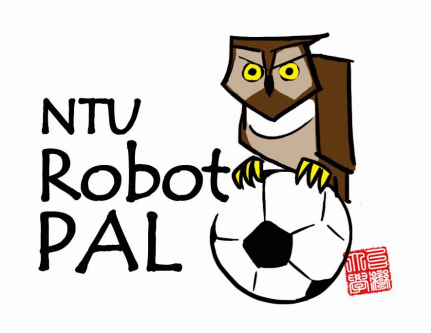
NTU RoboPAL

Team Report and Code Release 2011



Chieh-Chih Wang, Shao-Chen Wang, Chun-Hua Chang,
Bo-Wei Wang, Hsin-Cheng Chao and Chih-Chung Chou[1]

Department of Computer Science and Information Engineering
Graduate Institute of Networking and Multimedia
National Taiwan University, Taipei, Taiwan
bobwang@ntu.edu.tw,{alan,karate362}@pal.csie.ntu.edu.tw
http://www.csie.ntu.edu.tw/~bobwang/RoboCupSPL/

Revision: 2012-01-13

[1] Chih-Chung Chou is a new member of the team to participate in RoboCup 2012. He significantly contributed to this team report.

1 Introduction

This is the team research report describing the work of Team NTU RoboPAL for RoboCup 2011 SPL. Team NTU RoboPAL has been participated in the RoboCup Standard Platform League (SPL) competitions since 2009. We were in the top 8 teams in 2009, in the top 16 teams in 2010, and won the 3rd place in 2011.

The members of Team NTU RoboPAL. From left to right: Chieh-Chih Wang, Bo-Wei Wang, Shao-Chen Wang, Chun-Hua Chang and Hsin-Cheng Chao.

Our RoboCup software system used in RoboCup 2011 consists of three parts: Perception, Motion, and Behavior. As our system was developed based on the B-Human code release 2010 [1], only the newly developed modules and functions are described in this team report. The codes of these modules and functions are available in the NTU RoboPAL code release[2].

2 Perception

Localization and tracking are two key capabilities in the RoboCup competition. Knowing self-location in the field is essential for a robot to perform reasonable behaviors such as kicking the ball toward the goal. Accurately tracking the opponents and the ball is the basis for the robot to exhibit intelligent behaviors such as high-level planning and cooperative strategies.

However, in the RoboCup SPL games the performance of self-localization can degrade when the robot cannot collect adequate information from onboard sensors Fig. 1 and Fig. 2 illustrate these challenging scenarios. Fig. 1 demonstrates that only a couple of map features can be observed by a robot near the field boundary,

[2] http://www.csie.ntu.edu.tw/~bobwang/RoboCupSPL/NTURoboPAL_CodeRelease2011.rar

and Fig. 2 shows that a robot is asked to look at the ball for preparing a kick but unable to observe sufficient map features for localization.

Fig. 1 The Boundary Case: the robot indicated by the red rectangle detects only one field line and two corners. The bottom images are a sequence of images captured during a right to left head motion in 2 seconds.

Fig. 2 The Ball Gazing Case: the robot indicated by the red rectangle only detects the ball without any map feature. The bottom images are a sequence of images captured during the task.

Motivated by the self-localization challenges and stimulated by the importance of both localization and tracking in the RoboCup games, we designed and implemented a Cooperative Localization And Tracking (CLAT) module [3] based on our simultaneous localization, mapping and moving object tracking framework [2]. In the CLAT module, localization and tracking are performed simultaneously to get the assistance from teammate robots and exploit the information of moving objects. When the robot observes the moving object, the moving object information can be utilized for localization if his teammates can also observe that moving object.

In the next section, the overview of our perception system is firstly demonstrated and the details of each perception module are described from Sec. 2.2 to Sec. 2.5.

2.1 Perception System Overview

Fig. 3 shows the overview diagram of our perception system. The core of our perception system is the CLAT module [3], which is an EKF-based algorithm that exploits the assistance of the teammate robots and the information of moving objects by augmenting the states of all the teammate robots and the moving objects into one state vector. Localization and tracking are performed simultaneously. As the correlations among all the robots and the moving objects are maintained through the covariance matrix, localization and tracking are mutually beneficial.

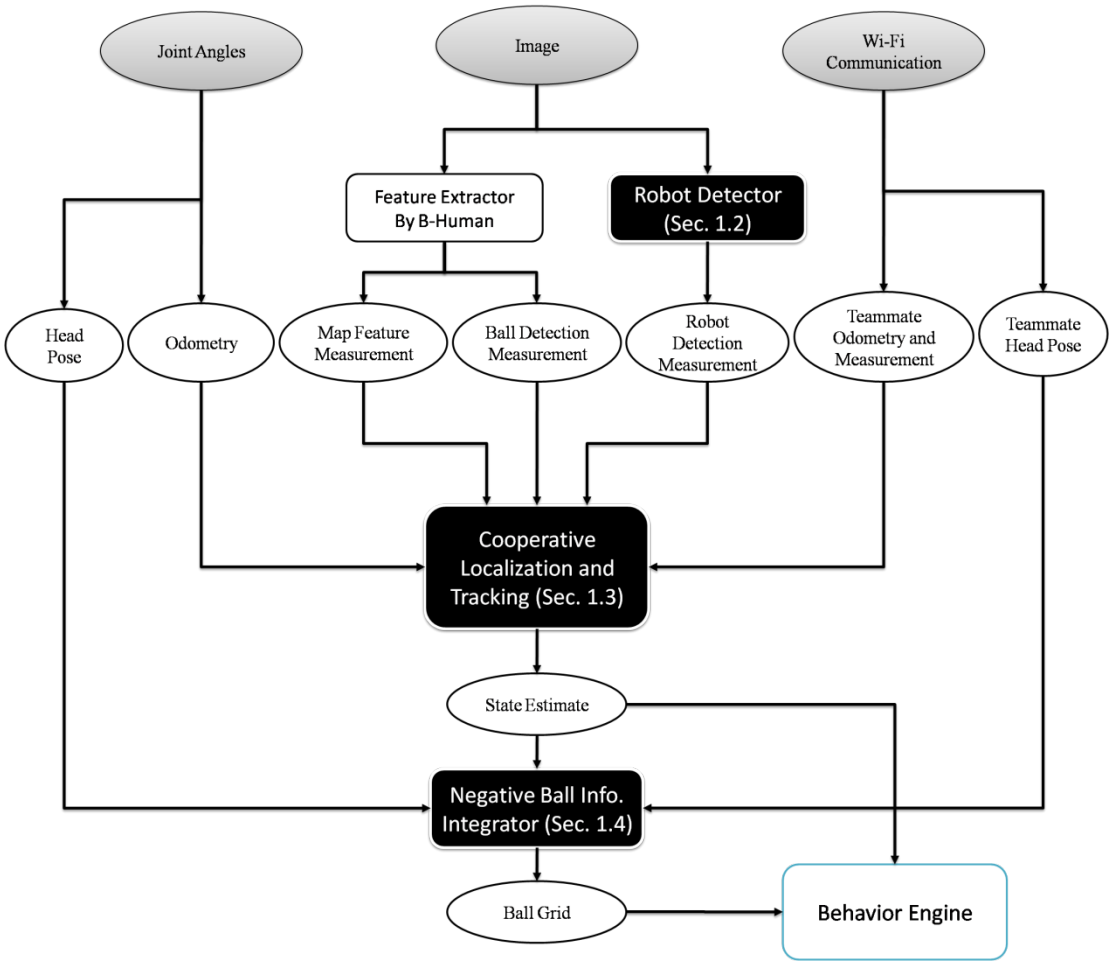


Fig. 3 Overview of our cooperative and perception system

Compared with traditional self-localization approaches, CLAT additionally utilizes relative measurements among robots and moving objects to achieve better estimation performance. Therefore, we have designed a Nao robot detection module to retrieve information between the robots on the field. The details of the detection algorithm and its performance evaluation are addressed in Sec. 2.2.

The goal of CLAT is to achieve accurate estimation by integrating information from all the teammate robots, so during the game, each teammate robot constantly shares its motion command, self-localization measurements, and the ball and robot detection measurements to the others through Wi-Fi. After the robot receiving the motion and measurements from the teammates, the information is integrated to improve the state estimates. The details of the CLAT algorithm are described in Sec. 2.3.

The position of the ball is undoubtedly the most critical information in soccer games. Although the CLAT algorithm can estimate the ball position once the ball is observed by at least one of the teammate robots, there are still cases that the ball is not detected by any teammate robot. Therefore, based on the CLAT algorithm, we additionally design a module to incorporate the negative observation of the ball. More specifically, when the ball is not observed, we also improve the ball position estimation by eliminating the impossible ball positions. The negative ball information integration algorithm is detailed in Sec. 2.4.

In addition, we also designed the vision-based short-range obstacle detection module, which is currently not yet connected to the other perception modules but directly reports information to the behavior module. The main purpose of this module is to compensate the instability and inaccurate obstacle detection by sonar. The details are described in Sec. 2.5

2.2 Nao Robot Detection

Th CLAT module requires relative measurements between the robot itself and the teammates and the moving objects. In the RoboCup scenario, the Nao robot detection is the basis of the robot-to-robot and robot-to-moving-object measurements.

Our Nao robot detection module works as follows:

1. The part of image above the field border (depicted as the black dashed line in Fig. 4) is ignored because the colors in the background above the field border are basically unpredictable and cannot be reliably used to infer the existence of the robot.
2. Based on the color segmentation module in [1], samples are drawn from the non-line white segments (depicted as the yellow dots in Fig. 4) and are then clustered by pixel distance in the image space.
3. The extracted clusters are classified as Nao robots if all of the following three criterions are satisfied:
   i. The number of segments in the cluster should be larger than 3.
   ii. The width-to-height ratio, where the height is defined in the longest direction of the cluster, should be larger than 0.2. This threshold is important to distinguish the robot from the field lines.
   iii. The highest point of the cluster should be close to the border line within 10 pixels as the observed robot should intersect with the field border in the camera view if both of the observing and observed robots are standing in the field.

