreasing in time.

The buffering of objects in the environment can be done either with their actual poses or their poses with respect to the observer. For buffering the actual poses of objects in time, the coordinates of the objects with respect to the environment are calculated using the current perceptions, and stored in an array of data structures as shown in Figure 4.7.a. The odometry update and the instantaneous pose calculations are very simple and could be done at a low cost.

For buffering the relative poses of the objects, the perceived distances and relative angles are directly stored in an array of data structures which are used as buffers as shown in Figure 4.7.b. This way, the odometry update and the instantaneous pose calculations are relatively more complex, and since they require trigonometric functions, the cost is higher.

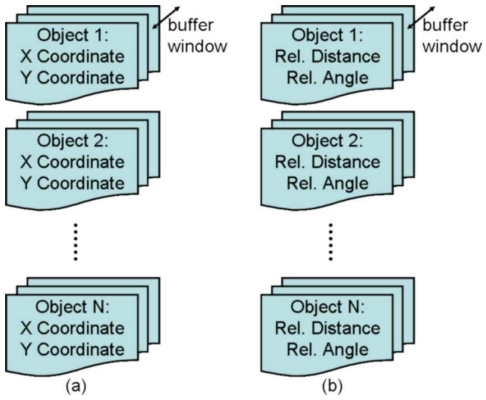

Figure 4.1: Buffering the poses of objects: (a) Buffering actual poses, and (b) Buffering the relative poses

Although the cost of buffering relative poses of objects is greater than the cost of buffering actual poses of objects in time, buffering relative poses is more robust since it does not involve localization. Involving localization in the calculation results in involving localization error in the output. For each stored value on an object's pose history, having instantaneous localization error added to the perception error, the estimations on the current pose of the object would suffer from more noise.

In addition, buffering the relative poses of the objects makes it meaningful to buffer the poses of the static objects. This is very valuable for localization in two ways. First, the processed static object poses would be more robust and lead to more accurate agent pose estimations. Secondly, this buffering will make it possible to process more static objects than that are seen in any moment of time, as long as the buffered relative poses could give a proper estimate for the current pose of the static object.

4.1.1 General Outline of ME

My Environment (ME) is a module between the perception module, or any other module that handles the perception of the environment objects, and the other modules that use the output of the perception module as shown in Figure 4.8. In some exceptional cases, where the position and the relative angle data are not sufficient, the perception data may be needed to be used directly. For instance, in the robot soccer domain, the ball tracking behavior for the head, where the coordinates of the perceived ball region on the camera frame image may be used to calculate the next pan - tilt parameters, uses the perception output directly.

Localization is one of the modules that requires the perception data. During the perception update of the localization module, the perceived static objects are used to estimate the current pose of the agent.

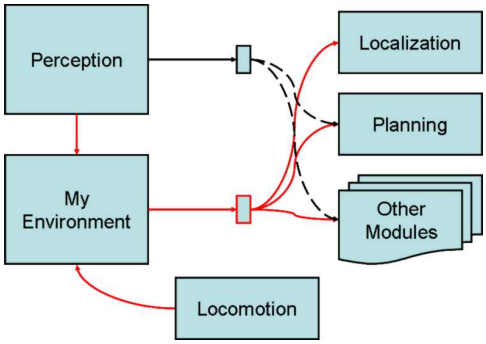

Figure 4.2: Interaction of ME with the other modules

The inputs of the perception module are the current internal state of the agent and the latest camera frame image. Input from the camera is very noisy most of the time and may cause false perceptions. Not only the distance of an object could be perceived erroneously, but sometimes an object itself could be recognized as another object. If the data from the perception module are used as they are, these errors could result in unwanted behavior in other modules.

By filtering the output of the perception module and using the past perceptions of the objects at the same time, more stable and robust data could be provided for the localization module as well as other modules. This way the effect of false perceptions and recognitions would be decreased.

It should be kept in mind that for a mobile agent, using the past perceptions can lead to problems if the motion of the agent is not reflected on the past perceptions.

4.1.2 Architecture of ME

There are two kinds of objects in the ME architecture: static objects and dynamic objects. Each object, either static or dynamic, has a buffer window for storing the most recent perception data for that object, and an additional buffer for the current estimation for that object. The data structure of ME is shown in Figure 4.9.

Each static object entry has a distance, a relative angle and a confidence regarding its perception. In the case when a static object has its own orientation, the orientation values are also to be stored in the buffer. For dynamic objects, the velocity should also be calculated, but as it is not directly extracted from a single camera frame, it is not necessarily be buffered. In Figure 4.10a and 4.10b the data structures for static and dynamic objects are defined.

The window size is a hyper-parameter of ME. The noise in odometry, the dynamicity of the agent's pose, and the frequency of the processed vi-

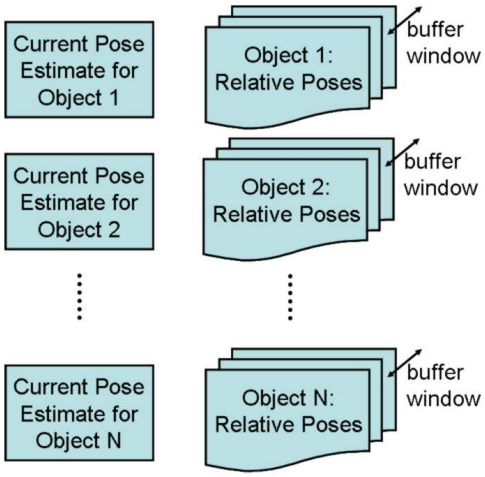

Figure 4.3: The data structure of ME

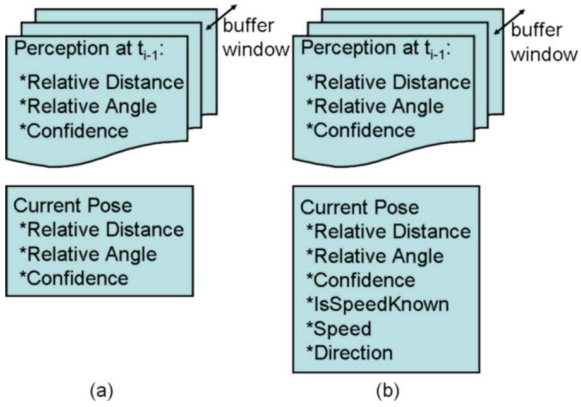

Figure 4.4: Data structures of (a) static objects, and (b) dynamic objects

sion frames play an important role in the selection of a good window size parameter. For dynamic objects, the speed of such objects should also be taken into account.

For instance, for Cerberus'05, the robots used have a maximum speed of approximately 30 cm/s (with ERS 210s), the odometry is quite noisy as the robots are legged, and the average vision frame processing performance was 18 fps. For such conditions, the size of windows for static objects and dynamic objects were chosen to be 32 and 12 respectively.

4.1.3 Procedures of ME

There are five main procedures of ME. Other than *initialization*, the first two procedures, *perception update* and *odometry update*, are triggered as new perception data from the perception module and new odometry data from the locomotion module are obtained. The other two, *current pose estimation for static objects* and the *current pose estimation for dynamic objects*, are called inside the *perception update* procedure.

Initialization

As an initialization, it is necessary for the pose estimations to set all the buffers in the windows of all the objects, for both the static objects and the dynamic objects. If there is no prior information about the pose of an object when the system has started, all the buffers in its window are to be marked as unknown. If the initial pose of an object is known a priori, then this can be provided to the system by filling the buffers in its windows according to that knowledge.

Perception Update

For each dynamic and static object, the oldest record on the buffer is deleted and a new record is stored from perception if the object is seen at the moment, otherwise it is marked that there is information available about the pose of the object at that time. After updating the buffer with the latest perception output, an estimation is done for each object concerning its pose by calling the appropriate procedure in Section 4.2.7 or Section 4.2.7.

Odometry Update

For the odometry update of each record, three trigonometric functions and a square root is used. For a ME with n_o number of objects and a window size of n_w, there are $n_o \times n_w$ number of records. These calculations increase the cost of ME, but they are mandatory for reasonable ME estimations.

Since all the information in the buffers of all objects' windows is relative to the agent, on each movement action of the agent, they need to be modified. For each record new relative distance and the relative angle values have to be calculated using Equations 4.44 and 4.45.

$$c = cos(\theta) \times d - \Delta x \tag{4.1}$$
$$s = sin(\theta) \times d - \Delta y \tag{4.2}$$
$$d' = \sqrt{c^2 + s^2} \tag{4.3}$$
$$\theta' = tan^{-1}(s/c) + \Delta\theta \tag{4.4}$$

where θ, d, Δx, Δy and $\Delta\theta$ are the previous relative angle, previous relative distance, the signed distance the agent moved in sideways, the signed distance the agent moved on its orientation and the angle the agent has turned, respectively. The new relative distance and the new relative angle are represented with d' and θ' respectively.

Current Pose Estimation for Static Objects

For each static object, this function is called once in every *perception update*. Using the window sized perception data records; a pose estimation is made for its use in other modules of the agent's architecture like localization and planning.

In Equations 4.50, 4.51 and 4.52 the confidence estimation, the relative distance and relative angle of the static object are calculated.

$$w_j = \sum_{i=0}^{n_w} KA_i^j \times CA_i^j \times f_{ws}(i) \tag{4.5}$$
$$\Delta x_j = \frac{\sum_{i=0}^{n_w} KA_i^j \times DA_i^j \times cos(AA_i^j) \times f_{ws}(i)}{w_j} \tag{4.6}$$
$$\Delta y_j = \frac{\sum_{i=0}^{n_w} KA_i^j \times DA_i^j \times sin(AA_i^j) \times f_{ws}(i)}{w_j} \tag{4.7}$$

$$n_j = \sum_{i=0}^{n_w} KA_i^j \tag{4.8}$$
$$c_j = \frac{\sum_{i=0}^{n_w} KA_i^j \times CA_i^j \times f_{ws}(i)}{w_j} \times f_{wc}(n_j) \tag{4.9}$$
$$d_j = \sqrt{\Delta x_j^2 + \Delta y_j^2} \tag{4.10}$$
$$\theta_j = tan^{-1}(\Delta y_j / \Delta x_j) \tag{4.11}$$

where j is the index of the object, n_w is the window size; w_j is the total weight for the j^{th} object, $f_{ws}(i)$ gives the weight of the i^{th} record for a

static object; $f_{wc}(n_j)$ gives the weight for the confidence of an object with n_j known records in its windows; KA_i^j is a flag which is equal to one if the i^{th} record of the j^{th} object exists (i.e. the object was perceived at the time that record was buffered) and zero otherwise; DA_i^j is the distance of the i^{th} record of the j^{th} object; AA_i^j is the relative angle of the i^{th} record of the j^{th} object; CA_i^j is the confidence of the i^{th} record of the j^{th} object; c_j is the confidence estimation of the j^{th} object; d_j is the relative distance estimation of the j^{th} object; and θ_j is the relative angle estimation of the j^{th} object.

The function $f_{ws}(i)$ is a monotonically increasing function. The value of $f_{ws}(i)$ is to be arranged such a way that the more recent a record it is the more weight it will receive, but at the same time it will not let too small number of records (i.e. one or two) dominate the value of the weight. Although a linear function could easily be used for that purpose, a sigmoid function was expected to give better results if configured properly for the application.

The function $f_{wc}(i)$ is also a monotonically increasing function for favoring the confidence with respect to the number of records of which poses are available.

If n_j is zero, meaning that none of the buffers in the window stores a perceived pose, then no estimation could be made and the object's pose is set as unavailable. After the confidence is calculated, if it is below a predefined threshold, the object's pose is also set as unavailable.

Current Pose Estimation for Dynamic Objects

The procedure of the current pose estimation for the dynamic objects is the same as the current pose estimation for the static objects except that for dynamic objects the speed and the direction of the speed should also be calculated when possible. For each dynamic object, this function is called once in every *perception update*. Using the same equations in Section 4.2.7 the pose estimation, with the exception of the speed related variables, is performed, but the $f_{ws}(i)$ function is replaced with $f_{wd}(i)$, which gives the weight of the i^{th} record for the dynamic object.

If an object is dynamic, perceiving the object in different poses may be either due to noisy perception or the object's movement. If the object's recent poses are buffered and used for the estimation of the current pose, the weights of the most recent records should be higher than they are for static objects, which should always be perceived in the same pose. As a remark, it should be noted that the effect of the movement of the agent is eliminated with the motion update procedure.

The function $f_{wd}(i)$ is also a monotonically increasing function as the function $f_{ws}(i)$ is, but favors the most recent records more than $f_{ws}(i)$. Since the older records are less important for the dynamic objects, the window size for the dynamic objects could be set smaller than the window size set for

the static objects.

The speed of a dynamic object is estimated only if the pose of the object was available from the previous run, otherwise the speed is set as unavailable.

In Equations 4.54 and 4.55 the speed and relative direction of the speed of the dynamic object are calculated.

$$h_j = d_j^2 + d_{j-1}^2 - 2 \times d_j \times d_{j-1} \times cos(\theta_j - \theta_{j-1}) \quad (4.12)$$

$$\alpha_j = \Pi - cos^{-1}(\frac{h_j - d_j^2 + d_{j-1}^2}{2 \times h_j \times d_{j-1}}) - \theta_{j-1} - \theta_j \quad (4.13)$$

$$s_j = \frac{\sqrt{h_j}}{\Delta t} \quad (4.14)$$

where j is the index of the object, d_j is the relative distance estimation of the j^{th} object; θ_j is the relative angle estimation of the j^{th} object; α_j is the relative direction of the speed estimation; and s_j is the speed estimation.

4.1.4 Advantages and Disadvantages of ME

ME provides more stable results for both static and dynamic objects. For static objects, especially in the case when localization does not give accurate results, the pose of the static object at ME would be more robust. For localization, the ME output poses can be used as if they were perceived from the sensors at that time. In this way, perceived objects are not forgotten just after they are perceived, but remain in ME for a specific period of time. In addition, the noisy perception, which from time to time may lead to false object detections, could be stabilized.

ME provides more stable results for dynamic objects as well. Using the ME output instead of the perception output directly, instantaneous fluctuations in the pose of the object are smoothened. Losing the dynamic object in some camera frames and perceiving it again frequently, which is not a rare thing in robot soccer, could lead to oscillations in the operations and the outputs of some of the modules. ME smoothens these oscillation with its pose estimation, where it uses the recent perceptions to calculate the current pose.

These advantages have a cost. The space needed to store the pose buffers and current estimations of objects in ME grows linearly with the product of window size of the pose buffers and the number of objects in ME. The complexity of the ME procedures is $O(n_w \times n_o)$, where n_w is the window size and n_o is the number of objects in ME. Using the output of the perception module, both the processing power and memory expenses of ME will be saved, but if the system can afford these expenses, the benefits of ME could be worthwhile.

It should also be noted that using ME, the agent would observe dynamic objects slower than they are. This is because of using the previous poses of

the dynamic object in the calculation of the current pose. This problem could be minimized theoretically by adding the velocity of the previous estimation times the time passed to the previous records of that object, but this could bring more noise than it makes corrections as the velocity estimations could be noisier than the relative position estimations.

4.2 S-LOC: Simple Localization

During the development of the localization module of Cerberus'05, many techniques were taken into consideration.

- Triangulation is a simple and accurate technique, but is not robust. It is too much effected by noise, specially by the false perceptions [9].
- The major disadvantage of Kalman Filter methods is that they do not have the capability of recovering from kidnapping [12, 13].
- ML approaches are generally expensive, where false perceptions could be big problems [14, 15].
- Raw MCL cannot recover from kidnapping, but a version of it, SRL is implemented [16, 17, 18].
- ML-EKF is also another expensive technique, which would not be preferred in a case where a much lower cost algorithm could give accurate and robust results [19, 20].
- Fuzzy localization techniques generally have high computational complexity, and do not give results with enough accuracy that are worth the cost [10, 11].
- R-MCL is also another technique, which is used in the experiments for comparison purposes [4, 5, 6].

