Cerberus'10 Team Description Paper

H. Levent Akın, Tekin Meriçli, Ergin Özkucur, Can Kavaklıoğlu, and Barış Gökçe

Boğaziçi University, Department of Computer Engineering
34342 Bebek, İstanbul, TURKEY
{akin, tekin.mericli, nezih.ozkucur, can.kavaklioglu, sozbilir}@boun.edu.tr

1 Introduction

The **Cerberus** team made its debut in the 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 in all RoboCups between 2001 and 2009 except the year 2004. 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. The team competed in both RoboCup 2008 and RoboCup 2009 competitions with the Nao robots. In 2009, Cerberus made it to the second round and also got the fifth place in the technical challenges.

From the very beginning, Cerberus has chosen to develop all the components of the software to form the basis for a more general robotics research rather than to be used for soccer only. Through the years, the members of the team have done many PhD and MS Thesis studies related to SPL and published more than 40 papers in journals and international conferences, including the RoboCup Symposia [1].

The organization of the rest of the report is as follows. The software architecture is described in Section 3. In Section 4, the details of the vision module are provided. Self localization method is described in Section 5. The locomotion module and gait optimization methods used are explained in Section 6. Finally, various approaches we use for the planning module are described in Section 7.

2 Cerberus Team

- **Team Advisor:** H. Levent Akın
- **Team Members:** Tekin Meriçli, Ergin Özkucur, Can Kavaklıoğlu, and Barış Gökçe

3 Software Architecture

The software architecture of Cerberus has been completely re-designed and implemented starting from 2008. The existing modular architecture was transformed into

[1] The full list of Cerberus publications are available here:
`http://robot.cmpe.boun.edu.tr/~cerberus/wiki/index.php?n=Publications.Publications`

a more general library architecture, where the code repository is separated into levels in terms of generality. Similar to the well known Model-View-Control architecture, the main goal of this new approach was to organize our code base into logical sections, all of which are easy to access, manipulate, and debug. The rewrite process was originally targeting the Aibo platform but the well designed architecture has made our initial development on Nao painless and quick.

Software architecture of Cerberus consists of mainly three parts:

- BOUNLib
- Cerberus Player
- Cerberus Station

Figure 1 illustrates the general structure of our most generic software architecture, including the Cerberus code base.

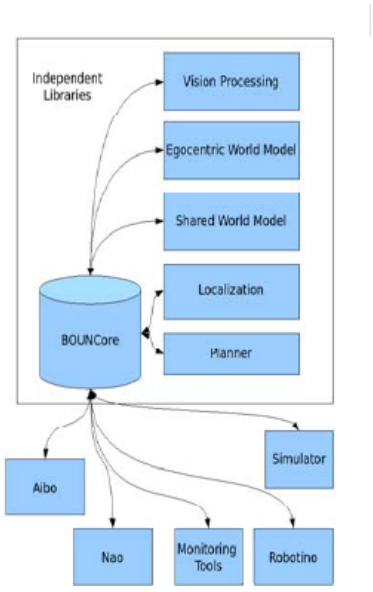

Fig. 1. BOUNRobotics software architecture.

3.1 BOUNLib

Past experience has demonstrated the previously used modular approach to be sub-optimal in some cases. Reuse of source code for multiple architectures and also multiple purposes, and making specific modifications to the special purpose modules is very time consuming and error prone.

We collected the more general parts of our code base in a library structure called *BOUNLib*. Using this library enables us to develop software for different platforms or different robots easily by reusing most of our code base, as illustrated in Figure 1.

3.2 Cerberus Station

BOUNLib library includes a versatile input output interface, called *BOUNio*, providing essential connectivity services to the higher level processes such as reliable UDP protocol, file logging, and TCP connections. Connections are made seamlessly to the sender, thus there is no need to write specific code for any application or test case.

Using *BOUNio* library enabled us to implement a very general version of our previous *Cerberus Station* using Trolltech's Qt Development Framework [1]. It is very easy to test new features to be added to the robot using the well structured architecture of our runtime code and *Cerberus Station*. This is a very vital resource for any experiment involving robots.

The new *Cerberus Station* has the same features of old its older version and more, mainly aimed at visualizing the new library based code repository, some of which are listed below:

1. Record and replay facilities providing an easy to use test bed for our test case implementations without deploying the code on the robot for each run.
2. A set of monitors which enable visualizing several phases of image processing, localization, and locomotion information.
3. Recording live images, classified images, intermediate output of several vision phases, objects perceived, and estimated pose on the field in real time.
4. Log to file and replay at different speeds or frame by frame.
5. Locomotion test unit in which all parameters of the motion engine and special actions can be specified and tested remotely.

