Project Report - Teaching a NAO to Play Soccer

Nora Baukloh, Ruvita Faurina, Decebal Mocanu,
Christian Andrich, Henning Metzmacher

January 25, 2012

Abstract

This paper reports the design and implementation of a robot controller for two NAOs who play soccer against another team of two NAOs. Except for the number of playing NAOs, the official rules of the RoboCup are used. The focus of this report is on recognition of objects as well as localization, learning algorithms and the coordination of different software parts. The recognition of objects is of course very important, as in the soccer game the robots have to recognize ball, goals, other players and lines on the field. It turns out that color detection is the most efficient way to do that. The focus of learning was put on learning how to score a goal.
Keywords: NAO, Robotics, Artificial Intelligence, Soccer, Computer Vision

Contents

1 Introduction

In order to implement a concept of NAOs playing soccer, several aspects have to be looked at. The NAOs have to recognize the ball as well as both goals of the soccer field and they have to know at all times where they are positioned in relation to each other, the goals and the lines on the field. Another important

aspect of the project is the ability of the robots to learn certain strategies and optimal positions to score a goal, or to catch the ball.
The robots need to do several things at the same time. For example, they have to keep the ball in sight, while they move in a certain direction. At the same time, they might receive messages from their team mate and they must store any useful data they might be needing in order to react on different situations. The following points need to be addressed while implementing this:

- Recognition of objects:
  - Goals
  - Ball
  - Field lines
- Walk to a destination
- Obstacle avoidance
- Motions of the robot
  - Kicking the ball
  - Side- and forward-stepping in order to move to the optimal position to shoot or catch the ball
  - Crouch down to be as broad as possible (Goalie)
  - Stand up after falling
- Learning strategies
- Receiving and processing messages from the team mate

Most of these issues seem to be trivial for a human, but they are challenging for a robot, which does not know anything about its environment and is not able to interpret situations by itself. The term "goal" does not mean anything to the robot- but it can be taught that two areas of the same color intercepting a white line form a goal.
For motions like for example the kicking motion, it is important that the robot does not fall over while doing so - balancing is a big issue with a robot that walks on two legs.
Anything the robot sees is a camera image that must be processed. The robot cannot estimate how far away he is from the ball just by seeing it- this must be calculated. It is crucial that calculating things like this does not take too long, because the game of course continues.

1.1 Research Questions

The study focuses on the following Research Questions:

1. How can the NAO recognize the ball, the goals and the lines on the field?

2. Positioning
    (a) How can the NAOs determine good positions in the field?
    (b) Can these positions be learned?
    (c) Do these positions depend on the team's strategy?
    (d) How should the NAOs coordinate their behaviours?
3. Shooting the ball
    (a) How can we program a shooting module?
    (b) Is it possible to teach the NAO to shoot a ball?
4. Goal keeping
    (a) Which position should a keeper choose?
    (b) Can the optimal position be learned?

2 Environment

In this section the Hardware and Software environment that was used will be introduced.

2.1 Hardware

The robot used for this study is NAO, a humanoid robot developed by the company Aldebaran Robotics. This robot has been the standard robot used for the standard platform competition of the RoboCup since 2008.