Considering the points above, it was decided to implement *S-Loc* together with a version of SRL. The existing R-MCL module implemented in our laboratory is also used in the experiments.

In general, the localization process has two main steps. The first one is the perception update, which is based on the perceptions in order to calculate the estimated pose of the agent. Since the movement of the agent changes its pose, the second step, the odometry update, is necessary for reflecting the effect of the movement on the calculations and the estimations.

Perception update, as it depends on the perceptional information, usually includes high amount of noise. Although the agent is dynamic, its pose should not be highly unstable, i.e. the pose should not jump to different

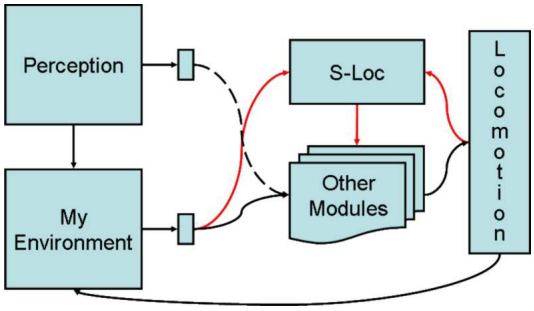

Figure 4.5: The relationship of the S-Loc module with the other modules

poses that are far away on the field frequently. Using a memory for the previous pose estimate, and updating it with the current estimate could handle the big fluctuations and increase the robustness to the false perceptions of static landmarks.

In order to use triangulation, three objects, which are not available at the same time frequently, are needed to be perceived. Also, even if three objects are available, in the case where one of the perceptions is wrong or is highly noisy, the calculation will lead to a very noisy pose estimation.

In the MCL, there is a large number of sample poses, for which many calculations should be made in order to find their confidences. Generally, most of these samples do not hold any useful information. Also, noisy perception data may lead to unstable pose estimations.

The principle of ML leaves open how the robot's belief is represented and how the conditional probabilities are computed. Existing ML approaches mainly differ in the representation of the state space and the computation of the perceptual model. These approaches are generally expensive, since the space is discretized and for each perception and for each location, the probabilities should be calculated at each frame. False perceptions could also be major problems.

In the perfect, noise-free case, the odometry data should be continuous and the pose should be updated continuously as the agent moves. On the other hand, in the real world case, the odometry data is generally very noisy, especially when the agent uses legs for locomotion; and arrives at discrete times, for instance after a step is completed. Both of these make the previous pose estimates less confident for the current estimate calculations.

4.2.1 General Outline of S-Loc

S-Loc is a localization module. It needs the perception data and the odometry data for updating the pose estimate, which it provides as the output. This pose estimate is then used in other modules. The relationship of the *S-Loc* module with the other modules is shown in Figure 4.11.

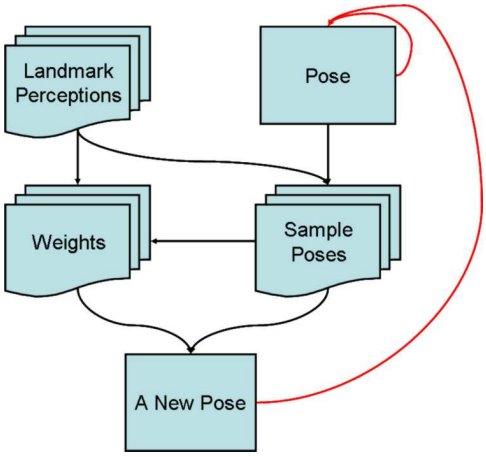

Figure 4.6: The perception update process

The perceived data can be obtained directly from the vision module (or any other perception module), or they can be supplied by the ME module where they are buffered and estimates using them are produced. It could perform better if the perceptional input is provided by the ME module, because the ME module provides more stable and robust data.

The locomotion module provides the odometry data at certain times, which is generally less frequent than the perception data. The effect of the movement of the agent should also be reflected on the pose estimate.

4.2.2 Architecture of S-Loc

The perception update of the S-Loc depends on the perception of landmarks and the previous pose estimate. The perception update process is shown in Figure 4.12. Even if the initial pose estimate is provided wrong, it acts as a kidnapping problem and is not a big problem as S-Loc will converge to the actual pose in a short period of time if enough perception could be obtained during this period.

For each perceived landmark, a sample pose is calculated according to this perception and the previous pose estimate of the agent. The previous pose estimate is also taken as a sample pose.

For each sample pose, using all the landmarks, the likelihood of this sample pose is calculated. This is done by assuming that the agent's actual pose is the sample pose being processed and calculating the difference of the perceived landmarks positions and their actual positions. Also, the confidence of the perception is reflected on the likelihood.

After these likelihood calculations are done for each sample pose, these likelihoods are used for calculating the weights of the corresponding sample poses, and a new pose is calculated as the weighted average of these sample

poses.

The weighted average of these sample poses is then used together with the previous pose estimate to calculate the current pose estimate. The purpose of not using the weighted average of these sample poses is to directly provide the system enough memory to prevent big jumps of the pose estimate and make it more stable.

After the current position of the agent is estimated, it could be safer to calculate the current orientation of the agent using the current position estimation and the perceptions.

In the case of having no perception at a certain time, the current pose estimate could be obtained by decreasing the confidence of the previous pose estimate.

The odometry update process is as simple as updating the pose estimation with the odometry data. Since only the pose estimation is used from the previous cycle of every estimation, no more update or calculation is necessary. On the other hand, if the frequency of the odometry update is much less than the frequency of the perception update, then it may be better to lower the weight of the odometry data accordingly. This is because having the original odometry data to be the result of the motion during more than one perception updates.

4.2.3 Procedures of S-Loc

There are three main procedures of *S-Loc*. The *initialization* is the first one. Other two procedures, *perception update* and *odometry update*, are triggered as new perception data from the perception module and new odometry data from the locomotion module arrive.

Initialization

The only thing to be done in the initialization procedure is to initialize the pose estimate to initial value. It does not have to be the actual pose that the agent will have at the beginning, since *S-Loc* module can recover from kidnapping. On the other hand, it should still be set to a valid pose initially in order not to cause a problem in the proceeding calculations.

Perception Update

As shown in Equations 4.56, 4.57, 4.58, 4.59 and 4.60, the first pose sample is the previous pose estimate.

$$PSA_x^0 = PE_x \tag{4.15}$$

$$PSA_y^0 = PE_y \tag{4.16}$$

$$PSA_\theta^0 = PE_\theta \tag{4.17}$$

$$PSA_c^0 = PE_c \tag{4.18}$$
$$PSA_w^0 = PE_c \times f_{wpu2}(PSA_x^0, PSA_y^0, PSA_\theta^0, PA) \tag{4.19}$$
$$PA_k^0 = 1 \tag{4.20}$$

where PS_x^0, PS_y^0, PS_θ^0, PS_c^0 and PS_w^0 are the x-coordinate, y-coordinate, orientation, confidence and weight of the first pose sample; PE_x, PE_y PE_θ, and PE_c are the x-coordinate, y-coordinate, orientation and confidence of the pose estimate before the perception update; PA, percepts array, is the collection of perception data of all the perceived landmarks together with their coordinates that are known initially; f_{wpu2} is the function that returns a weight component for a pose according to the current perceptions; and PA_k^0 is set to one in order to have the first element of pose sample array included in the proceeding calculations.

Then, a separate pose sample is calculated for each perception as in the Equations 4.63, 4.64, 4.66, 4.67 and 4.68.

$$\alpha_i = tan^{-1}\left(\frac{PE_y - PA_y^i}{PE_x - PA_x^i}\right) \tag{4.21}$$
$$PSA_x^i = PA_x^i + PA_d^i \times cos(\alpha_i) \tag{4.22}$$
$$PSA_y^i = PA_y^i + PA_d^i \times sin(\alpha_i) \tag{4.23}$$
$$\beta_i = tan^{-1}\left(\frac{PSA_y^i - PA_y^i}{PSA_x^i - PA_x^i}\right) \tag{4.24}$$
$$PSA_\theta^i = \pi + \beta_i - PA_\theta^i \tag{4.25}$$
$$PSA_c^i = PA_c^i \tag{4.26}$$
$$PSA_w^i = f_{wpu1}(PA^i) \times f_{wpu2}(PSA_x^i, PSA_y^i, PSA_\theta^i, PA) \tag{4.27}$$

where α_i and β_i are dummy angle variables; PSA_x^i, PSA_y^i, PSA_θ^i, PSA_c^i and PSA_w^i are the x-coordinate, y-coordinate, orientation, confidence and weight of the i^{th} pose sample; PA_x^i, PA_y^i are the actual x-coordinate and y-coordinate of the i^{th} landmark in the percepts array; PA_d^i, PA_θ^i and PA_c^i are the perceived relative distance, relative angle and the perception confidence of the i^{th} landmark in the percepts array; PA^i is the perception data of the i^{th} perceived landmark which is stored as the i^{th} element of the Perception Array; and f_{wpu1} is the function that returns a weight component for a pose according to the perception for which the pose sample is calculated.

The function f_{wpu1} returns the first component of the PSA_w^i for the argument PA^i. It may return PA_c^i directly or any other number that gives the confidence that the perception is correct. Since the accuracy of the pose sample will be taken into account by the function f_{wpu2}, this function is independent of the corresponding pose sample. The purpose of this function is to decrease the weight of the pose samples, of which the perception is less confident. In the case where the perception module does not provide

healthy confidence values, the perceived relative distance of the landmark can be used for the calculation of the return value. In such a case, a properly configured sigmoid function can be very suitable. If the landmarks are of different types and are known to have different perception accuracy, then this could also be reflected on the return value.

The function f_{wpu2} returns the second component of the PSA_w^i. The return value is related to the accuracy of the pose sample according to all the perceived landmarks. For each perceived landmark, the position of the perceived landmark is calculated by adding the perceived distance on the perceived relative angle to the pose sample, and the resulting position is compared to the actual position of the landmark. The difference gives the error. The return value should be a function of the error as in Equation 4.71.

$$par_x^j = \left| PA_x^j - (PSA_x^i + PA_d^j \times cos(PSA_\theta^i + PA_\theta^j)) \right| \quad (4.28)$$

$$par_y^j = \left| PA_y^j - (PSA_y^i + PA_y^j \times sin(PSA_\theta^i + PA_\theta^j)) \right| \quad (4.29)$$

$$f_{wpu2} = \prod_{j=1}^{N_L} PA_k^j \times f_{wpu3}\left(par_x^j, par_y^j\right) \quad (4.30)$$

where N_L is the number of landmarks; PA_k^j is one if the perception of the j^{th} landmark is available, and zero otherwise; and f_{wpu3} is a function that returns a value related to the difference in the x-coordinate and the y-coordinate.

The return value of the function f_{wpu3} is a value for the confidence of the sample pose for the corresponding perceived landmark. The greater the x-coordinate and y-coordinate differences provided as parameter to this function, the worse the sample pose fits to that landmark perception and therefore the less confidence the function shall return.

The calculation of the new pose estimate is the last step of the perception update. Except the new orientation estimate, all the estimation values are the weighted average of the recent calculation, which is in turn a weighted average of sample poses, and the corresponding previous estimate value. The new orientation estimate is calculated by using the new coordinate estimates and the perceptions. The new values of pose estimate are calculated from the Equations 4.74, 4.75, 4.78, and 4.79.

$$hp = f_{HP}\left(\sum_{j=1}^{N_L} PA_k^j\right) \quad (4.31)$$

$$tw = \sum_{j=0}^{N_L}\left(PA_k^j \times PSA_w^j\right) \quad (4.32)$$

$$PE_x^* = hp \times PE_y + (1 - hp) \times \frac{\sum_{j=0}^{N_L} \left(PA_k^j \times PSA_x^j \times PSA_w^j \right)}{tw} \quad (4.33)$$

$$PE_y^* = hp \times PE_y + (1 - hp) \times \frac{\sum_{j=0}^{N_L} \left(PA_k^j \times PSA_y^j \times PSA_w^j \right)}{tw} \quad (4.34)$$

$$\beta_i = tan^{-1} \left(\frac{PE_y^* - PA_y^i}{PE_x^* - PA_x^i} \right) \quad (4.35)$$

$$wa_i = f_{wpu1}(PA^i) \times f_{wpu2}(PE_x^*, PE_y^*, \beta_i, PA) \quad (4.36)$$

$$PE_\theta^* = tan^{-1} \left(\frac{\sum_{i=1}^{N_L} \left(PA_k^i \times sin(\beta_i) \times wa_i \right)}{\sum_{i=1}^{N_L} \left(PA_k^i \times cos(\beta_i) \times wa_i \right)} \right) \quad (4.37)$$

$$PE_c^* = hp \times PE_c + (1 - hp) \times \frac{\sum_{j=0}^{N_L} \left(PA_k^j \times PSA_c^j \times PSA_w^j \right)}{tw} \quad (4.38)$$

where PE_x^*, PE_y^*, PE_θ^* and PE_c^* are the updated x-coordinate, y-coordinate and orientation of the pose estimate; and f_{HP} is a function that returns a history coefficient according to the number of percepts available.

Odometry Update

For the odometry update, the only necessary thing is to update the current pose estimation with the new odometry data. No more update or calculation is necessary, because nothing is used from the previous cycle of estimation other than the pose estimation.

It should also be noted that, in the case where the frequency of the odometry update is much less than the frequency of the perception update, transforming the odometry data to lower values may lead to better results since the original odometry data is the result of the agent's motion from the previous odometry update to the current one, and this would last for more than one perception updates.

In Equations 4.80, 4.81 and 4.82 the new (updated) coordinates and orientation of the pose estimate is calculated.

$$PE_x^* = PE_x + \Delta x \times sin(PE_\theta) + \Delta y \times cos(PE_\theta) \quad (4.39)$$

$$PE_y^* = PE_y + \Delta y \times sin(PE_\theta) - \Delta x \times cos(PE_\theta) \quad (4.40)$$

$$PE_\theta^* = PE_\theta + \Delta\theta \quad (4.41)$$

where PE_x^*, PE_y^* and PE_θ^* are the updated x-coordinate, y-coordinate and orientation of the pose estimate; PE_x, PE_y and PE_θ are the x-coordinate, y-coordinate and orientation of the pose estimate before the odometry update; Δx, Δy and $\Delta\theta$ are the odometry data giving the change in the x-coordinate, y-coordinate and orientation.

4.2.4 Advantages and Disadvantages of S-Loc

In ML, for each landmark seen the probability distribution is modified accordingly, and as a result, the final probability distribution is expected to give the agent's real pose. Instead of a probability distribution, a pose, which is most likely to be the actual pose according to the previous pose estimate, is used in S-Loc. In this way, as it is the case in ML, the pose estimate converges to the actual pose of the agent.

Considering only the most likely sample poses, *S-Loc* acts like a kind of ML but with a local coverage. Although it has a local coverage, it responds in a fast manner to the kidnapping problem, as the most likely sample poses could be far away from the previous pose estimate. In addition, since only a sample pose for each landmark is calculated, S-Loc has a much lower cost than ML.

In comparison with triangulation, S-Loc does not calculate the best estimate according to the perception of the moment, but makes the estimation in a way that it converges to that point in a short period of time. On the other hand, the effect of the false perceptions is greatly decreased as the sample pose of such a perception would have a relatively small confidence and will not play a big role in the pose estimation. In this way, the robustness is increased without decreasing the performance.

In a way, *S-Loc* works similarly as the MCL since the sample poses are used in the same way they are used in MCL. The main difference is the selection of these sample poses. In MCL, there is a large number of pose samples, and they are populated according to their confidences, and randomly mutated for small changes. In S-Loc new pose samples are calculated for each estimation, and a pose sample is calculated for each perceived landmark. In this way, *S-Loc* becomes a much lower cost localization method with accurate pose estimation capability.

The memory used in the *S-Loc* increases the robustness of the system even further and the big jumps of the pose estimate are prevented.