The screen shot in Figure 2 demonstrates some of these features of the new *Cerberus Station* software.

3.3 Alternative Architecture

We also tried developing an alternative architecture, where it was possible to create the fields (i.e. memory entries) at run time and attach events to those fields such that a related function would be called when the value of the corresponding field is changed. The architecture also supported callback mechanism. However, due to real-time execution constraints, we decided not to switch to this new architecture.

4 Vision

4.1 Image Processing and Perception

The purpose of the perception module is to process the raw image and extract available object features from it. The input to the module is the image in YUV422 format and the output is the range and bearing values of the perceived objects and landmarks.

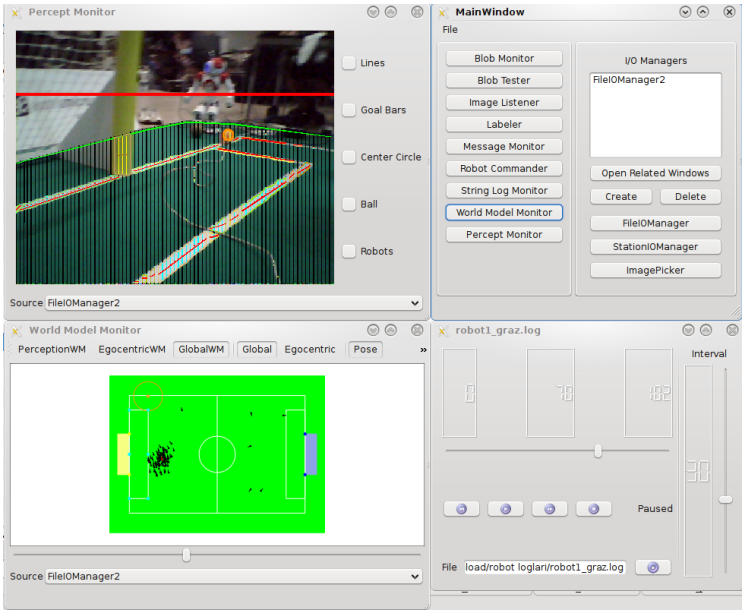

Fig. 2. The new Cerberus Station software.

Color Quantization In the raw image format, each pixel is represented with a three-byte value and can be one of the 255^3 values. Since it is impossible to efficiently operate in such an input space, the colors are quantized into a smaller set of pseudo-colors, namely, white, green, yellow, blue, robot-blue, orange, red, and "ignore". We utilize a Generalized Regression Neural Network (GRNN) [2] for mapping the real color space to the pseudo-color space.

In order to obtain the outputs of the trained GRNN in a time-efficient manner, a look up table is constructed for all possible inputs. *Y*, *U*, and *V* values are used to calculate the unique index and the value at that index gives the color group ID to determine the color group of a pixel. Figure 3 shows a screen shot from the *Labeler* component of the *Cerberus Station* software, where it becomes possible to visually evaluate the resulting classification performance of the GRNN right after the labeling and training phases.

Scanline Based Perception Framework Considering that the cameras of the Nao robots provide higher resolution images and the processors are slower compared to those of the Aibo robots', it becomes infeasible to process each pixel to find the objects of interests in the image due to computational efficiency and real-time constraints. Therefore, scan lines are used to process the image in a sparse manner, hence speeding up the entire process.

The process starts with the calculation of the horizon based on the pose of the robot's camera with respect to the contact point of the robot with the ground, that is the base foot of the robot. After the horizon is calculated, scan lines that are 5 pixels apart from

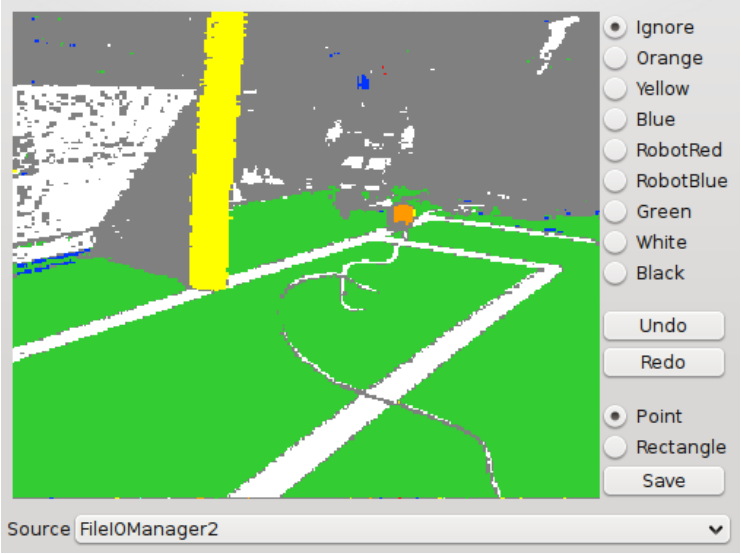

Fig. 3. A classified image constructed with a trained GRNN.