Fig. 4 illustrates an example detection result from our Nao robot detector. After the robot has been detected, the lowest-center point of the robot in the image is determined in order to compute the relative range and bearing of the robot in 2D based on the assumption that all the map features and the robots are on the same ground plane. This calculation procedure is illustrated in Fig. 5.

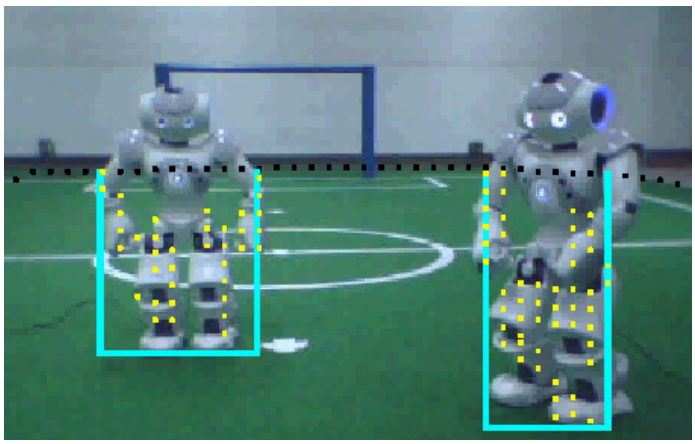


Fig. 4 The robot detection example. The robot detection results are indicated by cyan lines. The black dashed line indicates the field border and yellow dots are pixels sampled from non-line white segments.

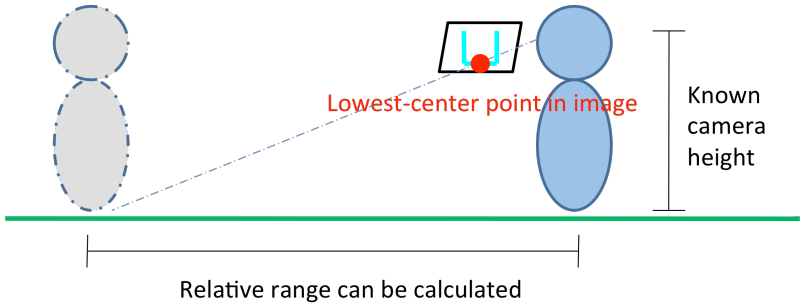


Fig. 5 Illustration for the procedure of computing the relative range given the lowest-center point of an object in the image

[Code Release]

Our Nao robot detection algorithm is implemented in the module named *RobotPerceptorNtu* in the files *Modules/Perception/RobotPercepterNtu.h* and *Modules/Perception/RobotPercepterNtu.cpp*. The module provides the representation *RobotPerceptNtu* defined in the file *Representations/Perception/RobotPerceptNtu.h.* Note that *module* and *representation* are programming structures following the definitions in the B-Human system architecture (Sec. 3.3 of [1]).

The performance of our Nao robot detection module is summarized in Table 1 and Fig. 6. The recall rate is higher when the target is in the front view than it is in the side view or the back view. As the proposed detection module is simply based on the detected white region, the detection performance degrades in either the side view or the back view compared to that in the front view. The samples of the observations from different views are shown in Fig. 6. The recall rate is above 0.6 within 2 meter in average and it decreases as the target gets farther. The range accuracy is around 30 centimeters when the target is closer than 2 meters but grows to around 60 centimeters as the target gets farther. The bearing accuracy is around 7.5 degrees. The overall detection precision is 0.92 where the false positives are mainly arising from the misclassifications of the field lines.

	100cm	150cm	200cm	250cm	300cm
σ_{range}(cm)	27.32	31.77	37.91	41.92	65.82
σ_{angle}(degree)	7.95	7.13	6.80	7.28	8.17

Table 1 Standard deviation of our robot detector in different ranges

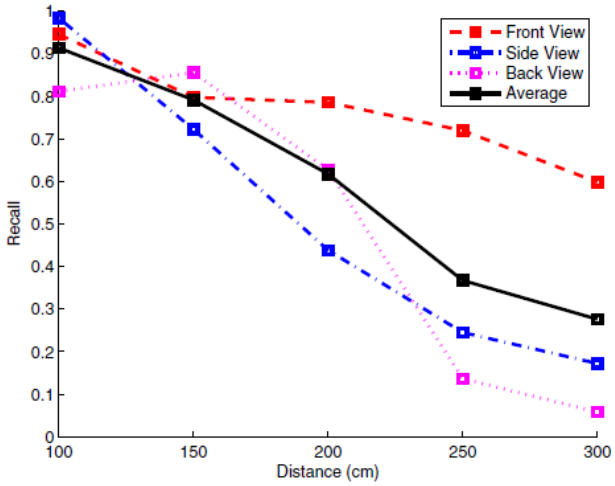


Fig. 6 Recall rates of our robot detector in different robot views and at different distances

Fig. 7 Photos of different views of the Nao robot

2.3 Cooperative Localization and Tracking (CLAT)

Our cooperative localization and tracking module is an EKF-based algorithm with the augmented state vector consisting of all the teammate robots and the moving objects. Aiming at providing accurate estimates, CLAT plays as an information fusion center that aggregates the motion commands and three types of measurements, including the robot-to-map measurements, the robot-to-robot measurements, and the robot-to-moving-object measurements, from all the teammate robots.

The main components of CLAT are described in the following:

I. Augmented State Vector

In order to simultaneously estimate the states of both the robots and the moving objects in one coherent framework, we augment them all into the state vector:

$$X_t = \left[\begin{array}{cccccc} (R_t^1)^T & \dots & (R_t^N)^T & (O_t^1)^T & \dots & (O_t^M)^T \end{array} \right]^T$$

where t denotes the time index and R and O represent the teammate and the moving object respectively. The main difference between R and O is whether the motion control information is known. In the RoboCup scenarios, R is the teammate robot and O can be the opponent robot or the ball.

II. The Motion Model

For teammate robots, as the motion commands and the feed backs from the joint encoders are known, the odometry motion model is used. For moving objects, as the odometry information is not available, the constant velocity (CV) model is applied for predicting the motion of the moving objects.

With the assumption that the motion of robots and tracked objects are independent, the propagation matrix of the whole system can be written as:

$$G_t = diag(G^1_{R_t}, \ldots, G^N_{R_t}, G^1_{O_t}, \ldots, G^M_{O_t})$$

where G_R and G_O are the Jacobian matrices of the motion models of the teammate robots and the moving objects, respectively. The mean and covariance of the state after the prediction stage can be calculated through the standard EKF procedure.

III. Data Association

Before updating with the incoming measurements, data association must be determined first. The purpose of data association is to establish the correspondences between the incoming detection measurements and the tracked objects in the state vector.. Here, we apply the maximum likelihood data association algorithm with a threshold gating on the Mahalanobis distance between the incoming measurement and the expected measurement. Then the state is updated according to the three sensor models defined in the following subsection.

IV. The Sensor Model

In the case that the i^{th} robot detects a moving object and associates it with the j^{th} moving object, the sensor model for robot-to-moving-object measurements is computed as:

$$z^{O_j}_{R_i} = h_{RO}(R^i_t, O^j_t) + \Sigma_{RO_t}$$

where $h_{RO}(\cdot)$ is the function that transforms the expected j^{th} moving object position in the global coordinate system to the local coordinate system of the i^{th} robot. The Jacobian matrix of this function is

$$\frac{\partial h_{RO}(\bar{R}_t^i, \bar{O}_t^j)}{\partial \bar{X}_t}$$
$$= \begin{bmatrix} 0 & \dots & 0 & H_{RO_t}^i & 0 & \dots & 0 & H_{RO_t}^j & 0 & \dots & 0 \end{bmatrix}$$

with

$$H_{RO_t}^i = \frac{\partial h(\bar{R}_t^i, \bar{O}_t^j)}{\partial \bar{R}_t^i}$$

and

$$H_{RO_t}^j = \frac{\partial h(\bar{R}_t^i, \bar{O}_t^j)}{\partial \bar{O}_t^j}$$

where $H_{RO_t}^i$ is the i^{th} and $H_{RO_t}^j$ is the $(M + j)^{th}$ element of the matrix, M is the number of teammate robots involved in the state vector. For the other two types of measurements, the measurement function and the Jacobian matrix can also be defined similarly. Accordingly, the state vector and the covariance matrix can be updated with new retrieved measurements following the standard EKF procedure.

V. Track Management

It is also necessary to infer the existences of the moving objects because the tracks of moving objects should be initialized when new ones have been detected and should be pruned when they have not been observed for more than a few seconds. In our work, a binary Bayes filter is applied here to infer the existence of the tracked moving objects, which is a counter-based approach to accumulate the probability of the existence of the tracked moving objects. The existence probability of a tracked object increases as more measurements have been associated with it and vice versa, where the increment is an experimental determined parameter.

The example results of our CLAT algorithm is shown in Fig. 8, which demonstrates that our CLAT algorithm is capable of simultaneously estimating the poses of the teammate robots and the opponent robots. The magenta crosses and ellipses show the pose estimates and the corresponding uncertainties of the

teammate robots, the cyan ones are the position estimates and uncertainties of the opponent robots, the red lines are the relative robot detection measurements, and the black crosses are the ground truth robot positions. This figure shows a general 3-by-3 case and our CLAT algorithm successfully estimates the states of all the robots on the field. Fig. 9 shows the case that the striker, T2, that CLAT improves the estimate by incorporating the moving object information.