The buffering of objects in the environment can be done either with their actual poses or their poses with respect to the observer. For buffering the actual poses of objects in time, the coordinates of the objects with respect to the environment are calculated using the current perceptions, and stored in an array of data structures as shown in Figure 4.7.a. The odometry update and the instantaneous pose calculations are very simple and could be done at a low cost.

For buffering the relative poses of the objects, the perceived distances and relative angles are directly stored in an array of data structures which are used as buffers as shown in Figure 4.7.b. This way, the odometry update and the instantaneous pose calculations are relatively more complex, and since they require trigonometric functions, the cost is higher.

Although the cost of buffering relative poses of objects is greater than

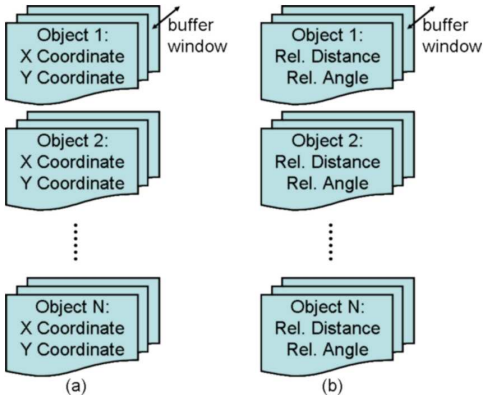

Figure 4.7: Buffering the poses of objects: (a) Buffering actual poses, and (b) Buffering the relative poses

the cost of buffering actual poses of objects in time, buffering relative poses is more robust since it does not involve localization. Involving localization in the calculation results in involving localization error in the output. For each stored value on an object's pose history, having instantaneous localization error added to the perception error, the estimations on the current pose of the object would suffer from more noise.

In addition, buffering the relative poses of the objects makes it meaningful to buffer the poses of the static objects. This is very valuable for localization in two ways. First, the processed static object poses would be more robust and lead to more accurate agent pose estimations. Secondly, this buffering will make it possible to process more static objects than that are seen in any moment of time, as long as the buffered relative poses could give a proper estimate for the current pose of the static object.

4.2.5 General Outline of ME

My Environment (ME) is a module between the perception module, or any other module that handles the perception of the environment objects, and the other modules that use the output of the perception module as shown in Figure 4.8. In some exceptional cases, where the position and the relative angle data are not sufficient, the perception data may be needed to be used directly. For instance, in the robot soccer domain, the ball tracking behavior for the head, where the coordinates of the perceived ball region on the camera frame image may be used to calculate the next pan - tilt parameters, uses the perception output directly.

Localization is one of the modules that requires the perception data. During the perception update of the localization module, the perceived static objects are used to estimate the current pose of the agent.

The inputs of the perception module are the current internal state of

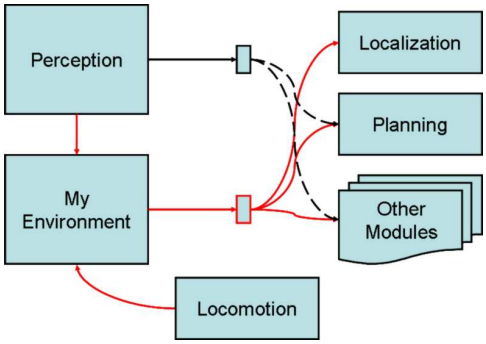

Figure 4.8: Interaction of ME with the other modules

the agent and the latest camera frame image. Input from the camera is very noisy most of the time and may cause false perceptions. Not only the distance of an object could be perceived erroneously, but sometimes an object itself could be recognized as another object. If the data from the perception module are used as they are, these errors could result in unwanted behavior in other modules.

By filtering the output of the perception module and using the past perceptions of the objects at the same time, more stable and robust data could be provided for the localization module as well as other modules. This way the effect of false perceptions and recognitions would be decreased.

It should be kept in mind that for a mobile agent, using the past perceptions can lead to problems if the motion of the agent is not reflected on the past perceptions.

4.2.6 Architecture of ME

There are two kinds of objects in the ME architecture: static objects and dynamic objects. Each object, either static or dynamic, has a buffer window for storing the most recent perception data for that object, and an additional buffer for the current estimation for that object. The data structure of ME is shown in Figure 4.9.

Each static object entry has a distance, a relative angle and a confidence regarding its perception. In the case when a static object has its own orientation, the orientation values are also to be stored in the buffer. For dynamic objects, the velocity should also be calculated, but as it is not directly extracted from a single camera frame, it is not necessarily be buffered. In Figure 4.10a and 4.10b the data structures for static and dynamic objects are defined.

The window size is a hyper-parameter of ME. The noise in odometry, the dynamicity of the agent's pose, and the frequency of the processed vision frames play an important role in the selection of a good window size

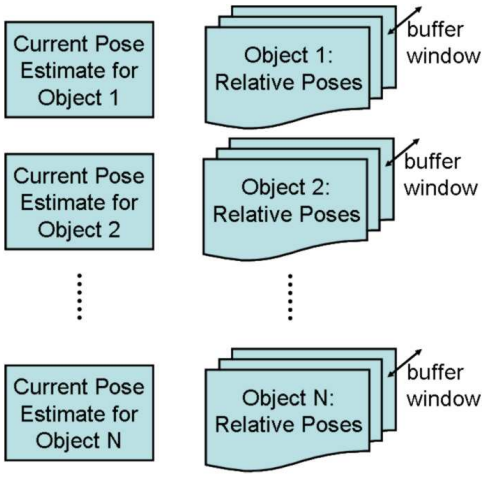

Figure 4.9: The data structure of ME

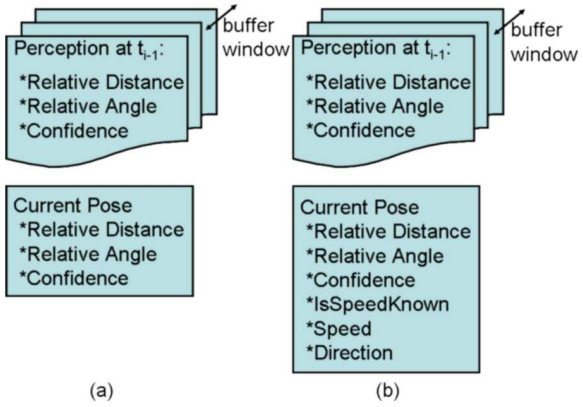

Figure 4.10: Data structures of (a) static objects, and (b) dynamic objects

parameter. For dynamic objects, the speed of such objects should also be taken into account.

For instance, for Cerberus'05, the robots used have a maximum speed of approximately 30 cm/s (with ERS 210s), the odometry is quite noisy as the robots are legged, and the average vision frame processing performance was 18 fps. For such conditions, the size of windows for static objects and dynamic objects were chosen to be 32 and 12 respectively.

4.2.7 Procedures of ME

There are five main procedures of ME. Other than *initialization*, the first two procedures, *perception update* and *odometry update*, are triggered as new perception data from the perception module and new odometry data from the locomotion module are obtained. The other two, *current pose estimation for static objects* and the *current pose estimation for dynamic objects*, are called inside the *perception update* procedure.

Initialization

As an initialization, it is necessary for the pose estimations to set all the buffers in the windows of all the objects, for both the static objects and the dynamic objects. If there is no prior information about the pose of an object when the system has started, all the buffers in its window are to be marked as unknown. If the initial pose of an object is known a priori, then this can be provided to the system by filling the buffers in its windows according to that knowledge.

Perception Update

For each dynamic and static object, the oldest record on the buffer is deleted and a new record is stored from perception if the object is seen at the moment, otherwise it is marked that there is information available about the pose of the object at that time. After updating the buffer with the latest perception output, an estimation is done for each object concerning its pose by calling the appropriate procedure in Section 4.2.7 or Section 4.2.7.

Odometry Update

For the odometry update of each record, three trigonometric functions and a square root is used. For a ME with n_o number of objects and a window size of n_w, there are $n_o \times n_w$ number of records. These calculations increase the cost of ME, but they are mandatory for reasonable ME estimations.

Since all the information in the buffers of all objects' windows is relative to the agent, on each movement action of the agent, they need to be modified.

For each record new relative distance and the relative angle values have to be calculated using Equations 4.44 and 4.45.

$$c = cos(\theta) \times d - \Delta x \tag{4.42}$$
$$s = sin(\theta) \times d - \Delta y \tag{4.43}$$
$$d' = \sqrt{c^2 + s^2} \tag{4.44}$$
$$\theta' = tan^{-1}(s/c) + \Delta\theta \tag{4.45}$$

where θ, d, Δx, Δy and $\Delta\theta$ are the previous relative angle, previous relative distance, the signed distance the agent moved in sideways, the signed distance the agent moved on its orientation and the angle the agent has turned, respectively. The new relative distance and the new relative angle are represented with d' and θ' respectively.

Current Pose Estimation for Static Objects

For each static object, this function is called once in every *perception update*. Using the window sized perception data records; a pose estimation is made for its use in other modules of the agent's architecture like localization and planning.

In Equations 4.50, 4.51 and 4.52 the confidence estimation, the relative distance and relative angle of the static object are calculated.

$$w_j = \sum_{i=0}^{n_w} KA_i^j \times CA_i^j \times f_{ws}(i) \tag{4.46}$$
$$\Delta x_j = \frac{\sum_{i=0}^{n_w} KA_i^j \times DA_i^j \times cos(AA_i^j) \times f_{ws}(i)}{w_j} \tag{4.47}$$
$$\Delta y_j = \frac{\sum_{i=0}^{n_w} KA_i^j \times DA_i^j \times sin(AA_i^j) \times f_{ws}(i)}{w_j} \tag{4.48}$$

$$n_j = \sum_{i=0}^{n_w} KA_i^j \tag{4.49}$$
$$c_j = \frac{\sum_{i=0}^{n_w} KA_i^j \times CA_i^j \times f_{ws}(i)}{w_j} \times f_{wc}(n_j) \tag{4.50}$$
$$d_j = \sqrt{\Delta x_j^2 + \Delta y_j^2} \tag{4.51}$$
$$\theta_j = tan^{-1}(\Delta y_j / \Delta x_j) \tag{4.52}$$

where j is the index of the object, n_w is the window size; w_j is the total weight for the j^{th} object, $f_{ws}(i)$ gives the weight of the i^{th} record for a static object; $f_{wc}(n_j)$ gives the weight for the confidence of an object with n_j known records in its windows; KA_i^j is a flag which is equal to one if the

i^{th} record of the j^{th} object exists (i.e. the object was perceived at the time that record was buffered) and zero otherwise; DA_i^j is the distance of the i^{th} record of the j^{th} object; AA_i^j is the relative angle of the i^{th} record of the j^{th} object; CA_i^j is the confidence of the i^{th} record of the j^{th} object; c_j is the confidence estimation of the j^{th} object; d_j is the relative distance estimation of the j^{th} object; and θ_j is the relative angle estimation of the j^{th} object.

The function $f_{ws}(i)$ is a monotonically increasing function. The value of $f_{ws}(i)$ is to be arranged such a way that the more recent a record it is the more weight it will receive, but at the same time it will not let too small number of records (i.e. one or two) dominate the value of the weight. Although a linear function could easily be used for that purpose, a sigmoid function was expected to give better results if configured properly for the application.

The function $f_{wc}(i)$ is also a monotonically increasing function for favoring the confidence with respect to the number of records of which poses are available.

If n_j is zero, meaning that none of the buffers in the window stores a perceived pose, then no estimation could be made and the object's pose is set as unavailable. After the confidence is calculated, if it is below a predefined threshold, the object's pose is also set as unavailable.

Current Pose Estimation for Dynamic Objects

The procedure of the current pose estimation for the dynamic objects is the same as the current pose estimation for the static objects except that for dynamic objects the speed and the direction of the speed should also be calculated when possible. For each dynamic object, this function is called once in every *perception update*. Using the same equations in Section 4.2.7 the pose estimation, with the exception of the speed related variables, is performed, but the $f_{ws}(i)$ function is replaced with $f_{wd}(i)$, which gives the weight of the i^{th} record for the dynamic object.

If an object is dynamic, perceiving the object in different poses may be either due to noisy perception or the object's movement. If the object's recent poses are buffered and used for the estimation of the current pose, the weights of the most recent records should be higher than they are for static objects, which should always be perceived in the same pose. As a remark, it should be noted that the effect of the movement of the agent is eliminated with the motion update procedure.

The function $f_{wd}(i)$ is also a monotonically increasing function as the function $f_{ws}(i)$ is, but favors the most recent records more than $f_{ws}(i)$. Since the older records are less important for the dynamic objects, the window size for the dynamic objects could be set smaller than the window size set for the static objects.

The speed of a dynamic object is estimated only if the pose of the object was available from the previous run, otherwise the speed is set as unavailable.

In Equations 4.54 and 4.55 the speed and relative direction of the speed of the dynamic object are calculated.

$$h_j = d_j^2 + d_{j-1}^2 - 2 \times d_j \times d_{j-1} \times cos(\theta_j - \theta_{j-1}) \quad (4.53)$$

$$\alpha_j = \Pi - cos^{-1}(\frac{h_j - d_j^2 + d_{j-1}^2}{2 \times h_j \times d_{j-1}}) - \theta_{j-1} - \theta_j \quad (4.54)$$

$$s_j = \frac{\sqrt{h_j}}{\Delta t} \quad (4.55)$$

where j is the index of the object, d_j is the relative distance estimation of the j^{th} object; θ_j is the relative angle estimation of the j^{th} object; α_j is the relative direction of the speed estimation; and s_j is the speed estimation.

4.2.8 Advantages and Disadvantages of ME

ME provides more stable results for both static and dynamic objects. For static objects, especially in the case when localization does not give accurate results, the pose of the static object at ME would be more robust. For localization, the ME output poses can be used as if they were perceived from the sensors at that time. In this way, perceived objects are not forgotten just after they are perceived, but remain in ME for a specific period of time. In addition, the noisy perception, which from time to time may lead to false object detections, could be stabilized.

ME provides more stable results for dynamic objects as well. Using the ME output instead of the perception output directly, instantaneous fluctuations in the pose of the object are smoothened. Losing the dynamic object in some camera frames and perceiving it again frequently, which is not a rare thing in robot soccer, could lead to oscillations in the operations and the outputs of some of the modules. ME smoothens these oscillation with its pose estimation, where it uses the recent perceptions to calculate the current pose.

These advantages have a cost. The space needed to store the pose buffers and current estimations of objects in ME grows linearly with the product of window size of the pose buffers and the number of objects in ME. The complexity of the ME procedures is $O(n_w \times n_o)$, where n_w is the window size and n_o is the number of objects in ME. Using the output of the perception module, both the processing power and memory expenses of ME will be saved, but if the system can afford these expenses, the benefits of ME could be worthwhile.

It should also be noted that using ME, the agent would observe dynamic objects slower than they are. This is because of using the previous poses of the dynamic object in the calculation of the current pose. This problem

could be minimized theoretically by adding the velocity of the previous estimation times the time passed to the previous records of that object, but this could bring more noise than it makes corrections as the velocity estimations could be noisier than the relative position estimations.

4.3 S-LOC: Simple Localization

During the development of the localization module of Cerberus'05, many techniques were taken into consideration.

- Triangulation is a simple and accurate technique, but is not robust. It is too much effected by noise, specially by the false perceptions [9].
- The major disadvantage of Kalman Filter methods is that they do not have the capability of recovering from kidnapping [12, 13].
- ML approaches are generally expensive, where false perceptions could be big problems [14, 15].
- Raw MCL cannot recover from kidnapping, but a version of it, SRL is implemented [16, 17, 18].
- ML-EKF is also another expensive technique, which would not be preferred in a case where a much lower cost algorithm could give accurate and robust results [19, 20].
- Fuzzy localization techniques generally have high computational complexity, and do not give results with enough accuracy that are worth the cost [10, 11].
- R-MCL is also another technique, which is used in the experiments for comparison purposes [4, 5, 6].