each other and perpendicular to the horizon line are constructed, such that they originate on the horizon line and terminate at the bottom of the image. The first step after that is to scan through these scan lines to find where the green field starts, which is done by checking for a certain number of consecutive green pixels along the line. Of course that results in a green region where all non-green parts that are close to the edges of the field ignored, such as the goal posts and balls that are on the border lines. In order to not lose information about those important objects, a convex-hull is formed for the starting points of the green segments. That way, we define the real green field borders where all objects of interests fall inside; hence, we can basically ignore, say all orange regions, that are outside the field borders. That provides a natural way of pruning false percepts without having to process them beforehand. After the field borders are constructed, the shorter scan lines are extended back to these borders, so that it is possible to use them to detect the goal posts and balls that are close to the borders.

After all these constructions and corrections, each scan line is traced to find colored segments on them. After only one pass over these scan lines, we end up with groups of segments with the colors we are interested in, namely, orange, white, blue, and yellow. The next step is to build regions from these segments, based on the information on whether two consecutive segments "touch" each other, that is they are on two consecutive scan lines and either of them has a start or end point within the borders of the other one. Two consecutive touching segments are merged into a single region. For white segments though, there are some additional conditions, such as change in direction and change in length ratio. These additional constraints guarantee that all field lines are not merged into a single, very big region, but instead into smaller and more distinctive regions. After the construction of these regions, they are passed to the so called the *region*

analyzer module to be further filtered and processed for the detection of the ball, the field lines and intersections of them, and the goal posts. Figure 4 shows the result of this processing, which takes less than 15 ms on the average. The thick red line represents the calculated horizon, the thin green line group represents the convex-hull, which corresponds to the green field border, thin red lines represent the white line segments to be further processed, the yellow line group represents the yellow goal post base, and the orange circle represents the detected ball. The egocentric positions of these objects are computed using the camera matrix and projecting them back on the field.

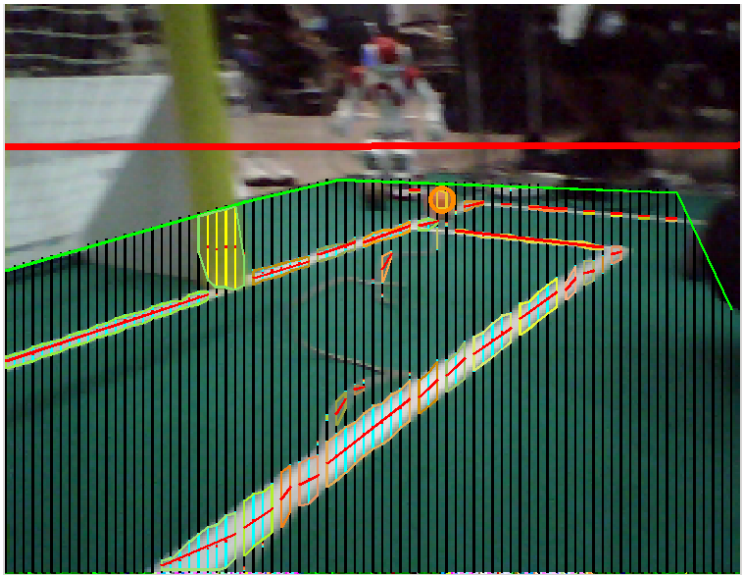

Fig. 4. Result of processing the image using scan lines.

4.2 World Modeling and Short Term Observation Memory

The perception module of Cerberus provides instantaneous information. While the reactive behaviors like tracking the ball with the head requires only instantaneous information, other higher level behaviors and the localization module needs more than that.

The planning and localization modules require perceptual information with less noise and more complete information. The world modeling module should reduce sensor noise and complete the missing state information by predicting the state. This is a state prediction problem and we use the most common approach in the literature, the Kalman Filter [3], for solving this problem.