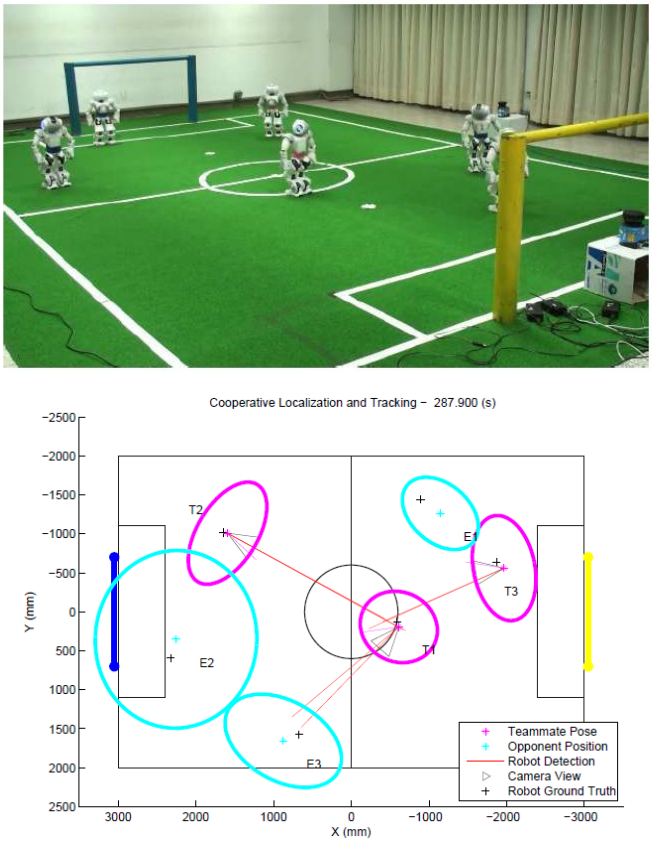
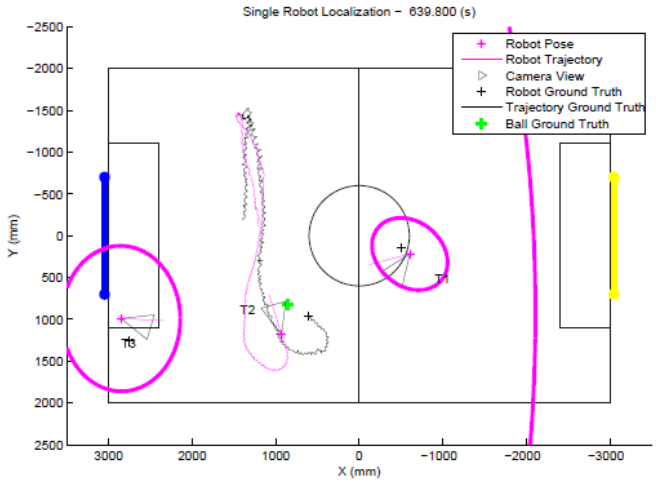


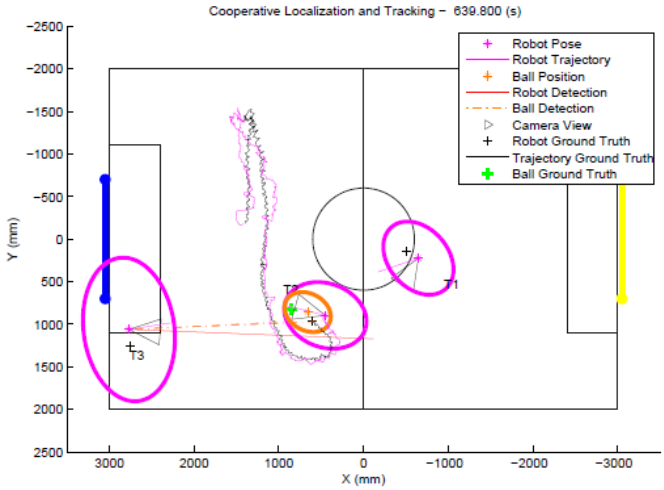
Fig. 8 An example result of CLAT, where three teammate robots and three opponent robots are correctly estimated.

[Code Release]

The CLAT algorithms are implemented in the *StateEstimator* module in the files *Modules/Modeling/StateEstimator.h* and *Modules/Modeling/StateEstimator.cpp*. The state estimator provides the representation *StateEstimate* which contains the current belief of the state estimates of the robots and the moving objects. *StateEstimate* is defined in the file, *Representations/Modeling/StateEstimate.h*.



(a) EKF-based Self-Localization



(b) Cooperative Localization and Tracking

Fig. 9 Comparison for trajectories estimated by self-localization and cooperative localization and tracking

As the particle filter (PF) based algorithm designed by B-Human [1] has already fused the robot-to-map measurements into a self-pose estimate, we presently directly integrate this self-pose estimate from PF as our self-localization measurement in CLAT. We found that PF can naturally adopt multiple hypotheses and makes the estimates more robust, especially for the robot falling and penalization cases.

For the moving objects, we only added the ball into the state vector as the moving object in the competition version. The reasons are: (1) adding the opponent robots into the state vector could make the algorithm suffer from the potential failure when data association accidently makes wrong associations between the teammate

robot and the opponent robot, (2) in RoboCup 2011, we have not yet designed the game strategy that considers the opponent positions, so the attraction for estimating the opponent positions degrades, and (3) for exploiting the advantages of improving localization using moving object tracking, putting the ball into the state vector is sufficient as most of the cases that require moving object tracking for improving localization happen when the striker is gazing and approaching the ball.

Aiming at developing more intelligent strategies that consider all the robots on the field, we are now developing a multiple hypothesis tracking (MHT) based version of the CLAT algorithm to enhance the robustness in data association by explicitly modeling the data association uncertainty for the RobCup 2012 competitions.

2.4 Negative Ball Information Integration

The main purpose of the negative ball information integration module is to improve the ball location estimation not only when the ball is observed but also when the ball is *not* observed. The idea is that in addition to increase the ball existence probability of a place when the ball is observed, the probability decreases when the ball is not observed at that place. Fig. 10 illustrates the approach.

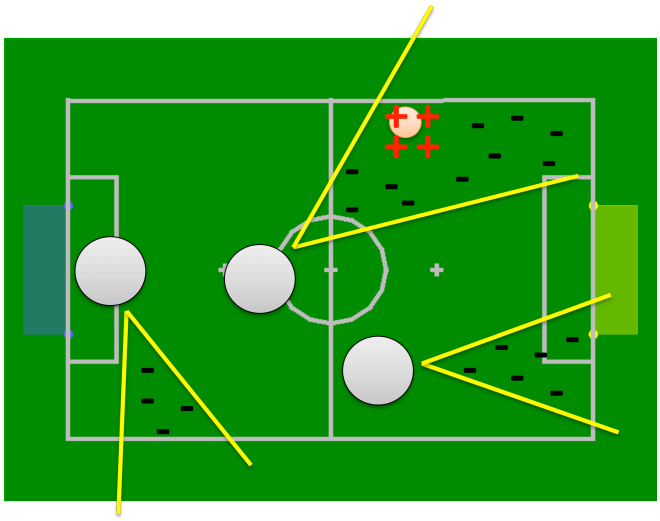


Fig. 10 Illustration for the negative ball information integration

For this purpose, an additional grid data structure is maintained on the top of CLAT, and the robot head poses are also shared with the teammates in order to infer the visible area of the teammates. The pseudo code of the proposed

algorithm is summarized below:

Algorithm Negative Ball Information Integration

Let p(X) be the grid structure maintaining the ball existence probability at each location

```
for each grid x
    for each teammate robot R
        if ball is detected by R at x
            increase p(x)
        else if x is visible by R
            decrease p(x)
        end
    end
end
```

[Code Release]
The related functions and data structures are implemented and added in the files *Modules/BehaviorControl/BH2010StableBehaviorControl/Symbols/BH2010StableBallSymbols.h* and *Modules/BehaviorControl/BH2010StableBehaviorControl/Symbols/BH2010StableBallSymbols.cpp.*

Though this module can be designed and implemented straightforwardly, this module serves as the basis to significantly improve the ball searching efficiency when the ball is presently not visible by any of the teammate robots according to our experiences at RoboCup 2011.

2.5 Vision-based Short-Range Obstacle Detection

The necessity of the vision-based short-range obstacle detection stemmed from the fact that the sonar readings are unstable and inaccurate for obstacle detection. The sonar-based obstacle detector has to deal with a large amount of false positives and false negatives due to the unstableness of the sonar readings. It is also difficult to determine the precise position of the short-range obstacle as the obstacle near in front usually makes both of the left and right sonar readings short. Without more

accurate obstacle location estimates, obstacle avoidance would be difficult.

As pushing other robots violates the rule and makes the robot penalized, we do not either want the robot bumping into another robot or becoming too conservative to miss the chance to approach the ball. Therefore, the vision-based short-range obstacle detection module is designed to compensate the flaw of the sonar sensor and to provide more precise estimates of the short-range obstacle in front of the robot. Our vision-based short-range obstacle detector is based on the percentage of the green pixels in the image. Note that it is designed only for short-range obstacles and is enabled only when the robot looks down in front. The algorithm works as follows:

a. When the ball is not in view, the image is firstly equally split into the left and right parts, and then the green pixel percentages are respectively computed. If one of the percentages is below a threshold, it is deemed that there is an obstacle in front, and both of the percentages will be exported to the behavior module in order to determine which side to move the robot to avoid the detected obstacle.
b. When the ball is in view, in addition to the above two ratio values, we also compute the ratios for the image part below the ball in order to determine whether and how the robot can approach the ball even when there are other obstacles in view.