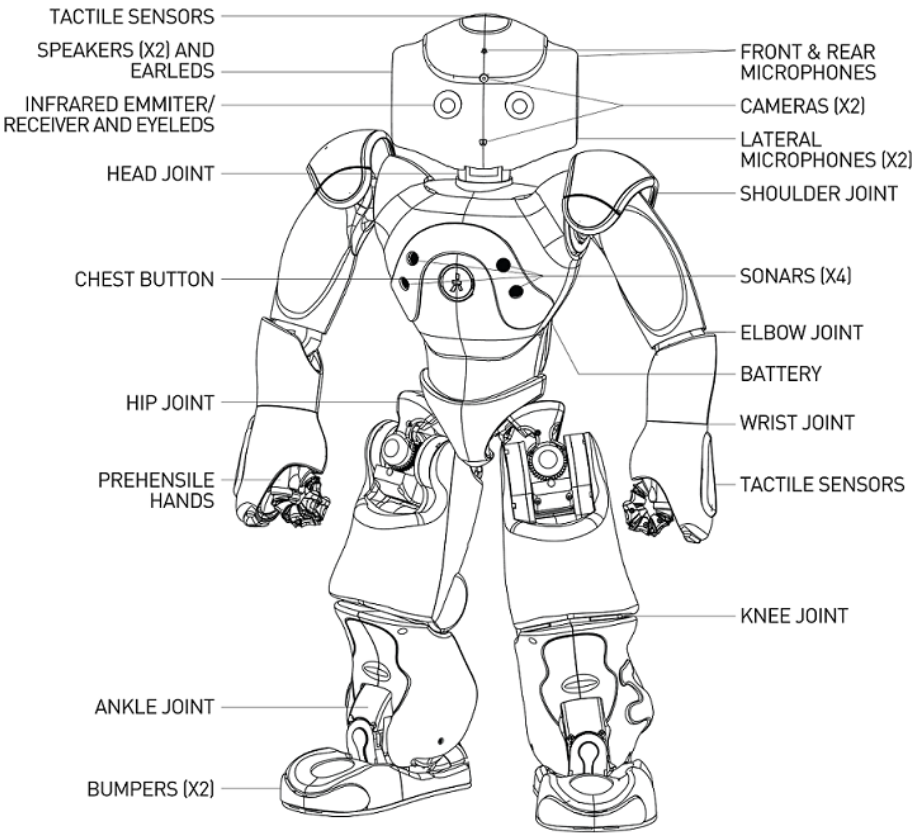

Figure 1: NAO Sensors and actuators

All joints depicted in Figure 1 are movable individually. In Table 1 the technical Specifications of the NAO are listed.

Technical Specifications	
Height	58 cm
Weight	4,3 kg
Autonomy	90 min. (constant walking)
Degrees of freedom	21 to 25
CPU	x86 AMD Geode 500 MHz
Built-in OS	Linux
Compatible OS	Windows, Mac OS, Linux
Programming languages	C++, C, Python, Urbi, .Net
Vision	Two CMOS 640×480 cameras[11]
Connectivity	Ethernet, Wi-Fi

Table 1: Technical Specifications

2.2 Software

The NAO comes with a SDK with a lot of examples which cover many of aspects of the robot, like how to extract pictures from the cameras or motion examples. Aldebaran also provides programs for graphical programming and for working with the camera images. For the longest time of our project, the SDK 1.10.52 was used until the software on the NAOs was updated. We then had to switch to the 1.12 SDK.
In the following list the used software is described.

1. **Programming Language: C++.** C++ was chosen as programming language for this project, because the NAO comes also with the SDK in C++. OpenCV, the object recognition software that was used, is also written in C++. Wrappers for other languages lack some of the latest features of OpenCV. Computationally expensive algorithms are also better in C++ because it is faster. Python was not chosen, because of the problems that came with different versions.

2. **Object recognition: OpenCV.** The *Open Source Computer Vision Library* provides programming functions to deal with computer vision.

3. **Recording software: Choregraphe.** Choregraphe is a graphical programming tool provided by Aldebaran. It is possible to record movements from the Robots and export the created code to Python. This can then be translated to C++ and be used in our program.

4. **Calibration of Vision: Telepathe.** Telepathe is a tool provided by Aldebaran which shows the camera images of the NAO and helps calibrating them.

5. **Simulator: Webots.** Webots is a simulator for many kinds of robots. The so-called naoqi_for_webots project, which simulates the NAO robot

can be downloaded from their website. There are some restrictions to Webots:

- The Chest Button is not available
- Camera Parameters cannot be set
- The physics engine is not very realistic. Any tests on Webots cannot represent the experience on a real NAO.

6. **UML Creation: BOUML**: BOUML can construct UML diagrams from existing C++ Code and has been freeware until Version 2.2.

3 Algorithms

3.1 Reinforcement Learning

Autonomous agents which are situated in a non-sequential real-world environment pose a challenging tasks to researchers in the field of artificial intelligence. The programmer needs to come up with algorithms that enable the machine to deal with uncertain occurrences and incomplete or distorted sensory information. Computers are very good in well defined, repetitive tasks but manually defining huge amounts of action-reaction cases for robots that are expected to show human-like behavior will quickly become infeasible.