In our setting, the observations are the distance and the bearing of the objects with respect to the robot origin, and the state we want to know consists of the actual distance and bearing information. In addition to that, for dynamic objects like the ball, the state vector also includes distance change and bearing change information to aid prediction.

For any object, the observation is $z = \{d, \theta\}$ where d and θ are distance and bearing, respectively, to the robot origin. For the stationary objects, the state is $m = \{d, \theta\}$ and the state evolution model is $m^1_{k+1} = I \times m_k$ and $z_k = I \times m_k$ where k is time and I is the unit matrix.

For the dynamic objects, the observation is the same but the state is represented as $m = \{d, \theta, d_d, d_\theta\}$ where d_d is the change in distance in one time step and d_θ is the change in bearing likewise. The state evolution model is:

$$\begin{pmatrix} d_{k+1} \\ \theta_{k+1} \\ d_{d,k+1} \\ d_{\theta,k+1} \end{pmatrix} = \begin{pmatrix} 1\,0\,1\,0 \\ 0\,1\,0\,1 \\ 0\,0\,1\,0 \\ 0\,0\,0\,1 \end{pmatrix} \begin{pmatrix} d_k \\ \theta_k \\ d_{d,k} \\ d_{\theta,k} \end{pmatrix}$$

and the observation model is:

$$\begin{pmatrix} d_{k+1} \\ \theta_{k+1} \end{pmatrix} = \begin{pmatrix} 1\,0\,0\,0 \\ 0\,1\,0\,0 \end{pmatrix} \begin{pmatrix} d_k \\ \theta_k \\ d_{d,k} \\ d_{\theta,k} \end{pmatrix}$$

As can be observed from the model specifications, we omit the correlation between the objects and use filter equations for each object separately. If an object is not observed for more than a pre-specified time step, the belief state is reset and the object is reported as *unknown*. For our case, this time step is 270 frames for stationary objects and 90 frames for dynamic objects.

In the update steps, odometry readings are used. The odometry reading is $u = \{d_x, d_y, d_\theta\}$ where d_x and d_y are displacements in egocentric coordinate frame and d_θ is the change in orientation. When an odometry reading is received, all the state vectors of known objects are geometrically re-calculated and the associated uncertainty is increased.

The most obvious effect of using a Kalman Filter is that the disadvantage of having a limited field of view is reduced. As the robot pans its head, it can be aware of distinct landmarks which are not in the same field of view at the same time.

The world modeling approach described here has an important shortcoming. It assumes that the identities of the observations are known. However with the new rules, such observations are rare, the observations often come without identity information. This problem is addressed in the localization module.

5 Self Localization

Cerberus employs vision based Monte Carlo Localization (MCL). In the MCL algorithm, the belief state is represented by a particle set and each element represents a possible pose of the robot. In Figure 5, a sample belief state is given. We use MCL with a set of practical extensions (X-MCL) which is detailed in [4]. Until last year, we used the output of the world modeling module as input to the localization module. Namely the filtered landmarks are used as observations for the localization module (Figure 6). However, this year we are modifying the algorithm to filter unidentified observations.

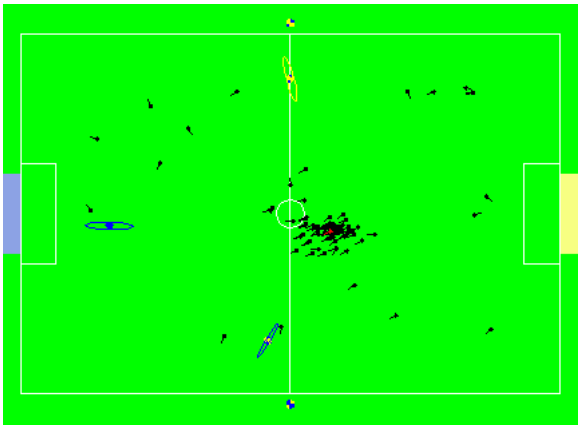

Fig. 5. Belief state of the robot in MCL algorithm.

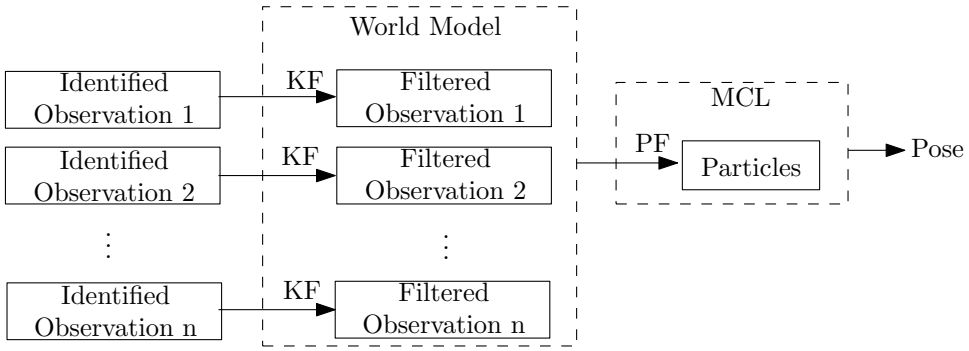

Fig. 6. Old localization algorithm.