Fig. 11 shows an example in which the algorithm reports there is an obstacle in front, but the robot can still approach the ball because it is free between the robot and the ball. Therefore, the robot will not conservatively stop and will keep fighting against the opponent for the ball.

[Code Release]
The related functions for are implemented and added in the files *Modules/BehaviorControl/BH2010StableBehaviorControl/Symbols/BH2010StableObstacleSymbols.h* and *Modules/BehaviorControl/BH2010StableBehaviorControl/Symbols/BH2010StableObstacleSymbols.cpp.*

To summarize our perception system, the robot detector provides the measurements containing the spatial relations between the robots on the field. The CLAT module integrates the motion commands and three types of measurements, including the robot-to-map measurements, the robot-to-robot measurements, and the robot-to-moving-object measurements, from all the teammate robots, to estimate the states of the teammate robots and the moving objects. For further improving the ball estimation, the negative ball information is incorporated. Additionally, a vision-based short range obstacle detector is developed to compensate the unstable and inaccurate sonar readings.

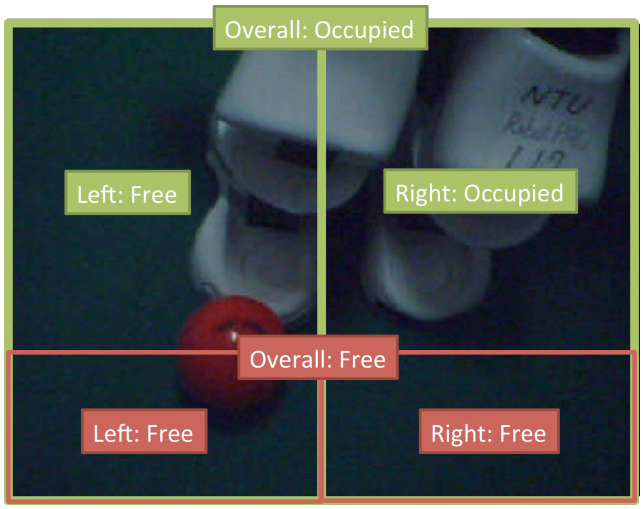


Fig. 11 Image from the Nao robot's view when looking down, for illustrating the vision-based short-range obstacle detection

3 Motion

Prompt kicking and blocking motions contributed significantly to our games in RoboCup 2011. In this section, several important motions we designed are introduced including three different kicking motions for the striker and a defensive diving motion for the keeper. As our walking engine is based on the B-Human Code Release, the readers are referred to Sec. 5.2.3 in [1] for more information.

3.1 Special Actions with Parameter Passing

Based on the B-Human framework [1], it is feasible to command the robot to execute the hand-coded fixed motions, which are called *special actions* (Sec. 5.2.4 of [1]). To design a special action, we manually make the robot act and record all of its joint angle values at the key frames in a *mof* file. The process of executing a special action is depicted in Fig. 12. When a special action is called from the behavior state machine, the ***SpecialAction*** module will receive the call, read the corresponding *mof* file, and send the joint angle values recorded in the *mof* file to the servo motors on the Nao robot.

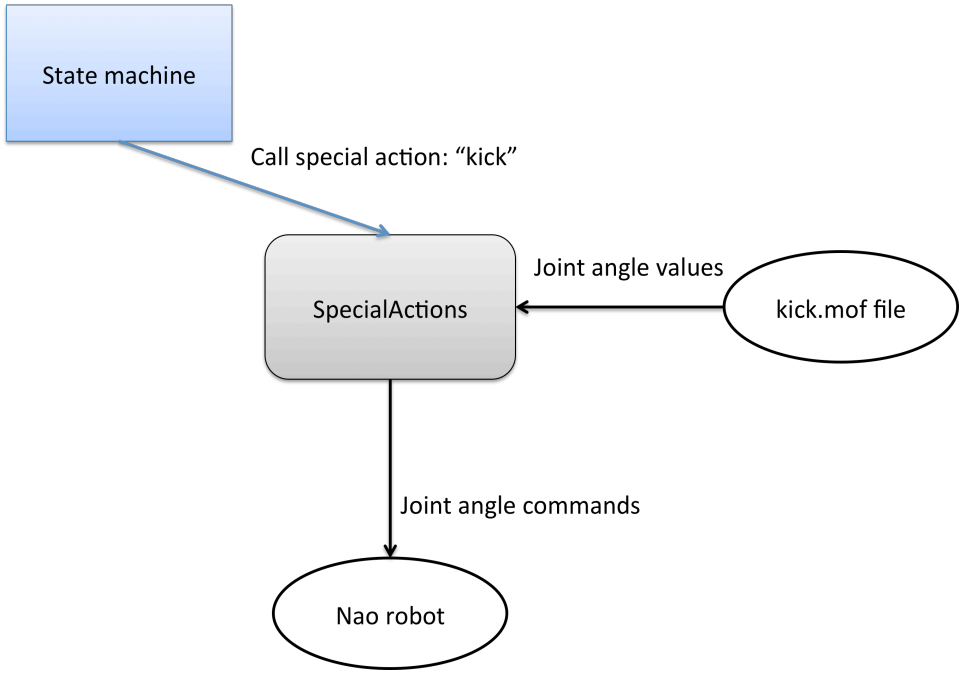


Fig. 12: Original process of executing a special action

However, it is not allowed to pass parameters to dynamically change the joint values of the special actions, which could be very inconvenient for specific situations. For example, we have to write a number of *mof* files for the robot to kick the ball to different directions, one for a certain angle. Therefore, we modify the working flow of the special action slightly, and by letting some values in the *mof* file become variables, we can design only one *mof* file and perform different motions by passing different parameters.

There are three steps for achieving parameter parsing in executing special actions and the modified procedure is illustrated in Fig. 13.

1. Define new symbols in the *mof* files. Therefore, instead of coding constant joint values in the *mof* file, we can let some joints be the variables that will be determined online.
2. During the game, determine the parameters online by the behavior engine, and pass the values to the *SpecialActions* module.
3. Accordingly, the *SpecialActions* module can replace the values of the variables in the *mof* file by the online-determined parameters from the behavior engine, and thus achieves the special action with parameter passing.

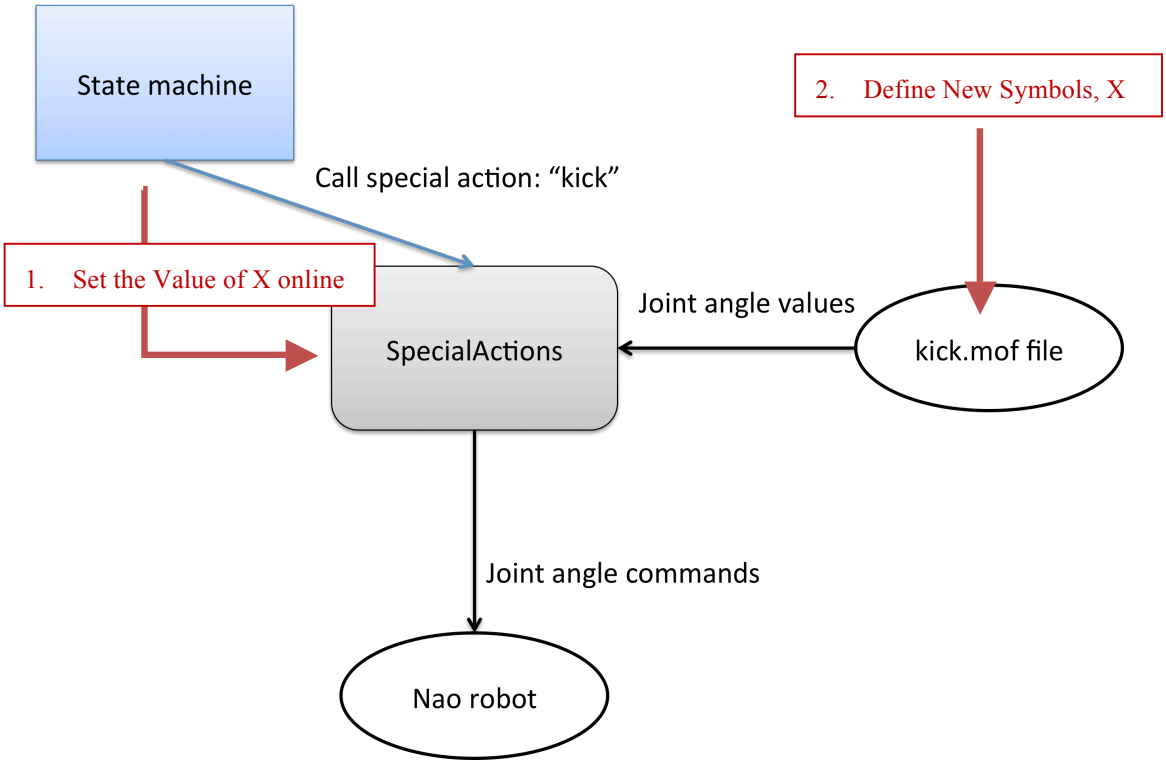


Fig. 13: Process of executing a special action with online-determined parameters

3.2 Wide-Angle Kick

Based on the functionality of parameter passing to the special action, we can develop the motion enabling the robot to kick toward different angles without spending a long time to adjust its body orientation, and we call this motion wide-angle kick. The most important insight for this wide-angle kick motion is that the kicking direction can be changed by just setting different values on the hip, knee and ankle joints. Therefore, the joint angle data of kicking toward different angles were collected offline, and the proper joint angle values can be computed online using linear interpolation.