Considering the points above, it was decided to implement *S-Loc* together with a version of SRL. The existing R-MCL module implemented in our laboratory is also used in the experiments.

In general, the localization process has two main steps. The first one is the perception update, which is based on the perceptions in order to calculate the estimated pose of the agent. Since the movement of the agent changes its pose, the second step, the odometry update, is necessary for reflecting the effect of the movement on the calculations and the estimations.

Perception update, as it depends on the perceptional information, usually includes high amount of noise. Although the agent is dynamic, its pose should not be highly unstable, i.e. the pose should not jump to different poses that are far away on the field frequently. Using a memory for the previous pose estimate, and updating it with the current estimate could handle

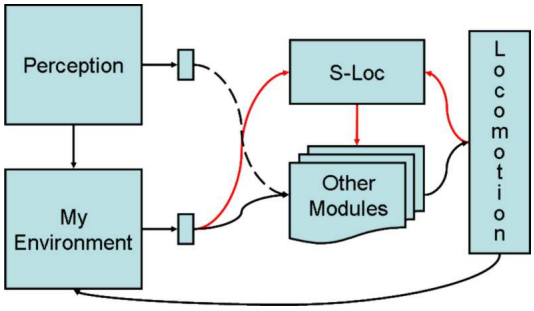

Figure 4.11: The relationship of the S-Loc module with the other modules

the big fluctuations and increase the robustness to the false perceptions of static landmarks.

In order to use triangulation, three objects, which are not available at the same time frequently, are needed to be perceived. Also, even if three objects are available, in the case where one of the perceptions is wrong or is highly noisy, the calculation will lead to a very noisy pose estimation.

In the MCL, there is a large number of sample poses, for which many calculations should be made in order to find their confidences. Generally, most of these samples do not hold any useful information. Also, noisy perception data may lead to unstable pose estimations.

The principle of ML leaves open how the robot's belief is represented and how the conditional probabilities are computed. Existing ML approaches mainly differ in the representation of the state space and the computation of the perceptual model. These approaches are generally expensive, since the space is discretized and for each perception and for each location, the probabilities should be calculated at each frame. False perceptions could also be major problems.

In the perfect, noise-free case, the odometry data should be continuous and the pose should be updated continuously as the agent moves. On the other hand, in the real world case, the odometry data is generally very noisy, especially when the agent uses legs for locomotion; and arrives at discrete times, for instance after a step is completed. Both of these make the previous pose estimates less confident for the current estimate calculations.

4.3.1 General Outline of S-Loc

S-Loc is a localization module. It needs the perception data and the odometry data for updating the pose estimate, which it provides as the output. This pose estimate is then used in other modules. The relationship of the *S-Loc* module with the other modules is shown in Figure 4.11.

The perceived data can be obtained directly from the vision module (or any other perception module), or they can be supplied by the ME module

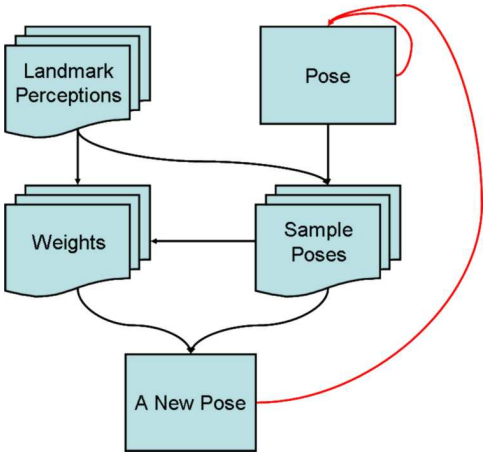

Figure 4.12: The perception update process

where they are buffered and estimates using them are produced. It could perform better if the perceptional input is provided by the ME module, because the ME module provides more stable and robust data.

The locomotion module provides the odometry data at certain times, which is generally less frequent than the perception data. The effect of the movement of the agent should also be reflected on the pose estimate.

4.3.2 Architecture of S-Loc

The perception update of the S-Loc depends on the perception of landmarks and the previous pose estimate. The perception update process is shown in Figure 4.12. Even if the initial pose estimate is provided wrong, it acts as a kidnapping problem and is not a big problem as S-Loc will converge to the actual pose in a short period of time if enough perception could be obtained during this period.

For each perceived landmark, a sample pose is calculated according to this perception and the previous pose estimate of the agent. The previous pose estimate is also taken as a sample pose.

For each sample pose, using all the landmarks, the likelihood of this sample pose is calculated. This is done by assuming that the agent's actual pose is the sample pose being processed and calculating the difference of the perceived landmarks positions and their actual positions. Also, the confidence of the perception is reflected on the likelihood.

After these likelihood calculations are done for each sample pose, these likelihoods are used for calculating the weights of the corresponding sample poses, and a new pose is calculated as the weighted average of these sample poses.

The weighted average of these sample poses is then used together with

the previous pose estimate to calculate the current pose estimate. The purpose of not using the weighted average of these sample poses is to directly provide the system enough memory to prevent big jumps of the pose estimate and make it more stable.

After the current position of the agent is estimated, it could be safer to calculate the current orientation of the agent using the current position estimation and the perceptions.

In the case of having no perception at a certain time, the current pose estimate could be obtained by decreasing the confidence of the previous pose estimate.

The odometry update process is as simple as updating the pose estimation with the odometry data. Since only the pose estimation is used from the previous cycle of every estimation, no more update or calculation is necessary. On the other hand, if the frequency of the odometry update is much less than the frequency of the perception update, then it may be better to lower the weight of the odometry data accordingly. This is because having the original odometry data to be the result of the motion during more than one perception updates.

4.3.3 Procedures of S-Loc

There are three main procedures of *S-Loc*. The *initialization* is the first one. Other two procedures, *perception update* and *odometry update*, are triggered as new perception data from the perception module and new odometry data from the locomotion module arrive.

Initialization

The only thing to be done in the initialization procedure is to initialize the pose estimate to initial value. It does not have to be the actual pose that the agent will have at the beginning, since *S-Loc* module can recover from kidnapping. On the other hand, it should still be set to a valid pose initially in order not to cause a problem in the proceeding calculations.

Perception Update

As shown in Equations 4.56, 4.57, 4.58, 4.59 and 4.60, the first pose sample is the previous pose estimate.

$$PSA_x^0 = PE_x \tag{4.56}$$
$$PSA_y^0 = PE_y \tag{4.57}$$
$$PSA_\theta^0 = PE_\theta \tag{4.58}$$
$$PSA_c^0 = PE_c \tag{4.59}$$
$$PSA_w^0 = PE_c \times f_{wpu2}(PSA_x^0, PSA_y^0, PSA_\theta^0, PA) \tag{4.60}$$

$$PA_k^0 = 1 \tag{4.61}$$

where PS_x^0, PS_y^0, PS_θ^0, PS_c^0 and PS_w^0 are the x-coordinate, y-coordinate, orientation, confidence and weight of the first pose sample; PE_x, PE_y PE_θ, and PE_c are the x-coordinate, y-coordinate, orientation and confidence of the pose estimate before the perception update; PA, percepts array, is the collection of perception data of all the perceived landmarks together with their coordinates that are known initially; f_{wpu2} is the function that returns a weight component for a pose according to the current perceptions; and PA_k^0 is set to one in order to have the first element of pose sample array included in the proceeding calculations.

Then, a separate pose sample is calculated for each perception as in the Equations 4.63, 4.64, 4.66, 4.67 and 4.68.

$$\alpha_i = tan^{-1}\left(\frac{PE_y - PA_y^i}{PE_x - PA_x^i}\right) \tag{4.62}$$

$$PSA_x^i = PA_x^i + PA_d^i \times cos(\alpha_i) \tag{4.63}$$

$$PSA_y^i = PA_y^i + PA_d^i \times sin(\alpha_i) \tag{4.64}$$

$$\beta_i = tan^{-1}\left(\frac{PSA_y^i - PA_y^i}{PSA_x^i - PA_x^i}\right) \tag{4.65}$$

$$PSA_\theta^i = \pi + \beta_i - PA_\theta^i \tag{4.66}$$

$$PSA_c^i = PA_c^i \tag{4.67}$$

$$PSA_w^i = f_{wpu1}(PA^i) \times f_{wpu2}(PSA_x^i, PSA_y^i, PSA_\theta^i, PA) \tag{4.68}$$

where α_i and β_i are dummy angle variables; PSA_x^i, PSA_y^i, PSA_θ^i, PSA_c^i and PSA_w^i are the x-coordinate, y-coordinate, orientation, confidence and weight of the i^{th} pose sample; PA_x^i, PA_y^i are the actual x-coordinate and y-coordinate of the i^{th} landmark in the percepts array; PA_d^i, PA_θ^i and PA_c^i are the perceived relative distance, relative angle and the perception confidence of the i^{th} landmark in the percepts array; PA^i is the perception data of the i^{th} perceived landmark which is stored as the i^{th} element of the Perception Array; and f_{wpu1} is the function that returns a weight component for a pose according to the perception for which the pose sample is calculated.

The function f_{wpu1} returns the first component of the PSA_w^i for the argument PA^i. It may return PA_c^i directly or any other number that gives the confidence that the perception is correct. Since the accuracy of the pose sample will be taken into account by the function f_{wpu2}, this function is independent of the corresponding pose sample. The purpose of this function is to decrease the weight of the pose samples, of which the perception is less confident. In the case where the perception module does not provide healthy confidence values, the perceived relative distance of the landmark can be used for the calculation of the return value. In such a case, a properly

configured sigmoid function can be very suitable. If the landmarks are of different types and are known to have different perception accuracy, then this could also be reflected on the return value.

The function f_{wpu2} returns the second component of the PSA_w^i. The return value is related to the accuracy of the pose sample according to all the perceived landmarks. For each perceived landmark, the position of the perceived landmark is calculated by adding the perceived distance on the perceived relative angle to the pose sample, and the resulting position is compared to the actual position of the landmark. The difference gives the error. The return value should be a function of the error as in Equation 4.71.

$$par_x^j = \left| PA_x^j - (PSA_x^i + PA_d^j \times cos(PSA_\theta^i + PA_\theta^j)) \right| \quad (4.69)$$

$$par_y^j = \left| PA_y^j - (PSA_y^i + PA_y^j \times sin(PSA_\theta^i + PA_\theta^j)) \right| \quad (4.70)$$

$$f_{wpu2} = \prod_{j=1}^{N_L} PA_k^j \times f_{wpu3}\left(par_x^j, par_y^j\right) \quad (4.71)$$

where N_L is the number of landmarks; PA_k^j is one if the perception of the j^{th} landmark is available, and zero otherwise; and f_{wpu3} is a function that returns a value related to the difference in the x-coordinate and the y-coordinate.

The return value of the function f_{wpu3} is a value for the confidence of the sample pose for the corresponding perceived landmark. The greater the x-coordinate and y-coordinate differences provided as parameter to this function, the worse the sample pose fits to that landmark perception and therefore the less confidence the function shall return.

The calculation of the new pose estimate is the last step of the perception update. Except the new orientation estimate, all the estimation values are the weighted average of the recent calculation, which is in turn a weighted average of sample poses, and the corresponding previous estimate value. The new orientation estimate is calculated by using the new coordinate estimates and the perceptions. The new values of pose estimate are calculated from the Equations 4.74, 4.75, 4.78, and 4.79.

$$hp = f_{HP}\left(\sum_{j=1}^{N_L} PA_k^j\right) \quad (4.72)$$

$$tw = \sum_{j=0}^{N_L}\left(PA_k^j \times PSA_w^j\right) \quad (4.73)$$

$$PE_x^* = hp \times PE_y + (1-hp) \times \frac{\sum_{j=0}^{N_L}\left(PA_k^j \times PSA_x^j \times PSA_w^j\right)}{tw} \quad (4.74)$$

$$PE_y^* = hp \times PE_y + (1 - hp) \times \frac{\sum_{j=0}^{N_L} \left(PA_k^j \times PSA_y^j \times PSA_w^j \right)}{tw} \quad (4.75)$$

$$\beta_i = tan^{-1} \left(\frac{PE_y^* - PA_y^i}{PE_x^* - PA_x^i} \right) \quad (4.76)$$

$$wa_i = f_{wpu1}(PA^i) \times f_{wpu2}(PE_x^*, PE_y^*, \beta_i, PA) \quad (4.77)$$

$$PE_\theta^* = tan^{-1} \left(\frac{\sum_{i=1}^{N_L} \left(PA_k^i \times sin(\beta_i) \times wa_i \right)}{\sum_{i=1}^{N_L} \left(PA_k^i \times cos(\beta_i) \times wa_i \right)} \right) \quad (4.78)$$

$$PE_c^* = hp \times PE_c + (1 - hp) \times \frac{\sum_{j=0}^{N_L} \left(PA_k^j \times PSA_c^j \times PSA_w^j \right)}{tw} \quad (4.79)$$

where PE_x^*, PE_y^*, PE_θ^* and PE_c^* are the updated x-coordinate, y-coordinate and orientation of the pose estimate; and f_{HP} is a function that returns a history coefficient according to the number of percepts available.

Odometry Update

For the odometry update, the only necessary thing is to update the current pose estimation with the new odometry data. No more update or calculation is necessary, because nothing is used from the previous cycle of estimation other than the pose estimation.

It should also be noted that, in the case where the frequency of the odometry update is much less than the frequency of the perception update, transforming the odometry data to lower values may lead to better results since the original odometry data is the result of the agent's motion from the previous odometry update to the current one, and this would last for more than one perception updates.

In Equations 4.80, 4.81 and 4.82 the new (updated) coordinates and orientation of the pose estimate is calculated.

$$PE_x^* = PE_x + \Delta x \times sin(PE_\theta) + \Delta y \times cos(PE_\theta) \quad (4.80)$$

$$PE_y^* = PE_y + \Delta y \times sin(PE_\theta) - \Delta x \times cos(PE_\theta) \quad (4.81)$$

$$PE_\theta^* = PE_\theta + \Delta\theta \quad (4.82)$$

where PE_x^*, PE_y^* and PE_θ^* are the updated x-coordinate, y-coordinate and orientation of the pose estimate; PE_x, PE_y and PE_θ are the x-coordinate, y-coordinate and orientation of the pose estimate before the odometry update; Δx, Δy and $\Delta\theta$ are the odometry data giving the change in the x-coordinate, y-coordinate and orientation.

4.3.4 Advantages and Disadvantages of S-Loc

In ML, for each landmark seen the probability distribution is modified accordingly, and as a result, the final probability distribution is expected to

give the agents real pose. Instead of a probability distribution, a pose, which is most likely to be the actual pose according to the previous pose estimate, is used in S-Loc. In this way, as it is the case in ML, the pose estimate converges to the actual pose of the agent.

Considering only the most likely sample poses, *S-Loc* acts like a kind of ML but with a local coverage. Although it has a local coverage, it responds in a fast manner to the kidnapping problem, as the most likely sample poses could be far away from the previous pose estimate. In addition, since only a sample pose for each landmark is calculated, S-Loc has a much lower cost than ML.

In comparison with triangulation, S-Loc does not calculate the best estimate according to the perception of the moment, but makes the estimation in a way that it converges to that point in a short period of time. On the other hand, the effect of the false perceptions is greatly decreased as the sample pose of such a perception would have a relatively small confidence and will not play a big role in the pose estimation. In this way, the robustness is increased without decreasing the performance.

In a way, *S-Loc* works similarly as the MCL since the sample poses are used in the same way they are used in MCL. The main difference is the selection of these sample poses. In MCL, there is a large number of pose samples, and they are populated according to their confidences, and randomly mutated for small changes. In S-Loc new pose samples are calculated for each estimation, and a pose sample is calculated for each perceived landmark. In this way, *S-Loc* becomes a much lower cost localization method with accurate pose estimation capability.

The memory used in the *S-Loc* increases the robustness of the system even further and the big jumps of the pose estimate are prevented.