The new approach is inspired from FastSLAM [5] algorithm and Multi-Hypothesis tracking [6]. In FastSLAM, each particle has its own world model (i.e. map). In Multi-Hypothesis tracking, there are multiple Gaussians where each relies on a different data association sequence and their numbers are limited by pruning. Let the *world model* be the set of Kalman filters for each landmark. In our localization approach, we maintain multiple world models with respect to different observation associations. And each particle is associated to the world model which is the most likely one for the pose particle represents. The overview of the new approach is given in Figure 7. With the new algorithm, we will be able to use line corners and partially observed bars for self localization.

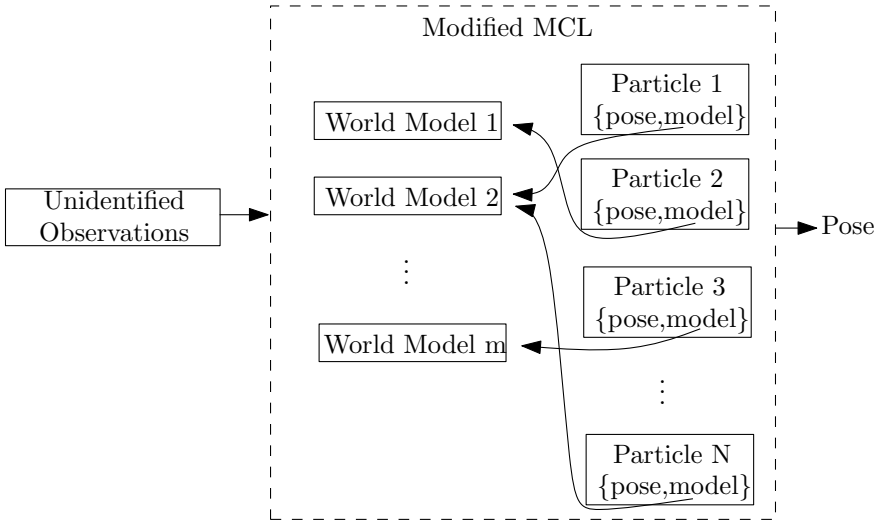

Fig. 7. New localization algorithm.

6 Motion

Our motion engine has two different infrastructure components, which allow different levels of abstractions.

- **Body:** The first infrastructure component determines the properties of the body of the robot and provides related functionalities; forward and inverse kinematic models, etc. It is composed of *Manipulators* which are legs, arms, and head. A *manipulator* is used for the functionality of the end effectors. A *Manipulator* is a combination of *Links*, which are used to store the transformation matrix, angle, and other related kinematic description parameters of each joint. Additionally they store the mass of the link connected to the next joint. This information is used for the calculation of center of mass and the zero-moment point. In order to be platform independent and generic, this infrastructure is defined to be independent of the robot hardware. The definition of a platform is stored in configuration files by giving the number of manipulators and links, kinematic parameters, masses for each link, etc.
- **Locomotion:** A root engine is defined at the very top level and the common properties and functions of each different locomotion engine are included in it. Different locomotion engines are inherited from the root engine. Platform and locomotion algorithm specific parts are defined in the inherited engines. For the Nao robot, three main features are implemented. The first one is a dynamic walking feature. A signal generation-based algorithm, which is very similar to Central Pattern Generator [7] is used. The motion of the head is separated from the motion of the rest of the body and implemented as the second component. The last feature is the motion player,

which reads the sequential joint angles from pose definition files and plays them to realize some special actions, such as kicking the ball and standing up from a fallen position.

6.1 Bipedal Locomotion

Model-Driven CPG-Based Biped Walking A walking method based on that of the champion of the Humanoid League in the RoboCup07, NimbRo [8], is implemented. They defined three important features for each leg; leg extension, leg angle, and foot angle. Leg extension is the distance between the hip joint and the ankle joint. It determines the height of the robot while moving. Leg angle is the angle between the pelvis plate and the line from hip to ankle. It has three components; roll, pitch, and yaw. The third feature, foot angle, is defined as the angle between foot plate and pelvis plate. It has only two components; roll and pitch. Using these features helps us have more abstract calculations for the motion.