Using the proposed wide-angle kick action, the striker can kick the ball to a direction with a range of -50 to 50 degrees. Fig. 14 shows the snapshots of a wide-angle kick motion toward the direction of 45 degrees. The main moving directions of the parts of the robot are depicted by the arrows.

[Code Release]
This special action is coded in the *rotkick.mof* file.

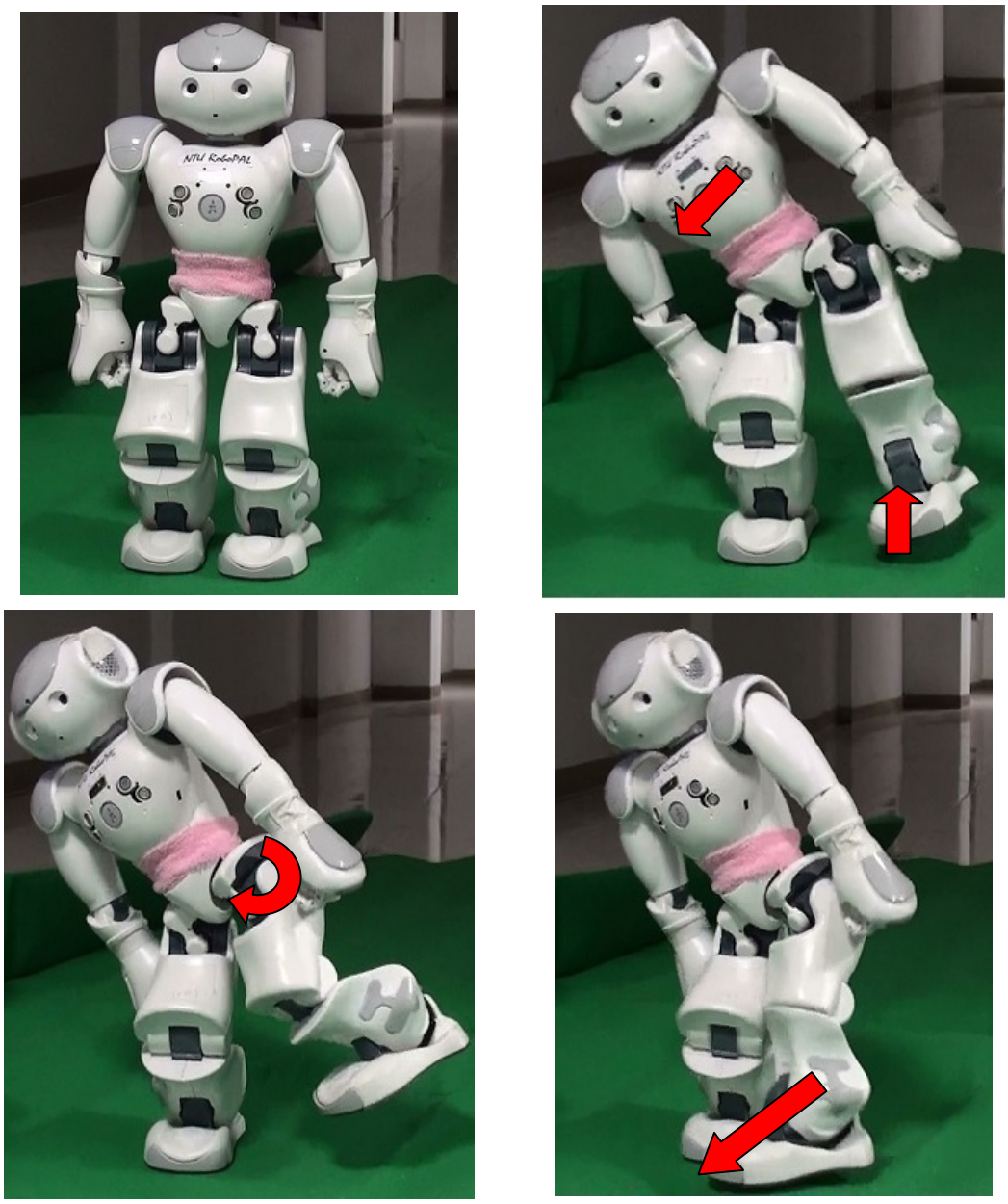


Fig. 14: Wide kick action, red arrows depict the moving direction of parts and joints.

3.3 Fast-Kick

Although the wide-angle-kick could be a powerful action, it takes about 3 seconds to make the shot. In some situations, the robot should just make the kicking as quickly as possible. For examples, when the striker and the ball are very close to the opponent's goal, the striker only has to touch the ball lightly and quickly. Otherwise, the opponent keeper may come and block the ball. Accordingly, we designed this fast-kick motion based on moving the robot's joints minimally to touch the ball lightly and quickly. With this fast-kick action, the robot can quickly move the ball and

be ready for the next kicking. Fig. 15 shows the snapshots of the fast-kick action. The fast-kick action can be completed in 1 second which is three times faster than the wide-angle kick action.

[Code Release]
This special action is coded in the *lightkick.mof* file.

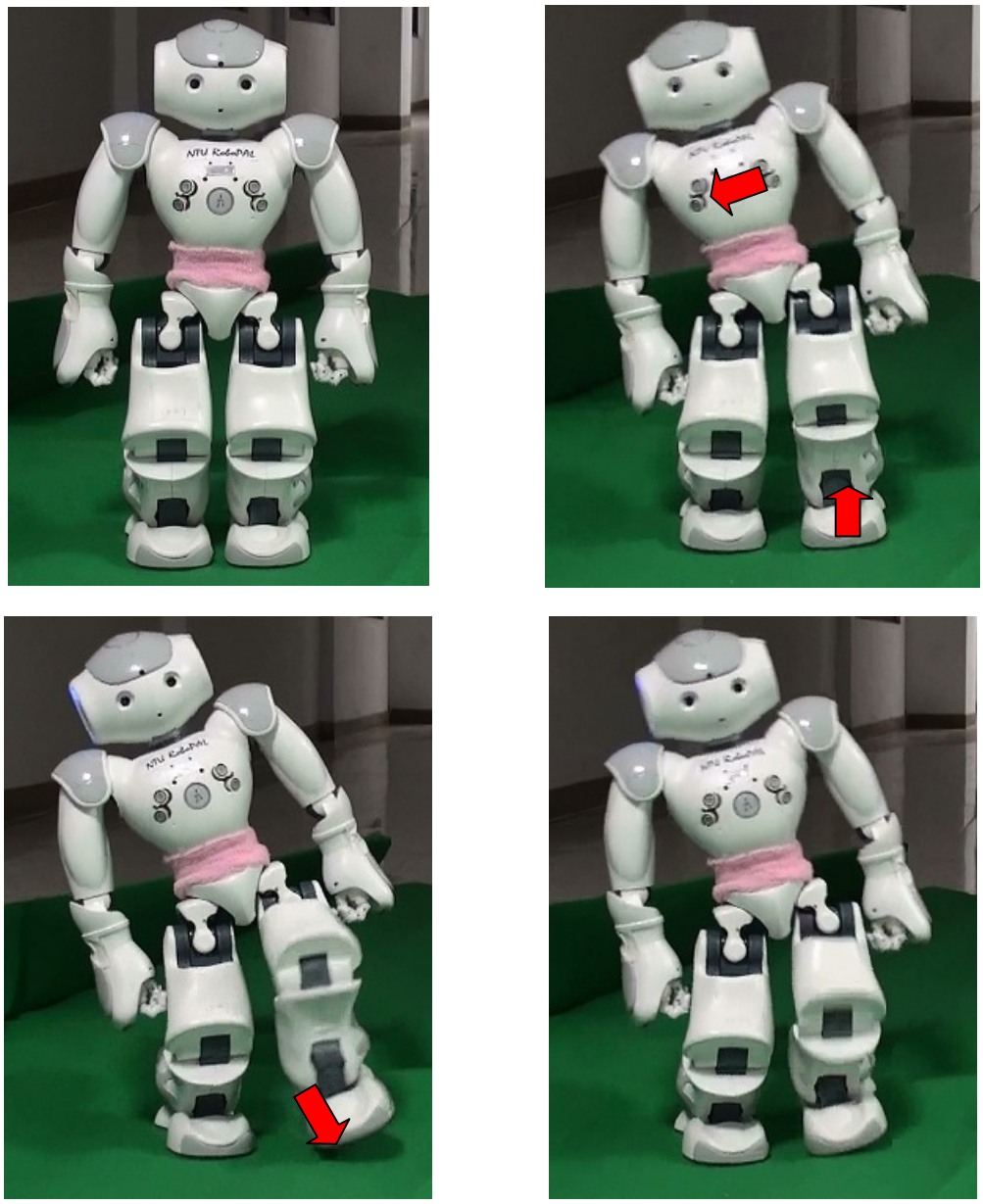


Fig. 15: Fast kick action, red arrows depict the moving direction of parts and joints.

3.4 Side-Kick

The goal of the side-kick action is to directly kick the ball to left or right without asking the robot to circle around the ball. It is especially important for situations that there are opponents blocking in the way from the ball to the opponent's goal. Our side-kick action is almost as fast as the fast-kick action. Fig. 16 shows the snapshots of the side-kick action.

[Code Release]
This special action is coded in the *sidekick.mof* file.