Reinforcement learning is an approach from the machine learning area that is build around the idea that an agent is situated in an environment where it is able to take actions in order to maximize a reward or receive a punishment for certain actions if they fail. The idea is, rather than pre-defining routines in the robot, to give the robot the ability to learn a behavior through gradually and allow it to adopt using its reward system. The approach is very much inspired by the way humans learn.

3.1.1 Reinforcement Learning as a Markov Decision Process

Reinforcement learning is often implemented as a Markov Decision Process. It is a mathematical model that can be applied to situations where outcomes are partly random and partly under the control of a decision maker. A Markov decision process consists of:

- A finite set of states S
- A finite set of actions A
- $P_a(s, s')$ is the probability that action a in a state s at time t will lead to state s' at time $t + 1$
- $R_a(s, s')$ is the expected immediate reward received after transition from state s to state s' with transition probability $P_a(s, s')$

A reinforcement learning algorithm can then be implemented using the following definitions:

- Finite state of possible actions $a \in A$
- Initial state $s_0 \in S$
- Transition function $T : S \times A \to \mathbb{R}(S)$, where $\mathbb{R}(S)$ is a probability distribution over S
- Reward function $R : S \times A \to \mathbb{R}$

3.2 Q-Learning

During this project we focused on the Q-learning technique. Q-Learning is a reinforcement learning technique that works by learning an action-value-function which returns an expected utility. The algorithm chooses actions using a fixed policy which is described in Section 3.2.1).

Q-Learning works as follows:

- By performing an action $a \in A$, the agent can move from one state to the next
- Each state provides the agent a reward (a real or natural number)
- The goal is to maximize its total reward
- Q-Learning implements a function that calculates the *quality* of a state-action combination: $Q : S \times A \to \mathbb{R}$
- Initially Q returns a fixed value chosen by the designer
- On each state change (new reward is given) new values are calculated for each combination of a state $s \in S$: $Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha_t(s_t, a_t) \times [R(s_{t+1}) + \gamma \times \max_{a_{t+1}} Q(s_{t+1}), a_{t+1}) - Q(s_t, a_t)]$

3.2.1 Action Selection Policy

There are three action selection policies that are commonly used for Q-Learning namely ϵ-greedy, ϵ-soft and $softmax$. They differ in the way that they enforce exploration and exploitation. ϵ-greedy and ϵ-soft have the drawback that they select actions uniformly. The probability for selecting the worst possible action is as high as selecting the second best. The softmax selection policy tackles this problem by assigning a weight according to the action-value estimate of that action at a particular state. This is done using the following formula:

$$P(s, a) = \frac{e^{\frac{Q(s,a)}{\tau}}}{\sum_b e^{\frac{Q(s,b)}{\tau}}}$$

We decided to use the softmax action selection policy. It almost excludes the worst action which takes away the actions that could not possibly result in a goal.

3.3 Implementation

During the process of the project we thought about various approaches to generically integrate the reinforcement learning algorithm into the code modules of our robot. However, expectations in the techniques were initially too high and we quickly noticed the limits of these approaches. Especially the complexity of the continuous environment sets tight restrictions on the level of abstraction in which the world can be discretized into states.

Initially we extended the state machine to a Markov Decision Process (see Section 4.2.6 using restrictive transitions from one state to another. However, this turned out to be obstructive when we tried to combine the Q-Learning algorithm with the low-level functions of the robot since it introduces a lot of checking and iterating which is also not optimal in terms of efficiency. In the final version, we decided for a lightweight Q-table class that merely holds states that are identified by an arbitrary number of numerical values. A state holds a reference vector to the possible actions in that state. Moreover, a state stores a fixed reward value. A "Hit-Goal" state, for example, would offer a reward of 100. An instance of an action stores a text ID and its Q-value. It is important to mention that action objects are uniquely linked to state so its value is actually $Q(s, a)$. Initially we extended the state machine to a Markov Decision Process (see Section 4.2.6) using restrictive transitions from one state to another. However, this turned out to be obstructive when we tried to combine the Q-Learning algorithm with the low-level functions of the robot since it introduces a lot of checking and iterating which is also not optimal in terms of efficiency. In the our extended version, we decided for a lightweight Q-table class that merely holds states that are identified by an arbitrary number of numerical values. A state holds a reference vector to the possible actions in that state. Moreover, a state stores a fixed reward value. A "Hit-Goal" state, for example, would offer a reward of 100. An instance of an action stores a text ID and its Q-value. It is important to mention that action objects are uniquely linked to state so its value is actually $Q(s, a)$.