Before finding motion features, a central clock (ϕ_{trunk}) is generated for the trunk which is between $-\pi$ and π. Each leg is fed with a different clock (ϕ_{leg}) with $ls \times \pi/2$ phase shift where ls represents leg sign and it is -1 for the left leg while $+1$ for the right leg. The synchronization of the legs can be preserved in this way. In the calculations of motion features at a given time, the corresponding phase value is considered and the values for features are calculated by using these phase values.

In order to find the leg angle and foot angle features, motion at each step is divided into five sub-motions; *shifting*, *shortening*, *loading*, *swinging*, and *balance*.

In the shifting sub-motion, lateral shifting of the center of mass is handled. For this purpose, a sinusoidal signal is simulated. The second important sub-motion is the shortening signal and it is not always applied. During the shortening phase, both a joint angle for the foot and a part of the leg extension value are calculated as a cosine function of the shortening phase value. The third sub-motion of the step is loading which is also not always applied. In this phase, only a part of the leg extension is calculated as that of shortening phase. Swinging is the most important part of the motion. In this part, the leg is unloaded, shortened and moved along the way of motion which reduces the stability of the system considerably. This movement has effects on each component of the leg and the foot angle features of the motion. As the last component of the step, which is balance, correction values for the deviations of the other operations are added to the system from the foot angle feature and the rolling component of the leg angle feature. At the end, the corresponding parts of the sub-motions are added, and the values for the motion features are calculated.

Because balance is not guaranteed in the model, and it is impossible to optimize the model with the maximum speed analytically, our biped walking is defined in terms of some parameters. After determining a feasible parameter set by hand, we applied an optimization algorithm, *Evolutionary Strategies*, to fine-tune the walking motion. Although both speed and balance is used in the fitness function, our walk engine is an open-loop engine during the game and it is vulnerable on the accumulation of balance disturbance. In order to compensate these disturbances, we are working on how to obtain a feedback from the sensors and estimate the state of the robot. For this purpose, we

log the readings of foot pressure and accelerometer sensors and simulate our walking style with the readings in Matlab, as can be seen in Figure 8.

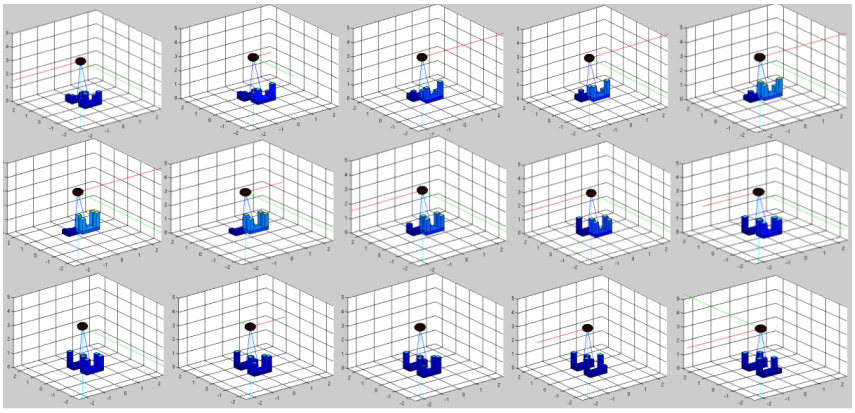

Fig. 8. Changes on the foot sensor reading while walking.

Aside from the implementation inspired from the work of the NimbRo team, we have also developed a CPG-based custom algorithm for bipedal walking. In our design, the main walking motion starts from the hip, specifically the roll joint, which makes the body to swing from one side to the other. In order to keep the feet parallel to the ground while swinging, the ankle roll joint angles should be set to the negative of the value of the corresponding hip roll joint angle. The periodic movement of the hip is obtained by using a sinusoidal signal to be supplied as the hip roll joint angle. In order to realize this movement, the hip roll and ankle roll angles are set according to the following equations.

$$\theta_{hip_{roll}} = A_{hip_{roll}} sin(period)$$
$$\theta_{ankle_{roll}} = -A_{ankle_{roll}} sin(period)$$

This motion is the basis of the entire walking since it passes the projection of the center of mass from one foot to the other periodically, letting the idle foot to move according to the requested motion command.