Chapter 5

Planning

The soccer domain is a continuous environment, but robots operate in discrete time steps. At each time step the environment, and the robots' own states change. The planner keeps track of those changes, and makes decisions about the new actions. Therefore, first of all, the main aim of the planner should be sufficiently modeling the environment and updating its status. Second, the planner should provide control inputs according to this model.

We wanted to come up with a unified planner model, that operates in discrete time steps, but exhibits continuous behaviors, as shown in Fig. 5.1. The unified model should be independent of roles and plans. For each time step the planner will be queried, and it will control the robot according to the current state. Control of the robot will on roles, since each role has different primary aims, e.g., a goalie's main aim is saving goal, but an attacker's main aim is making score.

Simple reflexive methods can not work in the continuous soccer environment. A planner should keep a memory about the status of the world, and the robot. It should also reflect the changes to this model and according to those changes it should adapt its actions, and take the next actions aim biased.

Although planning is a quite difficult task it can be brokendown into subtasks. Because of that we modeled a Multi-Layer Planning architecture since by distributing requirements and jobs to different layers, and subactions in these layers, the difficulty is distributed and decreased, too.

5.1 Multi-Layer Planning

For our planner we divided the plan into four layers, shown in Fig. 5.1.

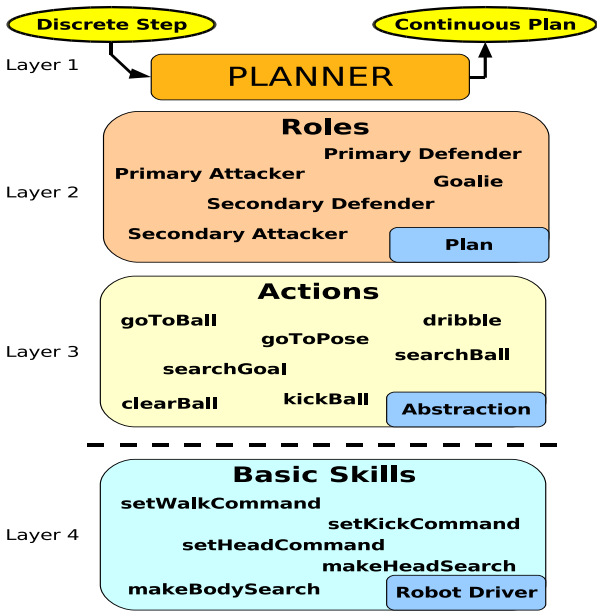

Figure 5.1: Multi-layer Planner

5.1.1 Top Planning Layer

The first layer, at the top of the multi-layer structure is the *Planner* layer. The planner is the only interface of the planning module to the outside. All the communication with the plan is done with this layer. At each time step, the Planner is used to take the next actions. Each robot in the field has different aims. According to those aims they have different roles. The planner keeps reference to some main modules, and a reference to one of these roles, for each robot. Each role bases its decisions on its aim, and hence shows different behaviors. However, Planner layer abstracts this variances, and shows the same interface to the outside.

5.1.2 Role Layer

A role is typically represented as a finite state machine. The nodes in this machine reflects the actions, and transitions are thepaths to the next actions according to the results of the previous ones. A role may or may not use a finite state machine, and may be implemented in a different manner. However, we designed a role to be seen unified from the Planner Layer side, independent of using Action Layer or not.

As an example, seen in Fig. 5.2, a role is combination of actions according to the aim. The figure shows an example Attacker Role. The aim of the

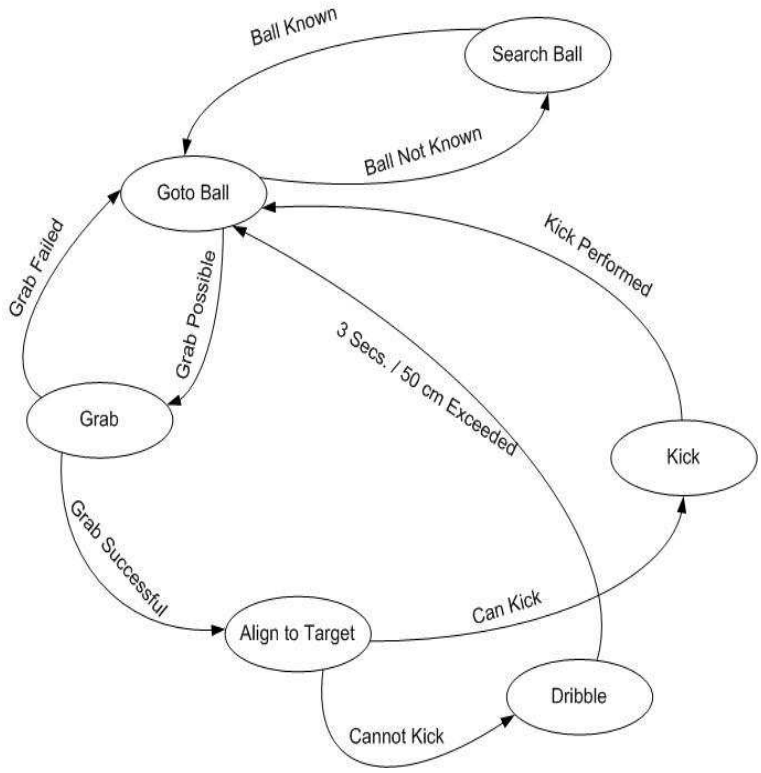

Figure 5.2: Example Role - Attacker Role

attacker is scoring goals. So the actions are combined to lead the robot to make score. It first searches the ball, so an action for this is modeled. Then it goes to the ball. If the ball is missed while going to it, the robot again starts to search it. If the robot reaches the ball, it tries to grab the ball. Then, with the grabbed ball, the robot tries to align with the goal. At the end, if the alignment is successfuly accomplished in the time limits, the attacker role leads robot to make a kick.

A role keeps track of the current action. Until the current action is finished, it continues to process it. When the current action is finished, according to the memory, the current action, and its result, next actions are set to be the current action.

The planner Layer abstractes all the planning module from outside, and Role Layer abstractes its content from the Planner Layer. When the Planner Layer is requested for the next operation, it passes this to the current role, and the current role passes this request to the current action. If the current action finishes, another action is selected, and the request is passed to it. To

achieve this model, we used *Chain of Responsibility Design Pattern* [25]. So, responsibility is passed between a chain of objects, from the top level layer down to the simple basic operations. This enables both easily variations in the implementations, and abstraction.

5.1.3 Action Layer

Each role contains nodes in their state machine. The nodes are implemented in Action Layer, as Actions. An Action is a series of operations. It keeps memory about the flow of these operations. For example, *align to goal* action has simple operations like, turn, keep ball grabbed, check the goal. Moreover, the action keeps a memory about the passed time. If the elapsed time exceeds the ball hold time limit, the action should release the ball and finish. If the robot successfuly aligns to the goal, it should finish in success and according to the role should pass to another action. For example in the attacker role, the current action should be changed to kick ball action.

As shown in Fig. 5.3, actions either continue or not. If it continues, it takes the next operation according to memory. If it does not, it should provide the cause of the halt. This cause will be the basis of the next decisions.

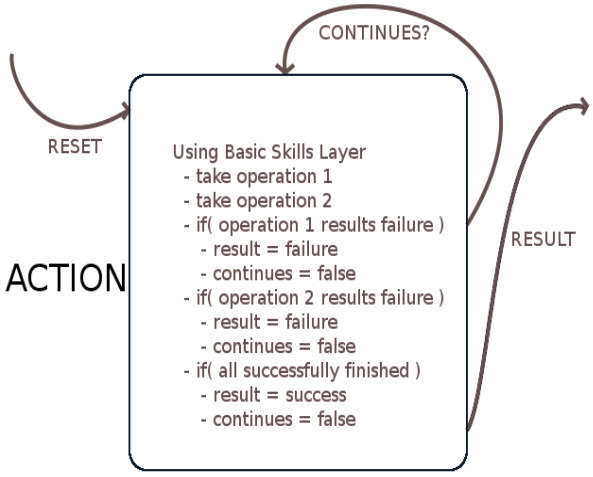

Figure 5.3: Action Diagram

All the planning module depends on switching from one action to the other. So we have a set of actions, and roles in hand. Creating all of them at the start may seem costly. However, because we will continuously switch from one action to the other, removing and reallocating them would lead to greater time loss. So instead of removing the deselected action, and creating

another one according to the state diagram, we generated all possible actions and roles beforehand, and stored them in the memory. For this purpose, we used *Factory Design Pattern* [26]. If a role was alreadyg enerated then when requested, the generated one is returned. If it was not generated, then when requested, it is created, and the newly created object is returned. This object is never removed again, but its status is reset for each call. Resetting an action makes the action las if it were newly generated.

Actions are self contained, and abstracted from each other and from roles that uses them. This separation and abstraction of actions gives us the ability to write them separately. So we achieved working in parallel for making a plan better. When an action is improved, it greatly affects the general plan. According to aspect of the plan, actions are organized, so in a way *Aspect Oriented Programming* model [27] is applied.

5.1.4 Basic Skills

The last and the most important layer is the *Basic Skills* layer. This layer abstracts the robot from the plan. It may be thought as a robot driver. The plan uses this driver to send commands to the robot. Independent of the implementation of these commands at the robot side, a plan can be designed. However, the success of the plan lies beneath the successful application of these basic skills. For example, a plan may command the robot to walk. Some robots will complete this command in less time as walking quite quick, but some will be slow and take more time to complete. The differences in the implementations of the basic skills will greatly affect the success of the plans.

5.2 Fuzzy Inference Engine

In control typically crisp values result in oscillations in the decisions, but fuzzy values make smoother transitions. Some actions in the planning module may be thought as control problems, like following the ball with the head of the robot and they were implemented as Fuzzy Inference Engines (FIS).

First, the fuzzy variables were defined. For ball following, as input, fuzzy variables distance, orientation, and as output, fuzzy variables nod, pan and tilt are used. Second, a fuzzy inference system is modeled in Matlab. In this example we used Fuzzy Mamdani Controller, but Tukego Sugeno Fuzzy Controller could also be used. Matlab also supports different fuzzy control methods for the inference part [28]. The system is tested with sample inputs, and 3D graphics are drawn for each input dimension sets. So instead of finding an equation for ball following problem, it is modeled as a control problem and solved with a fuzzy control structure. As a result, we achieved a tested and ready to be run on the robot formula, shown in Fig. 5.4. Matlab provides a C library for running the modeled inference system in

other applications. We extended this library, and converted it to C++ language.

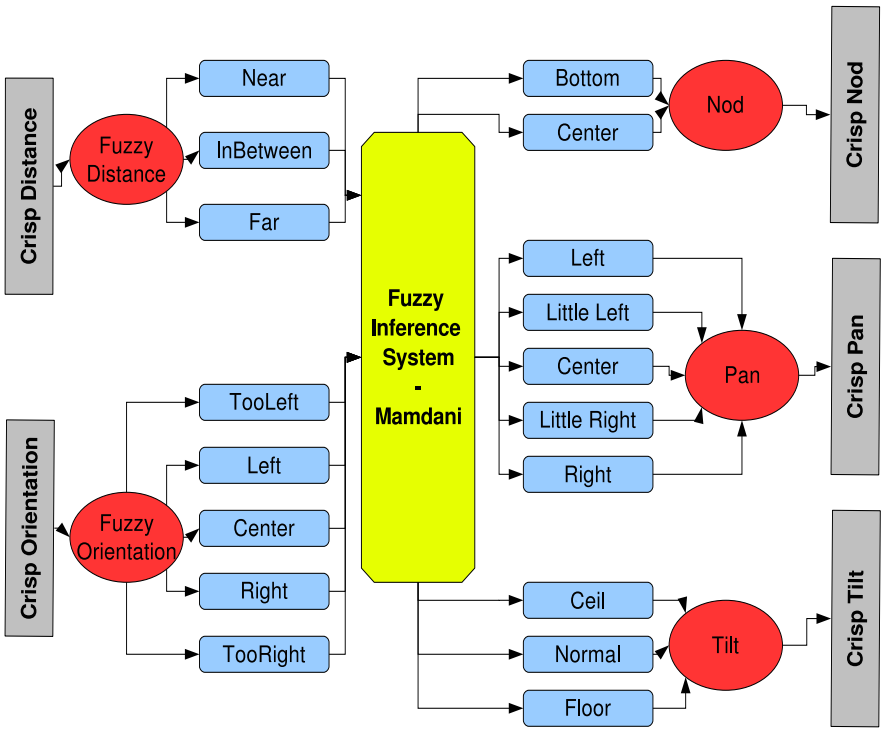

Figure 5.4: Using Fuzzy Control for tracking a ball with the head of the robot

As our layered planning structure infers, we made an action to follow the ball with the head. This action loads the engine from a file. At each time step the inputs, distance and orientation values, are given to the engine, and outputs, nod, tilt and pan values, are estimated, and applied. The library runs quite fast, and the system gives really robust results. One of the major benefit of using a FIS in actions those solve control problems is that, without changing any code, we may change the system with making changes on the FIS definition file. So it would be really easy to adapt the system to the new field conditions. If the ball is perceived smaller than before because of the light conditions, then only by changing the set boundaries of fuzzy variables will be quite enough. Moreover, because, the fuzzy values enable smoother passes for output values, the system at the hand, that is trained in the lab conditions, will also work for new conditions, with requiring some small fine-tunings only.

We classified actions depending on whether they can be modeled as a

control problem or not. For example tracking the ball with the head is classified as a simple control problem. Using fuzzy control for tracking the ball with the head eliminated noises, and resulted in a more stable action. Because of some erratic behaviors in some sub-modules, the conversion of control problem oriented actions to fuzzy controlled actions could not become stable before the tournament. Therefore, we decided to revoke all actions those uses fuzzy logic, and instead of them we used the modified old ones. Fuzzy Logic Control is really powerful in solving some big problems easily, and we want to use this force in the next tournaments.

Chapter 6

Motion

In 2005, we had developed our own walking engine for ERS 210s [3]. Due to the highly platform independent and modular structure of the modules of our system, porting our motion engine to ERS-7s and making our robots walk took only about half an hour. However, fine tuning the motion module in order to achieve an effective walking speed and stability took almost an entire semester.

At the lowest level, the Aibo's gait is determined by a series of joint positions for the three joints in each of its legs. More recently, trajectories of the Aibo's four feet through three-dimensional space have been used to develop a higher level representation for Aibo's gait. An inverse kinematics calculation is then used to convert these trajectories into joint angles. Among higher-level approaches, most of the differences between gaits that have been developed for the Aibo stem from the shape of the loci through which the feet pass and the exact parameterizations of those loci.

6.1 Kinematic Model

At the position level, the problem is stated as, "Given the desired position of the robot's paw, what must be the angles at all of the robots joints?".

In our approach, inverse kinematics techniques are used to calculate the desired joint angles while the paw is moving along the path determined by the locus of the leg. The locus is divided into *pStep* (to be explained in Section 6.5) points, and each of these points has (x,y,z) values. When the paw is to move to the location of the next point on the locus, the following formulas are used to calculate the necessary joint angles in order to make the paw move toward this point. A simple representation of an Aibo leg is illustrated in Figure 6.1.