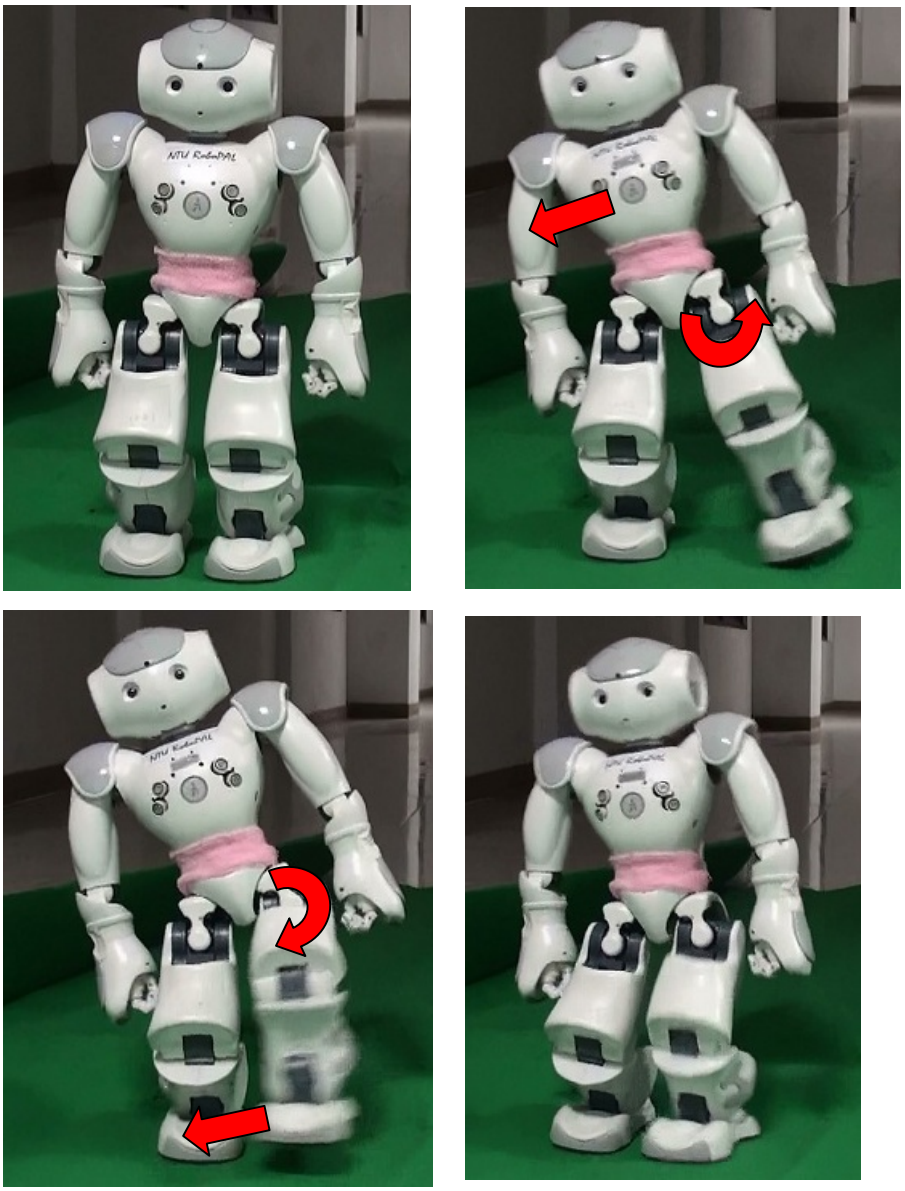


Fig. 16: Side kick action, red arrows depict the moving direction of parts and joints.

3.5 Diving

Diving is a special action for the keeper. When the keeper finds that the approaching ball is too fast to be blocked by walking, the keeper will actively lie down for fast blocking. To minimize the harm brought by falling, we let the robot relax the hardness of all the joints before it bumps into the ground. In practice, this action has saved us from many strong shots.

[Code Release]

This special action is written in the *dive.mof* file.

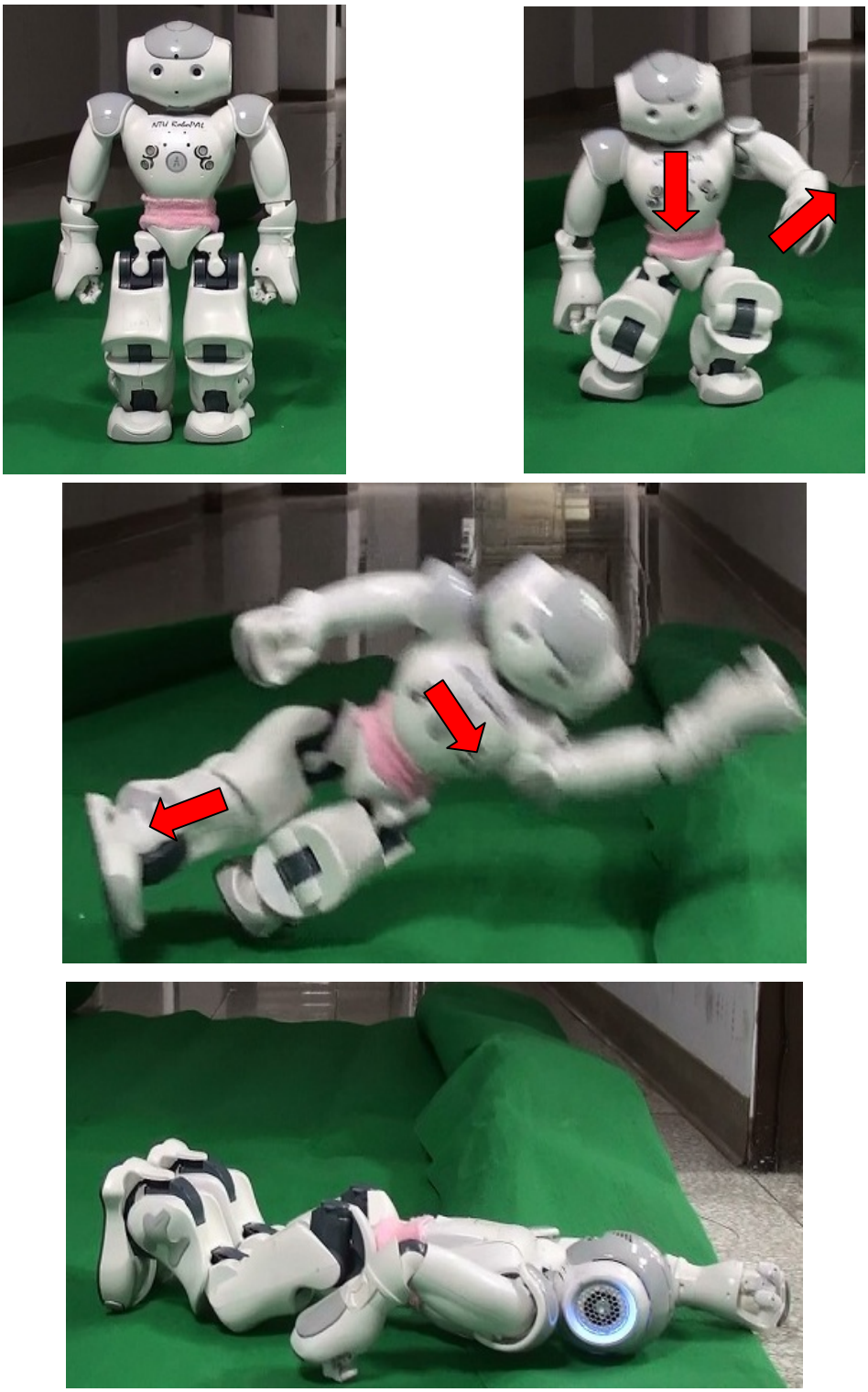


Fig. 17: Diving action, red arrows depict the moving direction of parts and joints.

4 Behavior

We have designed four different roles: the striker, the offensive supporter, the defensive supporter, and the keeper. Each robot is assigned as one of the four roles at the beginning of the game. As the game situation changes, the role assignment is changed accordingly. In this section, we will first overview the tasks for different roles, and then introduce the role switching conditions and the behaviors for the four different roles.

4.1 Task Overview of the Different Roles

For not making the robots hinder each other, we only allow one robot to be the striker and to kick the ball. The task of the striker is to reach the ball and make the kick as efficiently as possible. The other two robots should be supporters. Two types of supporters are designed, offensive and defensive. The main task of the offensive supporter is to reach the position where the ball is likely to be after the striker kicks, and therefore it can become the next striker quickly. The keeper and the defensive supporter focus on blocking the ball kicked from the opponents. The keeper stays in the penalty area for most of the time, and the defensive supporter stands on the line between the ball and our own goal. An example of role assignment is shown in Fig. 18.