In order to make the robot perform a stepping motion, the pitch joints on the leg chain should be moved. These joints again take sinusoidal angle values which are consistent with the hip roll angle. The following equations illustrate how the values of these angles are computed.

$$\theta_{hip_{pitch}} = A_{pitch} sin(period) + \theta_{hip_{pitch}^{rest}}$$
$$\theta_{knee_{pitch}} = -2A_{pitch} sin(period) + \theta_{knee_{pitch}^{rest}}$$
$$\theta_{ankle_{pitch}} = A_{pitch} sin(period) + \theta_{ankle_{pitch}^{rest}}$$

The A_{pitch} value determines how big the step is going to be. Obtaining backwards walk does not require much work but just reversing the iteration of the $period$ value, which is defined as $0 < period < 2\pi$.

Similarly, making the robot move laterally is possible by setting the $roll$ angles instead of the $pitch$ angles together with the knee pitch, while turning around is possible by setting the $hipYawPitch$ joint angles properly. The amplitudes $A_{pitch}, A_{roll}, A_{yaw}$ are multiplied with the corresponding motion component, namely $forward, left, turn$ which are normalized in the interval $[-1, 1]$, to manipulate the velocity of the motion. In order to make the robot move omnidirectionally, the sinusoidal signals that are computed individually for each motion component are summed up and the final joint angle values obtained in that way. For instance, it is possible to make the robot walk diagonally in the north-west direction by simply assigning positive values to both the $forward$ and the $left$ components.

7 Planning

The soccer domain is a continuous environment, but the 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 decides the new actions. The main aim of the planner is to sufficiently model the environment and update its status. Second, the planner should provide control inputs according to this model.

7.1 Market Based Approach

For coordination among the teammates and task allocation, we have so far employed a market driven task allocation scheme [9, 10]. In this method, the robots calculate a cost value (their fitness) for each role. The calculated costs are broadcasted through the team and based on a ranking scheme, the robots pick the most appropriate role for their costs. Here, each team member calculates costs for its assigned tasks, including the cost of moving, aligning itself suitably for the task, and the cost of object avoidance, then looks for another team member who can do this task for less cost by opening an auction on that task. If one or more of the robots can do this task with a lower cost, they are assigned to that task, so both the robots and the team increase their profit. The other robots take actions according to their cost functions (each takes the action that is most profitable for itself). Since all robots share their costs, they know which task is appropriate for each one so they do not need to tell others about their decisions and they do not need a leader to assign tasks. If one fails, another would take the task and go on working.

7.2 Dec-POMDP based Planner

We have started developing a DEC-POMDP based planner which uses an extended, scalable version of the approach described in [11]. In this approach the team uses an off-line learned policy during the games. The metrics in [12] form the basis of the rewards in the learning process. This was first tested in October 2009 at the Athens Digital Week during a friendly game against the Kouretes team.

7.3 Hierarchical Planner

The planner architecture we have used in the previous years had some design problems. This section describes a new hierarchical planner architecture addressing these problems.

Requirements

- **Behavior Design:** Our previous Finite State Machine (FSM) based planner module required several C++ classes to implement a given single behavior. Although this seemed like an appropriate requirement at first, we have realized that it is imperative to first design the behavior before implementing it. As expected the common software engineering practices are in effect in behavior design as well. The planning architecture should in some way force the developer to design the behavior to be implemented.
- **Development Speed:** Implementing a few C++ classes was never fast enough. Considering the compiling, loading the code to the robot and rebooting the application cycle, planning development with a strictly compiled language quickly becomes a burden. Especially in the case of developing a planner module, which requires empirical specification of magical numbers for behavior implementations, development and testing cycle speed is of crucial importance.
- **Architecture Concerns:** FSM based architectures have been the initial choice for most planner architectures. These provide a convenient and complete state definition for the agent. The problem with applying FSM based planners in highly dynamic environments, such as robotic soccer, is the sources of uncertainty that are present in the environment. The state determination procedure, which is based on the uncertain perception information, may not always be able to determine the correct state. Handling such a fast paced environment with an FSM based planner requires a lot of work mainly because FSM requires connections between all nodes to be able to cope with all possible choices of a highly dynamic environment.

Proposed Planner Architecture To handle the dynamics of the environment, which may be due to imperfect perception or rapid changes in the environment, a hierarchical planner architecture is proposed. The inspiration comes from the following simple principle "the planning module must come up with a rational decision at any time".

In an FSM based planner, the selected state defines all of the current actions as well as the possible next states. Although levels of FSMs can be defined to handle different possible actions, a much simpler hierarchical architecture is sufficient for coding the required behaviors.