The law of cosines is used for calculating the knee angle (θ_3).

$$d^2 = x^2 + y^2 + z^2 \tag{6.1}$$

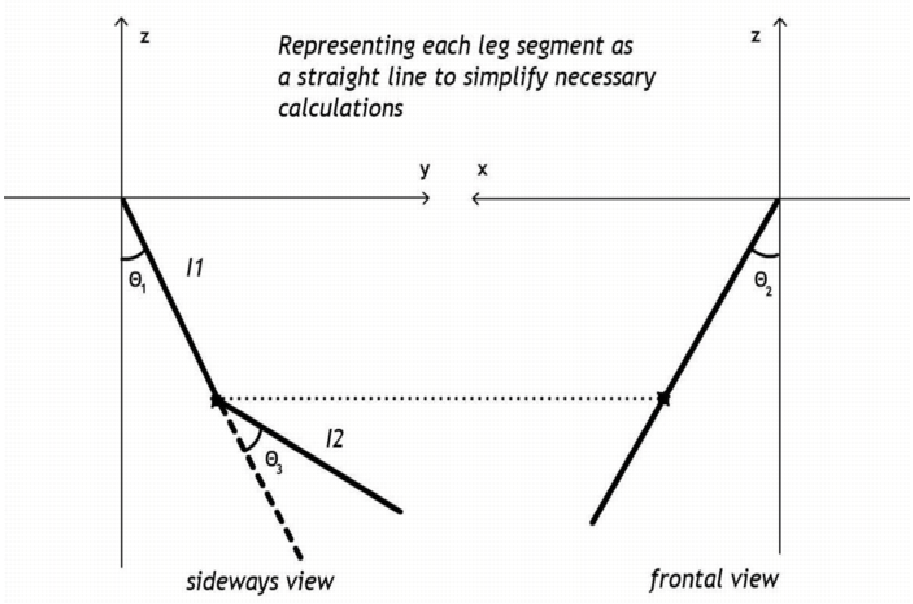

Figure 6.1: Simple kinematic model representation for front right leg.

where (x, y, z) represents the 3d coordinates of the paw according to the shoulder.

$$I_1{}^2 + I_2{}^2 - 2I_1I_2\cos(\pi - \theta_3) = d^2 \tag{6.2}$$

$$\theta_3 = \pi - \arccos(\frac{I_1{}^2 + I_2{}^2 - d^2}{2I_1I_2}) \tag{6.3}$$

Then the abductor angle θ_2 is calculated.

$$\theta_2 = \arcsin(\frac{x}{I_1 + I_2\cos(\theta_3)}) \tag{6.4}$$

Finally, rotator angle θ_1 is calculated.

$$\theta_1 = \frac{y\cos(\theta_2)(I_1 + I_2\cos(\theta_3)) + zI_2\sin(\theta_3)}{yI_2\sin(\theta_3) - (z\cos(\theta_2)(I_1 + I_2\cos(\theta_3)))} \tag{6.5}$$

6.2 Walking Styles

To produce a walking motion, the legs must not be at the same position on the walk locus at the same time. Essentially, the legs must move out of phase of each other. Human walking actually uses a similar approach. One leg is lifted, moved forward, and then dropped, while the other stays where

it was. Once the first step has been taken, the other leg is then lifted and basically mirrors the same action taken by the first leg.

Different gait types can be obtained by shifting the movement phases of each leg in different manners. Timing of each leg and resulting walking type is shown in Figure 6.2

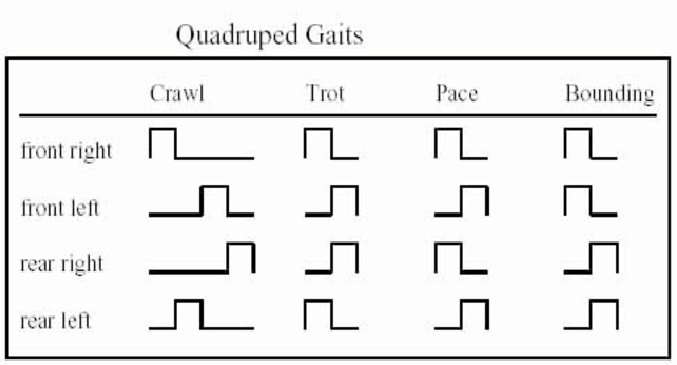

Figure 6.2: Different timing of each legs motion results in different walking styles [30].

6.3 Omnidirectional Motion

Omnidirectional walking can be thought as the motion of a shopping cart, and can be obtained by treating the legs as wheels. This is illustrated in Figure 6.3.

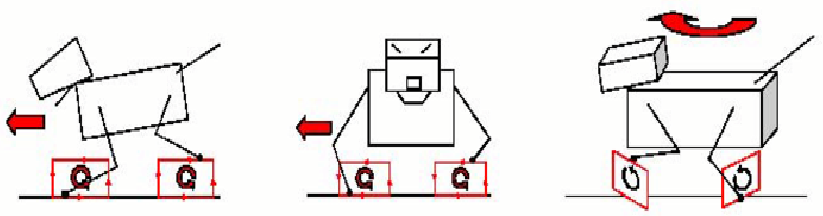

Figure 6.3: Using legs as the wheels of a shopping cart [29].

In order to achieve this motion, three walk components named *forward*, *sideways*, and *turn* are used. These components are represented as 2-dimensional vectors and they are added vectorally in order to obtain one resulting vector and its symmetric part according to the initial paw location. These two resulting vectors together produces the limits of the locus; that is the limits of each leg's area of operation. The body of the robot can be approximated as a rectangle from the top view and the angle of the turn

components can be calculated as the arctan($bodyWidth/bodyHeight$). The operation of obtaining locus limits by using forward, sideways, and turn components is illustrated in Figure 6.4.

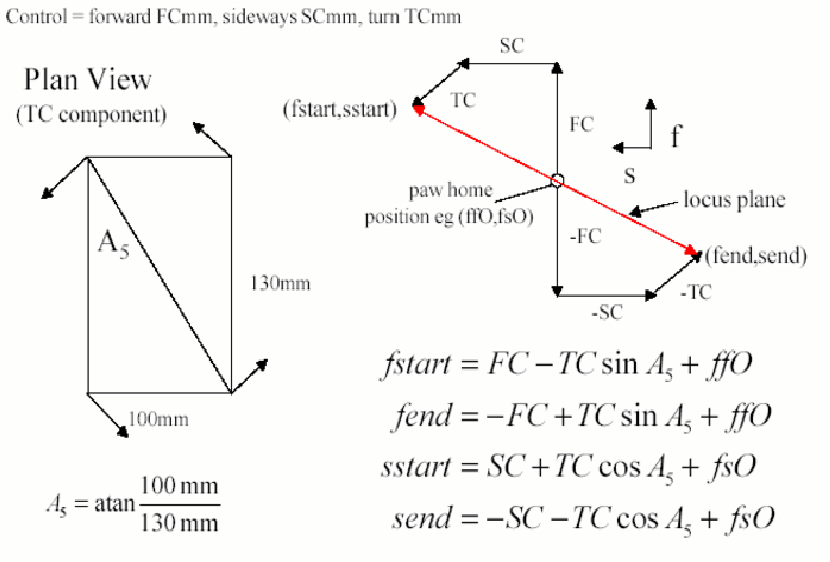

Figure 6.4: Forward, sideways, and turn components are added vectorally to obtain a resulting vector; which indicates the direction and limit of the paw movement [31].

Trot gait, in which the diagonally opposed legs are synchronized, is used as the primary gait type. Omnidirectional motion is inherited from ParaWalk as it is. The only difference is the meaning of sideways and turn components. In ParaWalk, default sideways direction is leftwards, and default turn angle increases counterclockwise. In our approach, default sideways direction is rightwards, and default turn angle increases clockwise. Resulting motions for different combinations of walk components is shown in Figure 6.5.

6.3.1 Representing the Locus

According to the research done so far, rectangular, trapezoidal and half elliptic loci are not effective; in fact they have a hindering effect on robot's movement. Especially the movement of rear legs is the cause of this effect. While performing these kinds of movements the leg touches the ground in the same direction of the movement, which in turn decreases the robot's momentum at that time.

Proposed locus is in the shape of an ellipse cut from below in some proportion. This shape can be approximated by a hermite curve and it is illustrated in Figure 6.7.

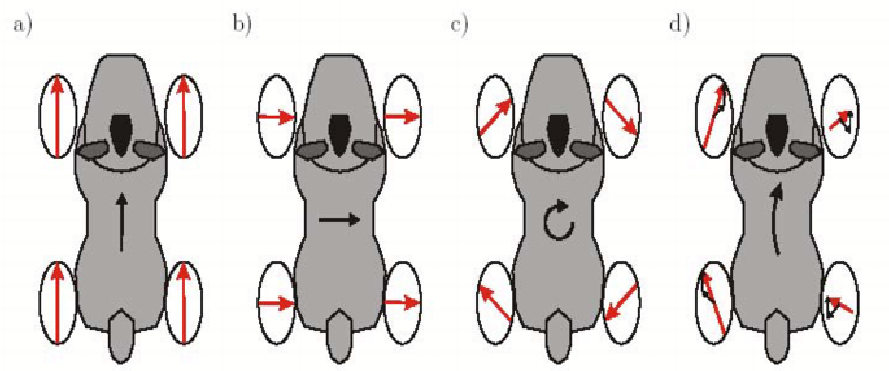

Figure 6.5: Resulting motions with different combinations of forward, sideways, and turnCW parameters: (a) only forward, (b) only sideways, (c) only turnCW, (d) forward and turnCW together [30].

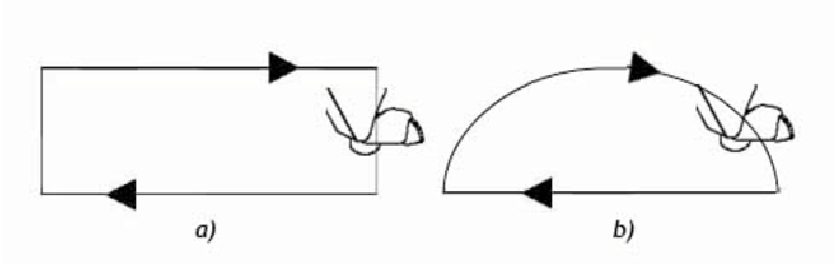

Figure 6.6: Movement of the paw on a (a) rectangular locus and a (b) half elliptic locus.

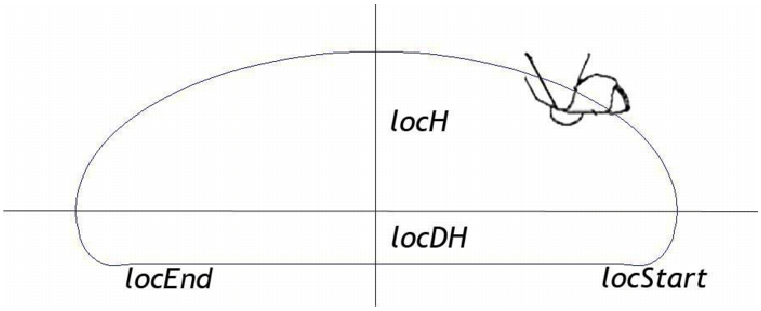

Figure 6.7: Proposed locus in the shape of a hermite curve.

With the introduction of elliptic locus, this effect is avoided since the leg touches the ground after moving in the reverse direction of the movement for a short period of time. This movement type guarantees that the moment of the robot is not hindered but increased. Also, elliptic locus makes the movement of the leg smoother.

6.4 Object-oriented Design

Locomotion module is designed by using an object-oriented approach. First of all, the robot is thought as a single object composed of many other objects. Specifically, an AIBO robot physically consists of four legs, a head, and a tail, each of which carries different number of joints. Each Leg has three Joints, which are the rotator, the abductor, and the knee joints. The Head has three Joints, which are pan, tilt, and roll joints. Finally, the Tail has two Joints, which are pan and tilt joints. All these objects are defined as a separate class. The classes used for the locomotion module is shown in Figure 6.8.

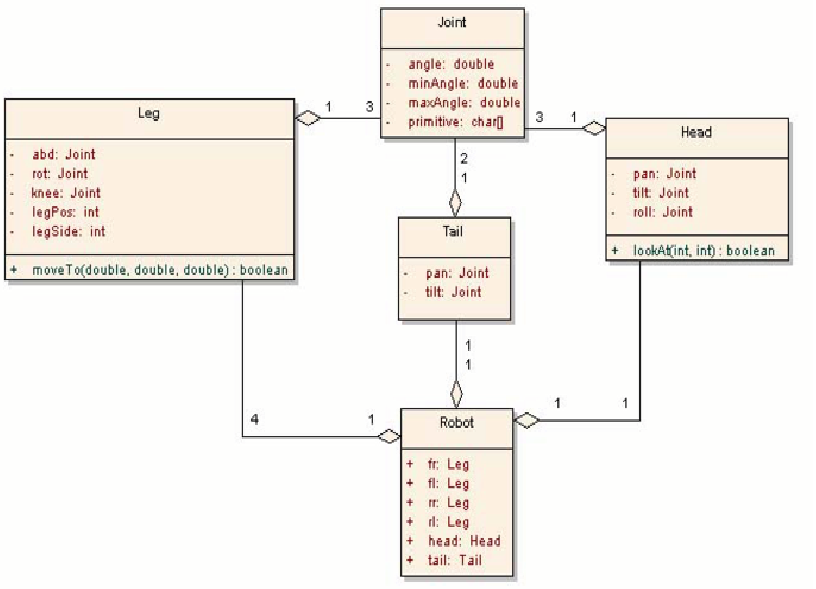

Figure 6.8: Class diagram showing the relations between classes used in the locomotion module.

Leg class has a method named *moveTo* for calculating the required joint angles to be able to reach a specific point in a 3-dimensional space. It performs the aforementioned inverse kinematics calculations and determines the knee, abductor, and rotator angles of the leg, respectively.

Besides these robot related classes, there are two very important classes. One is MotionManager class, which is responsible for the coordination of all movements, and the other is GA class, which is responsible for generating an initial population according to the sample string provided, and then performing the main GA operations (reproduction, crossover, mutation) on each population in order to generate parameter lists to be used during experiment processes.

6.5 Parameter Optimization

There are 11 parameters used by the new walking engine. These parameters can be categorized as step duration related, locus related, and initial paw locations related parameters.

1. Step duration related

   pStep: Number of steps needed to complete one full step (i.e. the paw comes back to its initial position).

2. Locus related

   fLocH: Height radius of the ellipse to be used as the locus for front legs.

   fLocDH: Perpendicular distance of the center of the ellipse of the front locus from the initial paw location.

   bLocH: Height radius of the ellipse to be used as the locus for rear legs.

   bLocDH: Perpendicular distance of the center of the ellipse of the rear locus from the initial paw location.

3. Initial paw locations related

   hF: Height of the chest of the robot from ground.

   hB: Height of the back of the robot from ground.

   fs0: Sideway distance of the paws of the front legs from shoulder.

   ff0: Forward distance of the paws of the front legs from shoulder.

   bs0: Sideway distance of the paws of the rear legs from shoulder.

   bf0: Forward distance of the paws of the rear legs from shoulder.

This year, evolution strategies (ES) [32], a kind of evolutionary algorithm, is applied to optimize the walking parameters. This technique is based on adaptation and evolution principles. The most important property of this algorithm is that vectors which represent the genes can have real values. So, ES can easily be applied our gait parameters, which have real values.

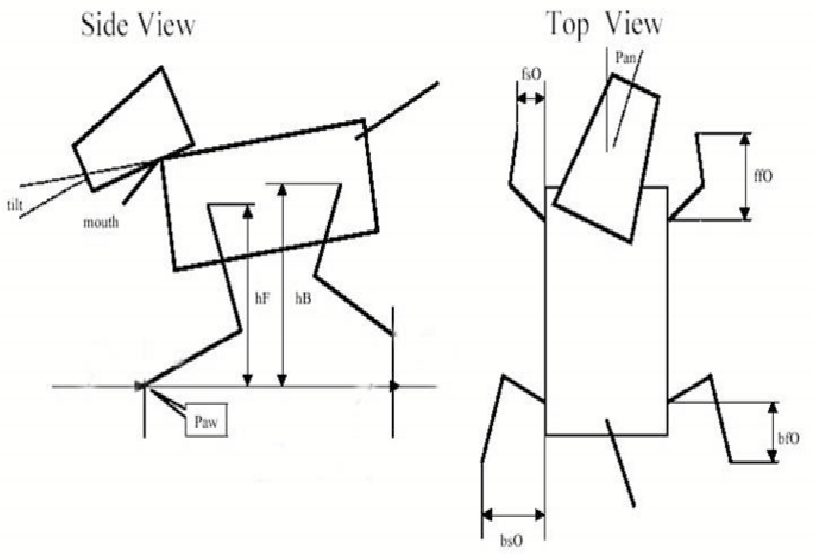

Figure 6.9: Parameters related to initial paw locations [30].