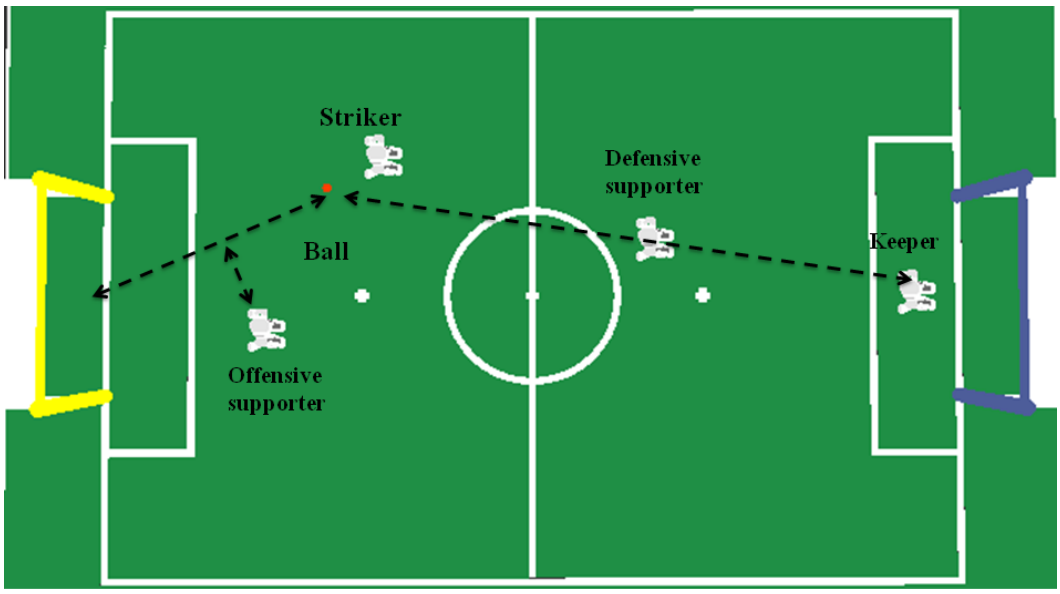


Fig. 18 An example of our role assignment

4.2 Role Switch

During the game, the striker, the offensive supporter and the defensive supporter change their roles according to their current poses and the ball location. The robot which can reach the ball in the shortest time will be the striker. Other than the striker, the robot which is nearer to the opponent's goal will become the offensive supporter and the other one will become the defensive supporter. If the distances between the ball and two robots are approximately equal, the two robots may interchange their roles frequently and then act unstably. A counter to record the time passed after the previous role changing is added. The role switch cannot happen again in a too short period of time in order to make the role assignments stable.

[Code Release]
The role-switching related functions are implemented and added in the files *Modules/BehaviorControl/BH2010StableBehaviorControl/Symbols/BH2010StableBallSymbols.h*, *Modules/BehaviorControl/BH2010StableBehaviorControl/Symbols/BH2010StableBallSymbols.cpp, and Modules/BehaviorControl/BH2010StableBehaviorControl/Options/Soccer/body_control.xabsl.*

4.3 Striker

The striker is the one with the most complicated behavior design in our system. We use two primary modules to hierarchically design its behavior.

4.3.1 Top level behavior

Fig. 19 shows the first module: ***playing_striker***. It is used to navigate the striker to the ball. In this module, if the striker hasn't seen the ball for a long time, it will enter the ***search*** state and start to search the ball using a simple behavior in which the striker stands at the same place and turns around until it finds the ball. If the ball is seen, the striker will go to the ***ball_found*** state and start to move toward the ball. The striker will firstly turn to the direction of the ball and then walk to it with the full speed. During walking, the striker still adjusts its orientation when it finds that the ball direction changes.

If there are obstacles, e.g. opponent robots, on the path, the striker will enter

the ***far_avoid*** state to do obstacle avoidance. For reaching the ball earlier, the striker will not stop or move back. It avoids obstacles by moving laterally or moving forward with a rotational speed depending on the direction and distance of the obstacle in front. Once the ball is near enough, the striker will change to the ***striker_kick*** module and prepare to kick.

[Code Release]
The top level behavior of our striker is implemented in the file *Modules/BehaviorControl/BH2010StableBehaviorControl/Options/Soccer/Striker/playing_striker2.xabsl.*

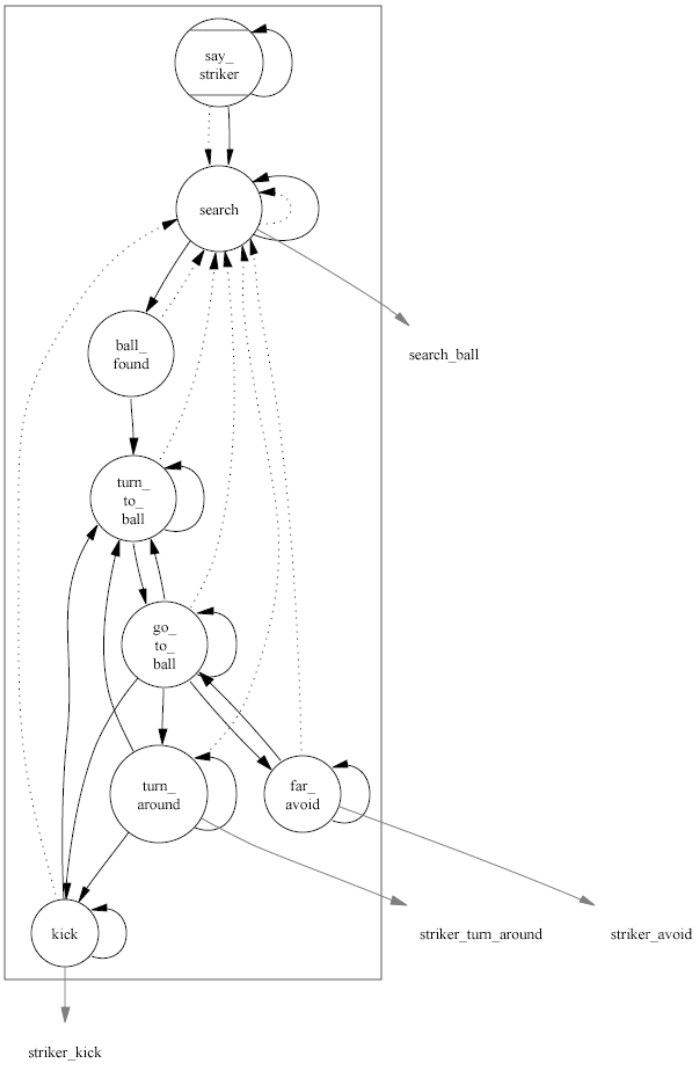


Fig. 19: State machine: playing_striker

4.3.2 Bottom level behavior

Fig. 20 shows the second module: ***striker_kick***. It is used to kick when the ball is near. Considering the spatial relations between the robot itself and opponent's goal, the ball, and the obstacles, we delicately classify the situations in to different cases, and design the corresponding actions. A pre-designed action is chosen by considering the three questions:

a. The neighborhood of the striker is occupied by the other robots or not? If yes, where are the opponent robots?
b. Where is the opponent's goal?
c. Where is the ball?

Fig. 21 shows the behavior of ***striker_kick*** under different situations. When the neighborhood is not crowded and the opponent's goal is distant, the striker will use the wide-angle kick to make a strong shot. If the opponent's goal is near, the striker will use fast-kick or side kick to make a quick shot for preventing the opponent keeper's blocking.

On the other hand, if the neighborhood is crowded, the striker will use the fast-kick action or the side-kick action to kick the ball quickly. If possible, it will select a kicking which can make the ball free from bumping into the obstacles and move toward the opponent's goal. The ***crowded_front*** and ***crowded_side*** cases in Fig. 21 show two examples of the kicking selection. So far this module is implemented by the multiple-layered conditional rules. It will be our future work to design a more general planning method.

[Code Release]
The bottom level behavior of our striker is implemented in the file *Modules/BehaviorControl/BH2010StableBehaviorControl/Options/Soccer/Striker/striker_kick.xabsl.*

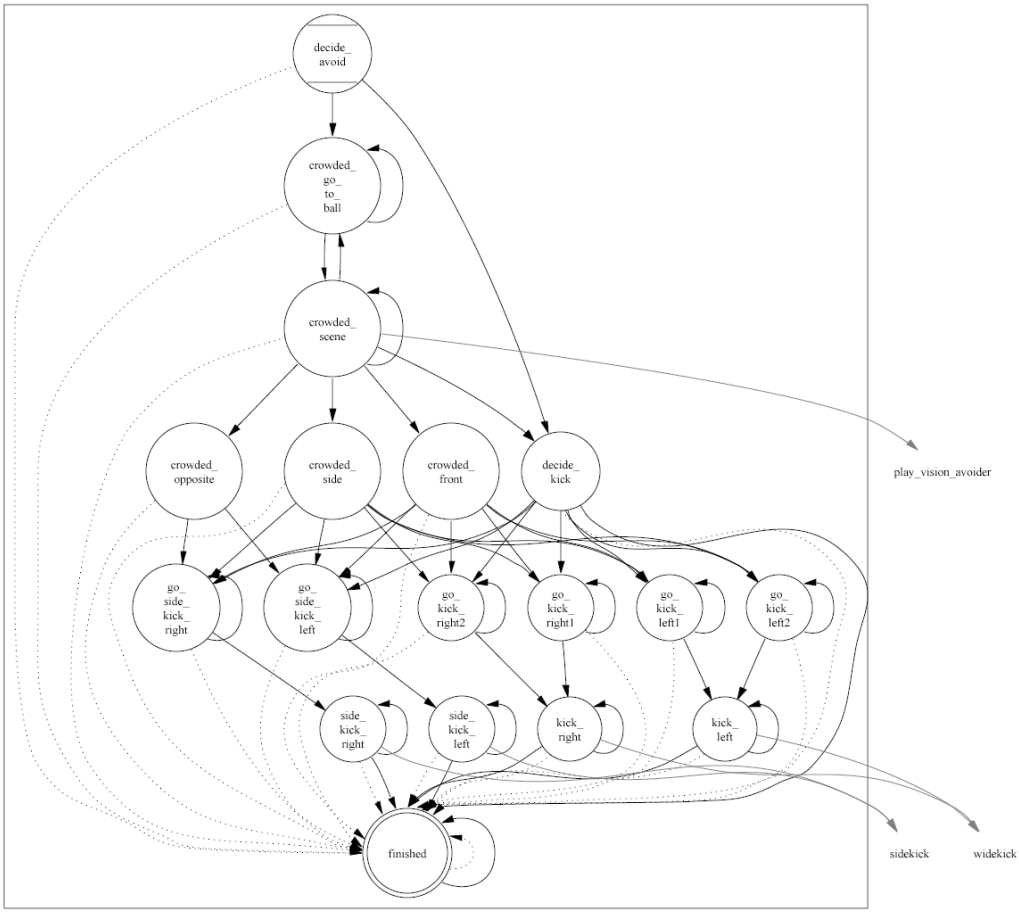


Fig. 20: State machine: striker_kick

4.4 Offensive Supporter

Fig. 22 illustrates the target position where our offensive supporter tries to reach. Our offensive supporter always tries to walk to a position next to and near the line connecting the ball and the goal, which is the assumed direction along which the striker kicks. We choose this position for three reasons: (1) the offensive supporter can easily reach the ball after striker's kicking, and thus becomes the next striker quickly, (2) the offensive supporter can observe the ball, the opponent's goal, and the striker, in order to assist the CLAT module, and (3) standing on this place will not hinder the striker's shot.