The hierarchical architecture is represented as a tree, where each node of the tree corresponds to a specific behavior. An example is provided in Figure 9. Each node is responsible for deciding on its activation condition and run time behavior. If a node's activation condition is met, then that node's children are also triggered. In this recursive manner a given behavior tree is traversed in depth first order. Making sure the correct node is activated at any given time is the job of the behavior designer.

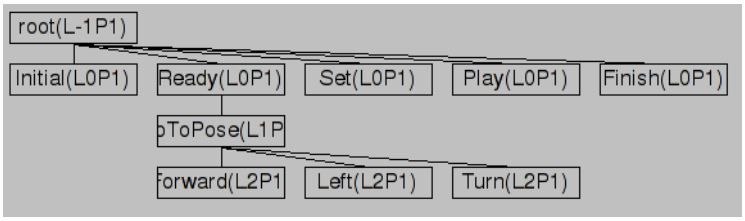

Fig. 9. An example behavior tree.

Implementation The behavior tree is specified using a configuration file. The configuration file of the behavior tree of Figure 9 is given in the Figure 10. This meets the behavior designing requirement since it enforces the behavior designer to first code the global design in a separate file.

```
behaviors/robocup/
Initial
  #DoNothing
Ready
 GoToPose initPose
  Forward
  Left
  Turn
Set
Play
Finish
```

Fig. 10. An example behavior configuration file.

In the configuration file, the developer specifies the location of the corresponding behavior code on the first line. All other lines represent nodes in the configuration tree. A single space before a behavior name means that the node is a child node of a node give specified above. Lines starting with a hash symbol are comments.

The configuration file is parsed by a Python script, which generates the behavior tree at run time and starts triggering the nodes defined by the corresponding Python scripts. If a node code is changed, the tree can be generated again at run time without any need for reboot or compiling, which satisfies the second requirement. Changes to the node code can be performed on the robot's system and the run time reloading can be performed by a signal sent over the network. Furthermore specifying behavior nodes is quite a simple task. Figure 11 presents an example behavior, which is specified with only 10 lines of Python code. Thus rapid testing can be performed with a very fast develop/test cycle.

```
import ppm
class Finish(ppm.PlannerNode):
    name = "Finish"
    def activationCondition(self) :
        if self.mem['robot'].gameState ==
                            self.mem['robot'].gameState.FINISHED:
            return True
        else :
            return False
    def run(self) :
        self.mem['robot'].motionCommand_vision.actionType =
                        self.mem['naolib'].BodyActions.actSitDown
```

Fig. 11. An example behavior node definition.

8 Acknowledgements

This work is supported by Boğaziçi University Research Fund through project 09M105, and TUBITAK Project 106E172.

References

1. Trolltech's Qt Development Framework. http://trolltech.com/products.
2. Ethem Alpaydın. *Machine Learning*. MIT Press, 2004.
3. Greg Welch and Gary Bishop. An introduction to the kalman filter. Technical report, Chapel Hill, NC, USA, 1995.
4. K. Kaplan, 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*, 4020:420–427, 2006.
5. M. Montemerlo. FastSLAM: A factored solution to the simultaneous localization and mapping problem with unknown data association. In *CMU Robotics Institute*, 2003.
6. S. Thrun, W. Burgard, and D. Fox. *Probabilistic Robotics*. MIT Press, Cambridge, MA, 2005.
7. C. Pinto and M. Golubitsky. Central pattern generators for bipedal locomotion. *J Math Biol*, 2006.
8. Sven Behnke. Online trajectory generation for omnidirectional biped walking. 2006.
9. H. Köse, 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 Antalya, Turkey, Proceedings LNCS 2869*, pages 529–536, 2003.
10. H. Köse, K. Kaplan, Ç. Meriçli, U. Tatlıdede, and H. L. Akın. Market-driven multi-agent collaboration in robot soccer domain. *Cutting Edge Robotics*, pages 407–416, 2005.
11. Baris Eker and H. Levent Akin. Using evolution strategies to solve DEC-POMDP problems. *SOFT COMPUTING*, 14(1):35–47, JAN 2010.
12. Cetin Mericli and H. Levent Akin. A Layered Metric Definition and Evaluation Framework for Multirobot Systems. In Iocchi, L and Matsubara, H and Weitzenfeld, A and Zhou, C, editor, *ROBOCUP 2008: ROBOT SOCCER WORLD CUP XII*, volume 5399 of *Lecture Notes in Computer Science*, pages 568–579, 2009.