There are two different types of ES algorithms. The first one generates the next population by using only current offsprings, its notation is (μ/ρ ,μ)-ES and it is called as *comma-selection* while in the other type both the offsprings and the parents are used for the generation of next population, its notation is ($\mu/\rho + \lambda$)-ES and it is called as *plus-selection*. In these notations, μ represents the number of parents, ρ represents the number of individuals used during cross-over and λ represents the number of offsprings.

ES uses randomness to find solution for the optimization problem. There are four main functions which are used by ES optimization algorithm which are 'reproduction', 'recombination', 'mutation' and 'selection'. The most important difference between other Evolutionary Algorithms and ES is that mutation and recombination are applied to both parameters and their maximum mutation amounts. So, mutation amounts change and converge to the their optimum values.

ES engine is implemented as a different workspace called 'ES'. As the selection method, we have decided to use comma-selection, so only the parents of the current population are used to generate the next population. In our algorithm, stopping criterion is the generation number. The user defines the maximum number of generations and when the number of current generation is equal to that number, the simulation is terminated. The maximum and minimum values for both genes and sigma which determine the maximum value of mutation for the corresponding gene are also determined by the user. By this way, it is tried to reduce the size of the domain. During

the recombination process, there are some alternative methods, like getting the average of the parents. In this project, discrete cross-over method is applied. This method chooses one of the genes of corresponding parents, which are chosen for cross-over operation, randomly and sets its gene as the gene of the corresponding offspring.

Class diagram of the Engine is as follows:

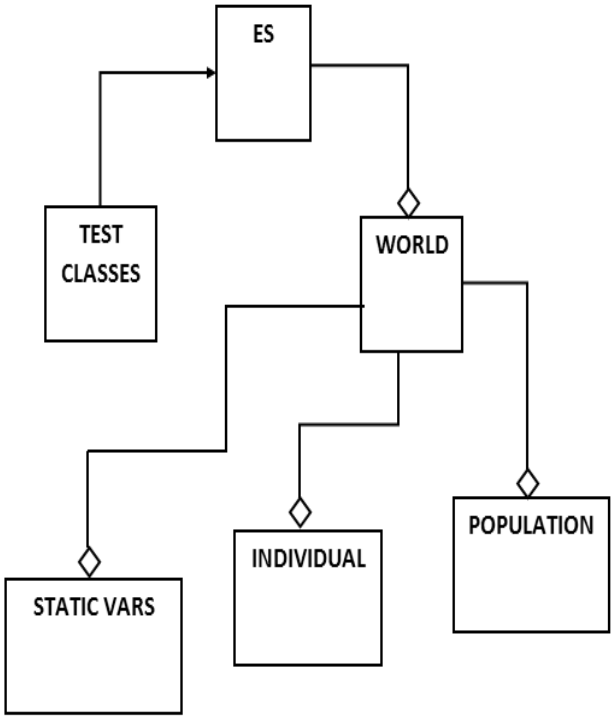

Figure 6.10: Class Diagram of ES Engine.

Flow chart of the main code is as follows:

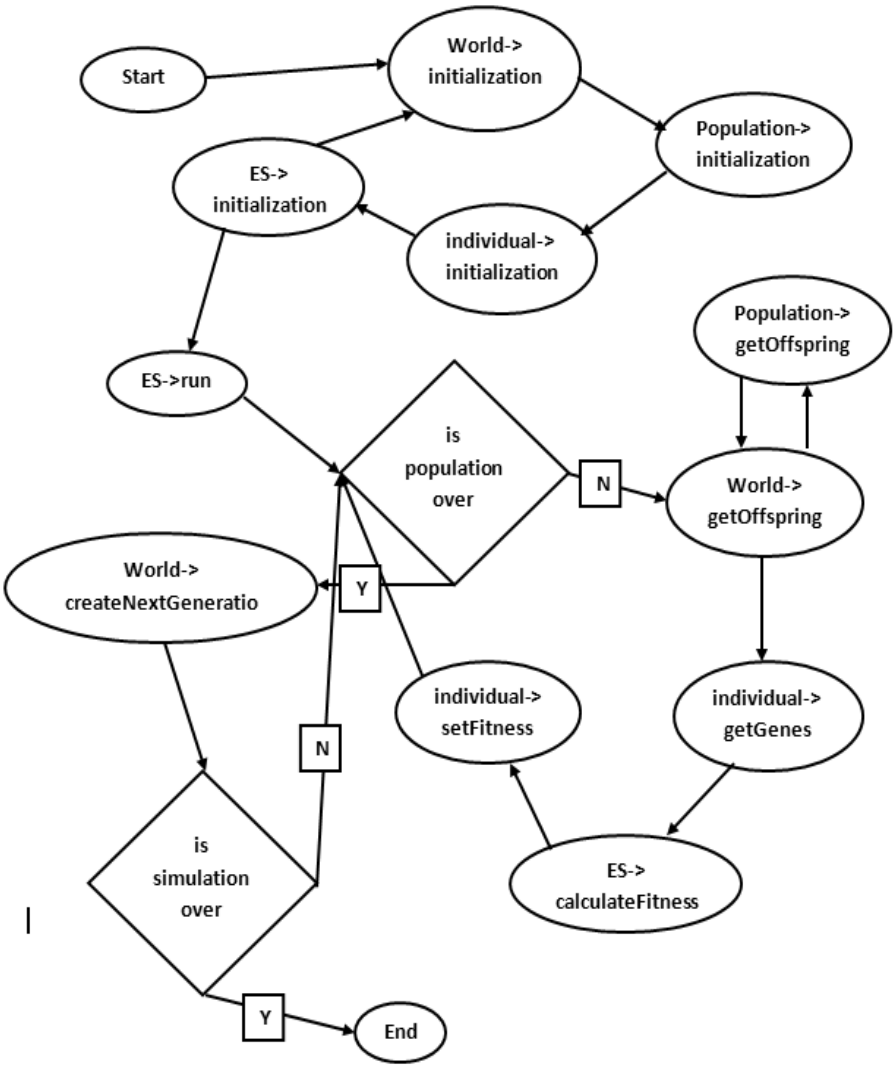

Figure 6.11: Flow Chart of Main.

Flow chart of the generation of the new population is as follows:

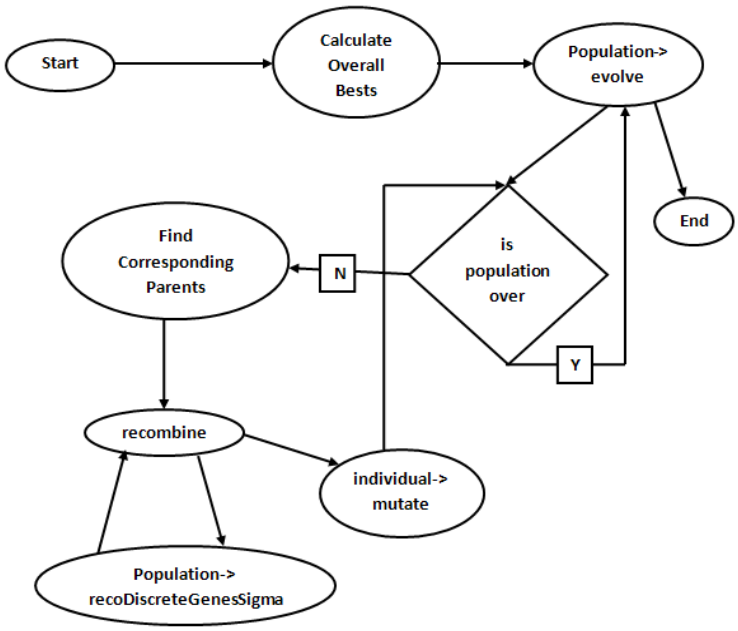

Figure 6.12: Flow Chart of Generation of New Population.

For the simulation process, a simple C++ source file with three static and one main method which are required by the simulator, Webots [33], are implemented. In Webots, three methods should be loaded for initialization, run and destruction of the execution. In the main, 'robot_live', 'robot_die' and 'robot_run' functions are called to determine the names of corresponding functions. For the initialization of the system, 'reset' static method is implemented. In this method, joint pointers are assigned, a pointer for motion manager is generated to calculate joint angles, a pipe between supervisor and robot is constructed and robot is placed at the beginning position where is at the middle of the yellow goal. This function is called at the beginning of the simulation. At each step of the simulation, 'run' method is called. In this method, initialization of the first generation is controlled and if it is not, the first generation is constructed by supervisor according to the data in initialization file and its first individual is sent to the robot via the buffer. This initialization is done in 'run' method, because the buffer cannot be used in 'reset'. After this control, currently loaded parameter set is run for a determined amount of time,'MOTIONTIME'. When the time is up, the robot is set to the initial position by the supervisior. This process is applied for 'NUMOFTRY' runs. For each run, a fitness value is calculated by the supervisor and stored in an array. The supervisor calculates overall the fitness value by getting the average of all fitness values without

the maximum and minimum values. This overall fitness value is set as the fitness value of current parameter set. After all individuals are executed by the robot and the overall fitness values are calculated by the supervisor, the supervisor generates the next population according to these values and resets the index of the current individual. Generation of the next population is handled by the 'ES' engine. During simulation, overall best individuals are stored and currently calculated ones are appended to the 'output.txt' file just before execution of next generation is started. For the destruction of the system, 'die' method is implemented. Because it is called only at the end of the simulation, nothing is done in this function.

There were two options for the implementation of this training project. The first approach was to embed the 'ES' engine and let the robot make the training itself, whereas the second approach was to make the training under the control of the station which runs on a PC. Both approaches have advantages and disadvantages. But, we have chosen second approach. The first approach had the advantage of working standalone, but running 'ES' is very time consuming. This calculation can be done on a PC much more faster than on robot. So, 'ES' is run on a PC and the robot is used only as an actuator of the parameter set. At that point, it is assumed that a reliable network exists between the robot and the corresponding PC. While fitness value is calculated, following process is followed:

- Station sends the parameter set
- Robot gets the set
- It goes to one beacon and gets ready to go another beacon
- Robot starts to go that beacon and counts the number of steps left during this process
- Robot reaches that beacon and sends counter to the station as the inverse of fitness value.

To be able to make this training, we need only two tools in addition to 'ES' engine: a player and a graphical user interface (GUI).

Player part of the project was the program which runs on the real robot. In this program, the robot waits for the message of the station. After the robot gets the station's message which contains the new parameter set, robot starts the timer and goes to the next beacon. When it reaches the beacon, it calculates the number of steps left and sends it as the inverse of fitness value. At that point, the robot starts to be ready for the next individual. 'receiveDebugMessage' function of CoreObject is the entry point of the messaging mechanism between CerberusPlayer and CerberusStation. 'msgWALKTRAIN' is the type of the message and if this message comes, CoreObject switches the state of the robot to go to next beacon. When the

robot reaches to the beacon or time is out, Planner sets the flag of sending fitness value. This flag activates 'sendFitnessValue' function and this function sends the fitness value of parameter set to the CerberusStation.

As the GUI, a frame is added to the CerberusStation. In this frame, only the initialization file is determined by a textbox. In addition to this, there are two buttons to start and stop training. There is no more need of human control on training. For debugging purposes, a label is placed at the bottom of the GUI. For current generation number and currently overall best fitness, two additional textboxes are added to the frame. Apparence of the GUI is as follows:

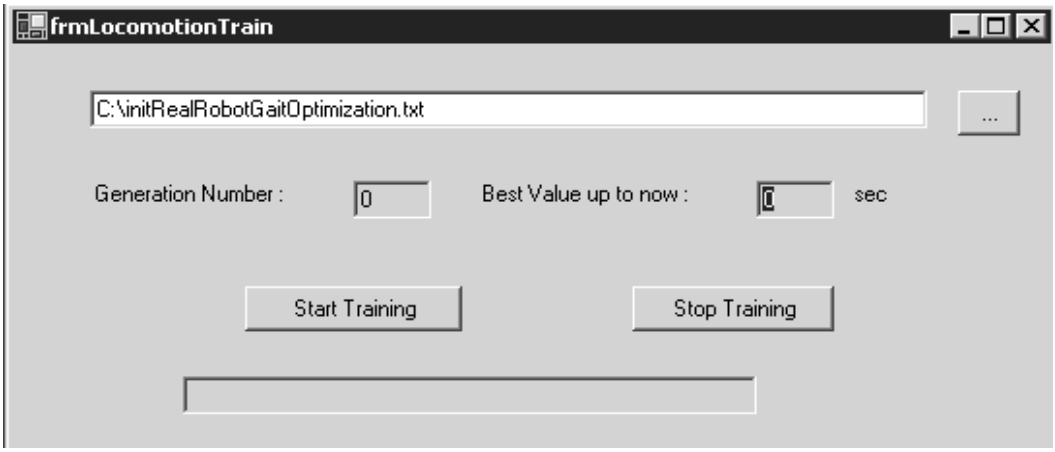

Figure 6.13: GUI of the Engine.

This frame adds itself to the image and data receivers of the 'robotConnection'. Image receiver is not used in this training. When a message comes from CerberusPlayer, it comes to the 'OnDataReceived' method and the correct message is found according to the header. Because new parameter set is sent after the fitness value of previous set has arrived, 'ES' engine is embedded to this method. This embedded situation is very similar to that of SimPlayer. The only difference is that the trial of the same parameter set is handled by the robot. So, overall fitness value comes to the CerberusStation, and CerberusStation calculates the next generation.

In initialization file, many parameters of the engine are defined. There are two main aims to use initialization file. First one is the modularity. Second aim is not to compile again and again, when one parameter of the engine is changed. For example, when number of maximum generations is changed or base genes are changed, there is no need to recompilation, because these values are read from the file and not hard coded. Structure of the initialization file is as follows:

- Number of parameters

- Number of all parents in one population
- Number of parents used for cross over
- Neighborhood which is used to find parents for cross over
- τ_i value
- τ_o value
- Number of overall best individuals stored
- Maximum generation number
- Base genes
- Maximum value of genes
- Minimum value of genes
- Minimum σ
- Maximum σ

Experiments of the ES engine are performed on two different environments namely the simulator, and the robot. During the experiment on the simulator, the initial population is generated randomly and they are fully-distributed to the domain of parameter set. So, the individuals start from very distant points of the domain. Fitness criterion of the simulation is the distance the robot moves with the corresponding parameter set. At the beginning of the simulation, the fitness value of the overall best individual is very low. But, there is a very rapid increase on the fitness value of the overall best individual. After a point which is around 20, the increase on the overall fitness value is not as much as before. After this generation, the best fitness value converges to 2.0. Overall best fitness value exceeds the value of 1.9 and it is less than 2.0 for the rest of the simulation. The value of the fitness is given the forward distance between starting point and reached point of the robot along the x coordinate. The results of the simulation are as follows:

The second experiment is performed on the robot. In this experiment, the initial population is not fully distributed, but base individual, which is given in the input file, oriented. Lack of time, resources and danger of breaking legs are the reasons to generate initial population around the base individual. The fitness criterion of this experiment is the time of movement of the robot between two beacons defined as 1000 / number of steps of planner during movement. The reason to multiply with 1000 is to get bigger values which can be used easily to compare fitness values of two individuals. As a result of the experiment, a very slight improvement has occurred.