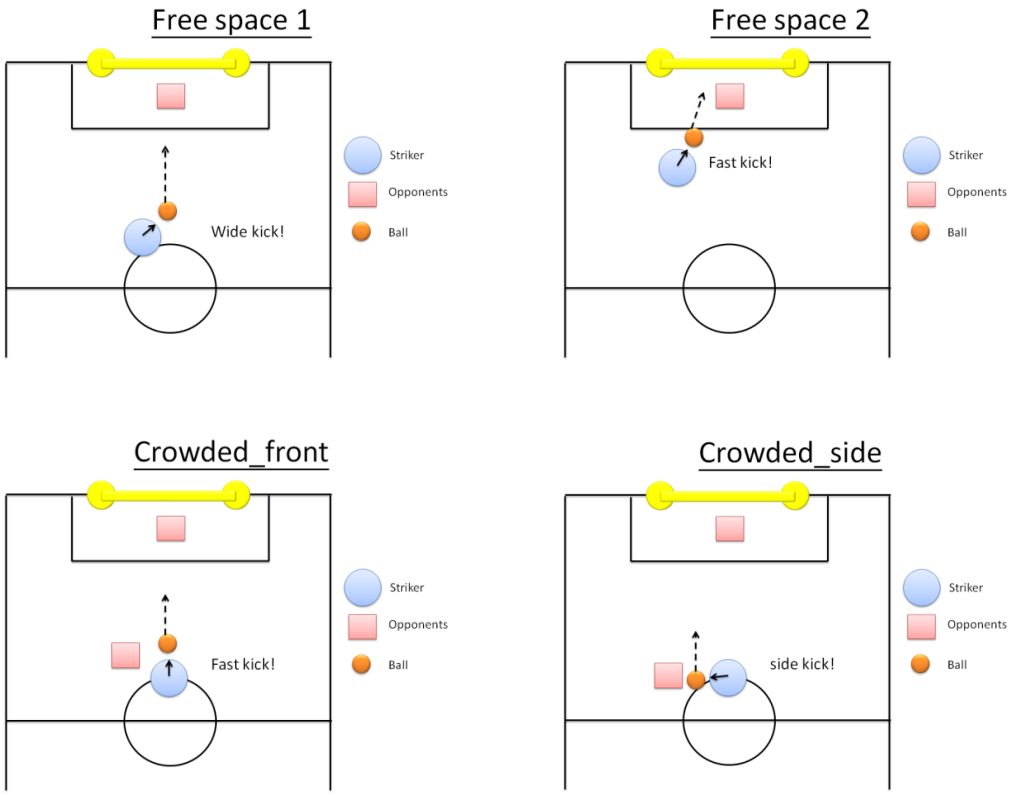


Fig. 21: Examples of situations and reactions of our striker

Once the striker starts the kick action, a message will be sent to the offensive supporter and the offensive supporter will actively track the ball in which the role switching can be done promptly if the offensive supporter should become the next striker.

[Code Release]

The offensive supporter behavior is implemented in the file *Modules/BehaviorControl/BH2010StableBehaviorControl/Options/Soccer/Supporter/playing_supporter.xabsl.*

4.5 Defensive Supporter

A defensive supporter helps with blocking the ball, so it always walks to the position between ball and our own goal. For not entering the penalty area and hinder the keeper, the defensive supporter will just stand on a position near our own goal when the ball is very near to our own goal. The desired position of a defensive supporter is also shown in Fig. 22.

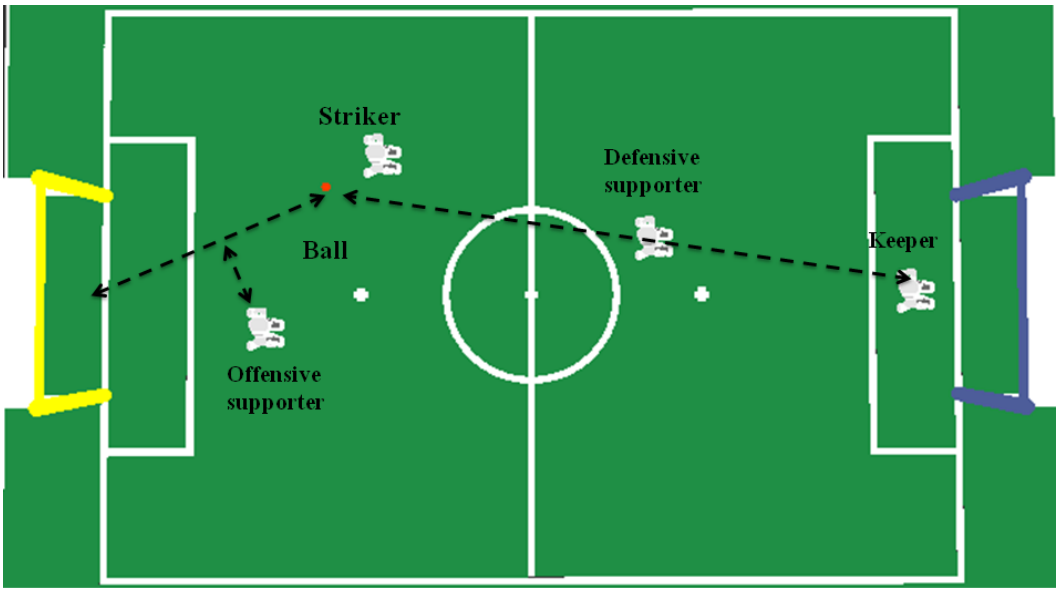


Fig. 22: Supporter positions

[Code Release]
The defensive supporter behavior is implemented in the file *Modules/BehaviorControl/BH2010StableBehaviorControl/Options/Soccer/Supporter/playing_defense_supporter.xabsl.*

4.6 Penalty Area Avoidance

Only our keeper can stay in our penalty area. This module is used to prevent the robots, except for the keeper, from entering the penalty area. This module firstly checks the position to see if the robot is going to be in the penalty area. If so, the robot will be asked to step away from the penalty area. Our striker and two supporters have to do the position checking using this module before performing their individual behaviors. Fig. 23 shows an example of combining this penalty area avoidance module for the striker, and this is done in the same way for the supporters.

[Code Release]
The penalty-area avoidance behavior is implemented in the file *Modules/BehaviorControl/BH2010StableBehaviorControl/Options/Soccer/Striker/ playing_striker_illdef.xabsl.*

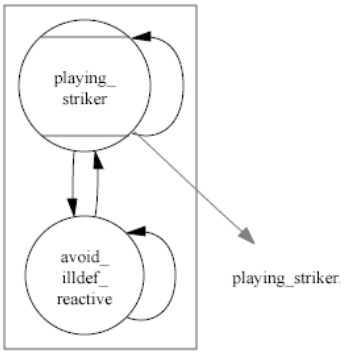


Fig. 23: State machine: playing_striker_illdef

4.7 Keeper

The keeper should just stay in the penalty area and keep tracking the ball. When the keeper finds that the ball moves too quickly to be blocked by just walking, it will dive to stop the ball. When the keeper finds that the ball stops nearby or just moves slowly without the chance to enter our goal, the keeper will perform the striker behavior temporarily to kick the ball away.

[Code Release]
The defensive supporter behavior is implemented in the file *Modules/BehaviorControl/BH2010StableBehaviorControl/Options/Soccer/Keeper/playing_keeper.xabsl.*

Acknowledgements

Team NTU RoboPAL was supported by Department of Computer Science and Information Engineering, National Taiwan University, and by National Science Council, National Taiwan University and Intel Corporation under Grants NSC100-2221-E-002-238-MY2, NSC99-2911-I-002-201, NSC 100-2911-I-002-001 and 10R70501.

A. References

1. T. Röfer, T. Laue, J. Müller, A. Burchardt, E. Damrose, A. Fabisch, F. Feldpausch, K. Gillmann, C. Graf, T. J. de Haas, A. H□artl, D. Honsel, P. Kastner, T. Kastner, B. Markowsky, M. Mester, J. Peter, O. J. L. Riemann, M. Ring, W. Sauerland, A. Schreck, I. Sieverdingbeck, F. Wenk, and J.-H. Worch, "B-human team report and code release 2010," 2010, only available online: http://www.b-human.de/file_download/33/bhuman10 coderelease.pdf.
2. Chieh-Chih Wang, Charles Thorpe, Sebastian Thrun, Martial Hebert and Hugh Durrant-Whyte. Simultaneous Localization, Mapping and Moving Object Tracking. *The International Journal of Robotics Research*, Vol. 26, No.9, September 2007, pp. 889-916.
3. C.-H. Chang, S.-C. Wang, and C.-C. Wang, "Vision-based cooperative simultaneous localization and tracking," in IEEE International Conference on Robotics and Automation (ICRA), Shanghai, China, May 2011.