An iteration is performed by executing an action, gathering sensory input to determine the new state and, since there are no hard transitions, look this state up in the Q-table. After that, depending on the current use of the Q-Learning technique it can either be assumed that the detected state is actually the state the robot is in or a supervised feedback can be given through the robot's button layout.

3.4 Example Application

We set up an customizable example of how to use q-learning with the NAO robot. The idea is to make the robot learn how to position himself properly to aim for a goal 2. Initially it was planned to discretized the field into fixed width and length tiles and pre-define connection between action and states. However, as already mentioned, this is highly infeasible. Moreover, it turned out to be convenient to choose very little but essential features for the specific task. The following features where chosen. The float values from the sensor are discretized and floating numbers are rounded to the closest integer in the range: We set up an customizable example of how to use q-learning with the NAO robot. The idea is to make the robot learn how to position himself properly to aim for a goal 2. Initially it was planned to discretized the field into fixed width and length tiles and pre-define connection between action and states. However, as already mentioned, this is highly infeasible. Moreover, it turned out to be convenient to choose very little but essential features for the specific task. The following

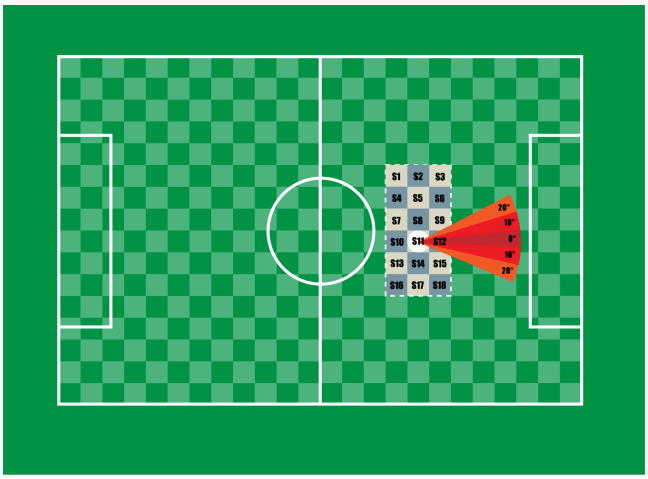

Figure 2: Visualization of the Q-Learning implementation.

features where chosen. The float values from the sensor are discretized and floating numbers are rounded to the closest integer in the range:

- x and y value on the field
  - The resolution can be set arbitrarily
  - An offset can be set on each side so that only a certain part of the field is taken into account (e.g. the space before the goal)
- The angle to the goal
  - The resolution can be set arbitrarily
  - A good choice is in the range between $-20°$ and $20°$ with a resolution of $10°$ resulting in 5 different feature instances
- Distance to the ball
  - This feature was held fixed for simplicity

Moreover, the robot button layout is used to give the following feedback:

- Hit Goal 100
- Miss Goal -100
- Miss Ball -200 (possible but not applied since the value is held fixed)

3.5 Results

Specifying the learning task and carefully selecting features makes it possible to analyze the learning behavior of the robot. Starting with a toy example of 3 aligned states in front of the goal it is possible to observe the convergence of the values after a few iterations. From the outward pointing actions at the boarder of the outer states it is clear to see that this approach works at least for this simple case. Since these states guarantee that the NAO does not hit the goal they are quickly excluded from selection. Specifying the learning task and carefully selecting features makes it possible to analyze the learning behavior of the robot. Starting with a toy example of 3 aligned states in front of the goal it is possible to observe the convergence of the values after a few iterations. From the outward pointing actions at the boarder of the outer states it is clear to see that this approach works at least for this simple case. Since these states guarantee that the NAO does not hit the goal they are quickly excluded from selection.