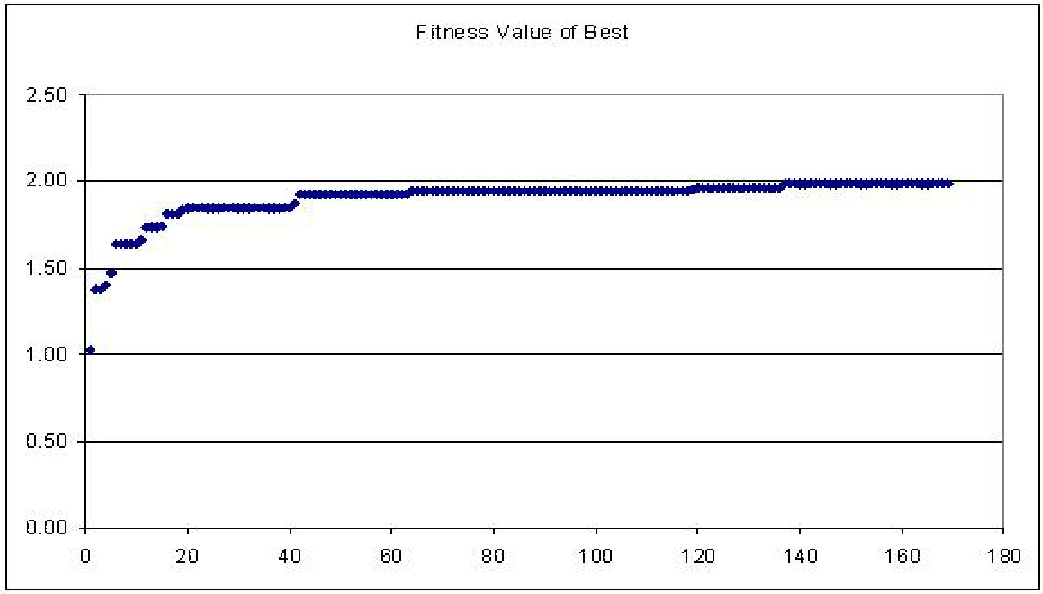

Figure 6.14: Results of Simulation on Webots.

Generation of initial population around base individual is the main reason to get such a slight improvement. In addition to close initial population to a point, an important problem of this experiment can be seen as the reason to have such a situation. This problem is the loss of connection between robot and station. When the robot loses the connection with station, it starts to wait parameter set, and sends the fitness value to get new parameter set again. At that point, two parameter sets may be received. So, one of the parameter sets have fitness value without try. So, some successful parameter sets may be lost because of this problem. Two solutions can be applied to solve this problem. The first solution is to run ES on the robot, so there would be no need of connection between the robot and the station. Second possible solution is to send timestamps. But this structure makes the problem more complicated and more data are exchanged between the robot and the station. So, this makes the system slower. A rapid increase at the end the experiment may give the attitude of a better increase for the future, but more experiments with more generation should be collected to have such a conclusion. Results of the run on the robot is as follows:

Results of these two experiments are very different. There are two important differences between these two experiments. First of all, the robot moves only forward with maximum value on the simulator. So, the given fitness value does not guarantee the success of the turn and strafe speeds. In the second experiment, speed is calculated by considering the functions used to approach an object. So, the fitness value guarantees not only the success

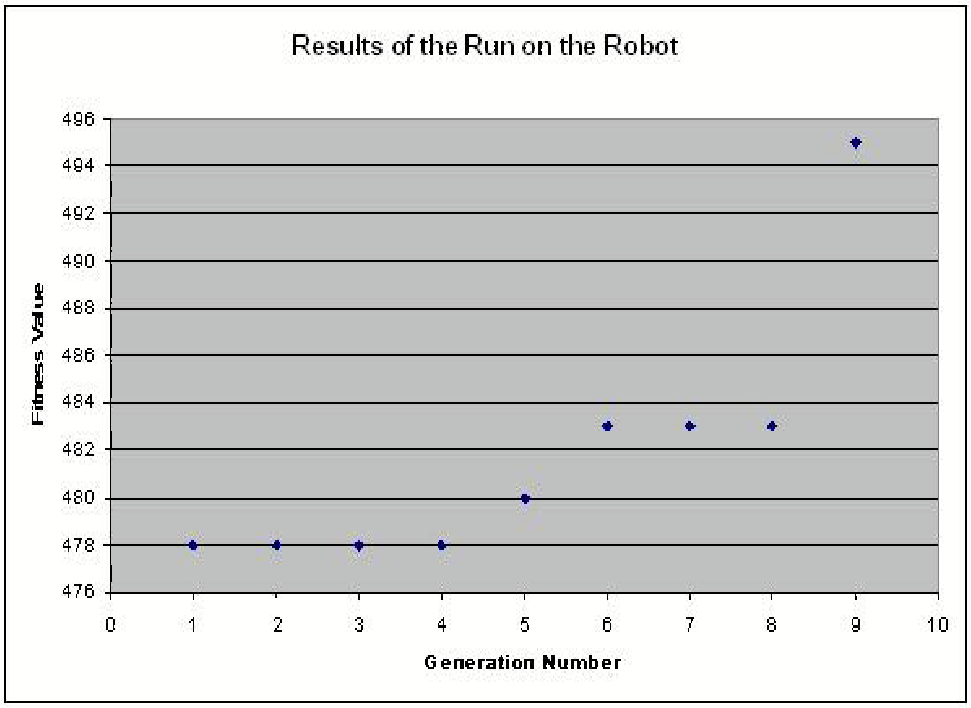

Figure 6.15: Results of Run on the robot.

of the parameter set but also the success of parameter set with the functions used to approach the object. Under this condition, the fitness value of the second experiment is more realistic than that of the first one. The second important difference is the applicability of the experiments. The first experiment can be applied to the whole range of the parameter set. But, the second experiment can be used only to adjust a reasonable parameter set. So, it can be concluded that the first experiment is more suitable to approach global optimum of the problem whereas the second experiment is to approach local optimum.

Chapter 7

Results

This year, we have achieved our best results in the soccer competition part by reaching to the quarterfinals. However, we could not repeat our success in the Technical Challenge competition due to some technical difficulties and failed to get an upper position in the ranking list.

7.1 Games

In the first round robin pool, our opponents were MS Hellhounds and Araibo. In the first game, we played against Araibo. We had serious vision problems making us to play without localization and some motion problems in turning with ball (which remained until the end of our games and later was discovered and fixed) but we undoubtedly had a speed advantage over them, so in most of the time we had the ball possession. However, we were unable to score a goal so the game ended with a score 0-0. The second game was against MS Hellhounds. In the first half, we showed a very good performance and were able to score a goal so the first half finished with a score 2-1. In the second half, again we stuck to the turning problem which gave the opponent forward players many chances to shoot against our goal and our goalkeeper suffered severely from localization problems. They scored four more goals and the game finished with score 6-1 in favor of MS Hellhounds. Fortunately, Araibo lost to MS Hellhounds 7-0 so we proceeded to the intermediate round.

Our opponent was The Impossibles in the intermediate round. They were a new yet good prepared team but we had a great speed and maneuverability dominance over them so the whole game was played with our possession over ball. We won 5-3 and we have scored some goals on our own goal due to the same turning problem.

In the second round robin pool, our opponents were NUBots, Hamburg Dog Bots and S.P.Q.R. In the first game, we played against S.P.Q.R. and were able to win the game with a score 2-1. In the second game, we played

against Hamburg Dog Bots and we won 4-1 in an easy game and we have guaranteed the quarter finals for the first time in our history. In our last game, NUBots beat us easily 10-0.

In the quarter finals, we played against the last two year's champion, German Team. Despite the final score which is 9-0 in favor of them, we played fairly good against them but our problem with turning with the ball and localization problems that our goalkeeper has encountered led them to such a victory.

We have lost only three games in the soccer competition and those three teams were the eventual 1^{st}, 2^{nd} and 4^{th} place teams. We were one of the two teams who managed to score against MS Hellhounds and the other team was NUBots, which won the competition.

7.2 Technical Challenges

This year, we could not repeated our success in 2005 due to some technical problems we have faced.

7.2.1 Open Challenge

Our open challenge topic was a speech recognition module embedded to an Aibo which enables people to give certain commands like 'sit', 'stand up', 'walk', 'roll', and so on as they would give to a living pet dog. The major strongness of our system was its speaker independence. However, we wrongly underestimated the base noise level in the competition site so we could not make the dog distinguish a given command from the background noise and hence, we failed to present our system.

7.2.2 Passing Challenge

The passing challenge was one of our best prepared challenges. Two students spent nearly an entire semester for developing the necessary communication infrastructure and an efficient strategy for coordinating the three robots. The challenge code needed some last minute touches and bugfixes when we departed for the RoboCup. However, we made it to the quarterfinals so we didn't have enough time to fix the bugs and we have failed to make any successful passes during the challenge. Hence we have finished this challenge without any points.

7.2.3 New Goal Challenge

For new goal challenge, we worked on two modules. As the first module, we have updated perceptor of the goal. Because the middle of the goal is empty, most of the belief of the goal is determined by the ratio of number

of pixels inside of the region to the area of the it. The area of the candidate region gives us the rest of our belief for the corresponding region to be the region of the goal. As second module, planner has been updated. Flow chart of the new goal challenger is very similar to that of primary attacker. The only difference is the object to search when the grab process is successful.

We have misunderstood a rule of the challenge. We have assumed that the static robots looks forward. But, we have seen that it was not a rule. During the challenge, the robots were placed in different orientations, so they reduced the area to be able to shot. We had a very close shot to the goal. From here, we can conclude that perception has been successful for the game, but the planner was not, and our score for this challenge is zero.

Bibliography

[1] Yıldız, O. T, L. Akarun and H. L. Akın, "Fast nearest neighbour testing algorithm for small feature sizes", *Electronics Letters*,Vol 40, No 3, pp. 171-172, February 2004.

[2] Schioler, H. and U. Hartmann, "Mapping Neural Network Derived from the Parzen Window Estimator", *Neural Networks*, 5, pp. 903-909, 1992.

[3] Akın, H. L.,Ç. Meriçli, T. Meriçli, K. Kaplan, and B. Çelik, *Cerberus 2005 Team Report*, 2005

[4] Kose, H and H. L. Akın, "Experimental Analysis And Comparison Of Reverse-Monte Carlo Self-Localization Method", CLAWAR/EURON Workshop on Robots in Entertainment, Leisure and Hobby, December 2 – 4, 2004, Vienna, Austria, pp, 85-90.

[5] Kose, H and H. L. Akın, "Robots From Nowhere," *RoboCup 2004: Robot Soccer World Cup VIII,* LNCS 3276, pp.594-601, 2005.

[6] Kose, H and H. L. Akın,"A fuzzy touch to R-MCL localization algorithm", *RoboCup 2005: Robot Soccer World Cup IX*, LNCS Vol. 4020, pp. 420 - 427, 2006.

[7] Kaplan, K., B. Çelik, T. Meriçli, Ç. Mericli and H. L. Akın, "Practical Extensions to Vision-Based Monte Carlo Localization Methods for Robot Soccer Domain", *RoboCup 2005: Robot Soccer World Cup IX*, LNCS Vol. 4020, pp. 420 - 427, 2006.

[8] Kose, H., , B. Çelik and H. L. Akın, "Comparison of Localization Methods for a Robot Soccer Team", *International Journal of Advanced Robotic Systems*, Vol. 3, No. 4, pp.295-302, 2006.

[9] Hightower, J., and G. Borriello, "A Survey and Taxonomy of Location Systems for Ubiquitous Computing", Technical Report UW-CSE Tech Report No:01-08-03, 2001.

[10] Buschka, P., A. Saffiotti, and Z. Wasik, "Fuzzy Landmark-Based Localization for a Legged Robot" Proc. of the IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems (IROS) Takamatsu, Japan, July 2000, pp. 1205-1210, 2000.

[11] Saffiotti, A., A. Bjorklund, S. Johansson, and Z. Wasik, "Team Sweden", RoboCup 2001: Robot Soccer World Cup V, Springer-Verlag, Seattle, Washington, Lecture Notes in Computer Science Series, Vol. 2377, pp 725-729, 2002. 2001.

[12] Stroupe, A.W., and T. Balch, "Collaborative Probabilistic Constraint Based Landmark Localization", Proceedings of the 2002 IEEE/RSJ Intl. Conference on Intel- ligent Robots and Systems EPFL, Lausanne, Switzerland, pp. 447-452, 2002.

[13] Stroupe, A.W., K. Sikorski, and T. Balch, "Constraint-Based Landmark Localization", RoboCup 2002: Robot Soccer World Cup VI, Springer-Verlag, Fukuoka, Busan, Lecture Notes in Computer Science Series, Vol. 2752, pp 8-24, 2003. 2001.

[14] Fox, D., W. Burgard, and S. Thrun, "Markov Localization for Mobile Robots in Dynamic Environments", Journal of Artificial Intelligence Research, Vol. 11, pp. 391-427, 1999.

[15] Schulz, D., and W. Burgard, "Probabilistic State Estimation of Dynamic Objects with a Moving Mobile Robot", Robotics and Autonomous Systems 34, Elsevier, pp. 107-115, 2001.

[16] Thrun, S., D. Fox, W. Burgard, and F. Dellaert, "Robust Monte Carlo Localization for Mobile Robots", Artificial Intelligence, Elsevier, Vol. 128, pp. 99-141, 2001.

[17] Schulz, D., and W. Burgard, "Probabilistic State Estimation of Dynamic Objects with a Moving Mobile Robot", Robotics and Autonomous Systems, Elsevier, Vol. 34, pp. 107-115, 2001.

[18] Lenser, S., and M. Veloso, "Sensor Resetting Localization for Poorly Modelled Mobile Robots", Proc. ICRA 2000, IEEE, Vol. 2, pp. 1225-1232, 2000.

[19] Gutmann, J.S., and D. Fox, "An Experimental Comparison of Localization Methods Continued", In Proc. of the 2002 IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS02), Lausanne, Switzerland, pp 454-459, 2002.

[20] Gutmann, J. S., "Markov-Kalman Localization for Mobile Robots", Int. Conf. on Pattern Recognition (ICRP), Vol. 2, No. 2, pp. 601-604, 2002.

[21] Kaplan, K. and H. L. Akın, "A Controller Design for Soccer Robot Teams", *IJCI Proceedings of International XII Turkish Symposium on Artificial Intelligence and Neural Networks TAINN 2003*, 1, 1, July 2003.

[22] Kose, H., Ç. Meriçli, K. Kaplan and H. L. Akın, "All Bids for One and One Does for All: Market-Driven Multi-Agent Collaboration in Robot Soccer Domain", *Computer and Information Sciences-ISCIS 2003, 18th International Symposium Proceedings*, LNCS 2869, pp. 529-536, 2003.

[23] Kose, H., K. Kaplan, C. Mericli and H. L. Akın, "Genetic Algorithms Based Market-Driven Multi-Agent Collaboration in the Robot-Soccer Domain", FIRA Robot World Congress 2003, October 1 - 3, 2003, Vienna, Austria.

[24] Kose, H, U. Tatlidede, C. Mericli, K. Kaplan and H. L. Akın, "Q-Learning based Market-Driven Multi-Agent Collaboration in Robot Soccer", Proceedings, TAINN 2004, Turkish Symposium On Artificial Intelligence and Neural Networks, June 10-11, 2004, Izmir, Turkey, pp.219-228.

[25] Anon., "Chain-of-responsibility pattern", *Wikipedia*, 2007.

[26] Anon., "Factory method pattern", *Wikipedia*, 2007.

[27] Anon., "Aspect-oriented programming", *Wikipedia*, 2007.

[28] Anon., "Matlab Fuzzy Logic Toolbox", *Mathworks, Inc.*, 2007.

[29] Hengst, B. D. Ibbotson, S. B. Pham, and C. Sammut "Omnidirectional Locomotion for Quadruped Robots", *RoboCup 2001 : Robot Soccer World Cup V* , LNAI vol. 2377, pp. 368-373, 2002.

[30] UNSW 2003 team report "http://www.cse.unsw.edu.au/∼robocup/report2003.pdf"

[31] UNSW 2000 team report "http://www.cse.unsw.edu.au/∼robocup/2002site/2000PDF.zip"

[32] Beyer, H. G., *Theory of Evolution Strategies*, Springer, 2001.

[33] Michel, O. "Webots: Professional Mobile Robot Simulation", *International Journal of Advanced Robotic Systems*, Vol. 1, pp. 39-42, 2004.