It is important to mention that Q-Learning works well in this example because there are rewards or punishments after almost every action. If the same approach was used for general positioning the rewards are much more sparse and need to be estimated differently. It is important to mention that Q-Learning works well in this example because there are rewards or punishments after almost every action. If the same approach was used for general positioning the rewards would be much more sparse. In this case it would be necessary to estimate rewards for "better" or "worse" states.

4 Implementation

4.1 Gamecontroller

The game controller is software controlled by a person, which sends messages to the robots providing them with the actual state of the game. When a robot acted against the rule, like for example pushing another robot, it will be penalized by the game controller which means that the robot has to stop moving completely. By the official rules, the NAOs can either listen to the Game Controller and receive messages via UDP or the states are switched by pressing the chest button. To keep things simple, it has been decided that only the button interface will be used during this project. In Figure 3 the different states the

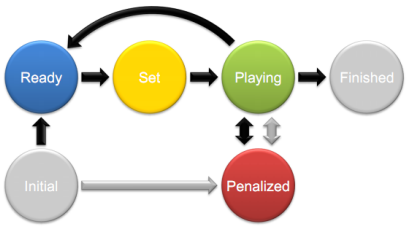

Figure 3: Gamecontrol states

robot can be in are depicted. Black transition arrows mean that these states are switched by sending the messages to the robots. Grey arrows show the possible transitions that can happen when only the button interface is available.
The possible state meanings are as follows:

- Initial: Button interface is active, the robots are not allowed to move besides standing up. By pressing the foot bumpers, the team color is switched.
- Ready: The robot walks to the kick-off position.
- Set: Head movement is allowed, but no locomotion in any fashion
- Penalized: No Movement at all is allowed, including head motion.
- Finished: Reached when a game half is finished.
- Playing: The robots play soccer.

With just the button interface, only Initial, Penalized and Playing state is available.

4.1.1 Button Interface

The basic use of the button interface is switching the robot between *penalized* and *unpenalized* state. In order to be able to react on penalization immediately, a separate thread continuously sends information about the button status to a *RobotStatus* class. If the button is pressed, the robot moves into a stable pose. As long as the robot is penalized, it will perform no other action. In the *penalized* state it is possible to switch team colors by pressing the left foot bumper. The Teamcolor is then displayed at the left foot LED.
Additionally, the red foot bumper is used to switch between goalie and field player role. The role is displayed at the right foot LED: Red for field player, blue for goalie.

4.2 Program Logic

The program logic is based on one initial module that initializes all the other components of the program. Most components are implemented as Singleton. For controlling the behaviour of the robot, a state machine is used. There are two different machines for the goalie and the field player, shown in figures 4 and **??** The boxes have different colours, each colour representing a component of our program.

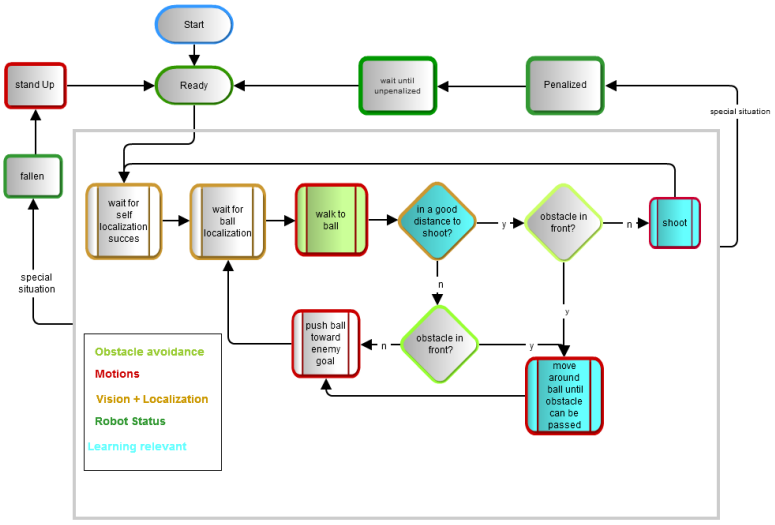

Figure 4: Behaviour of field player

In the following sections, the classes and their key features are described.

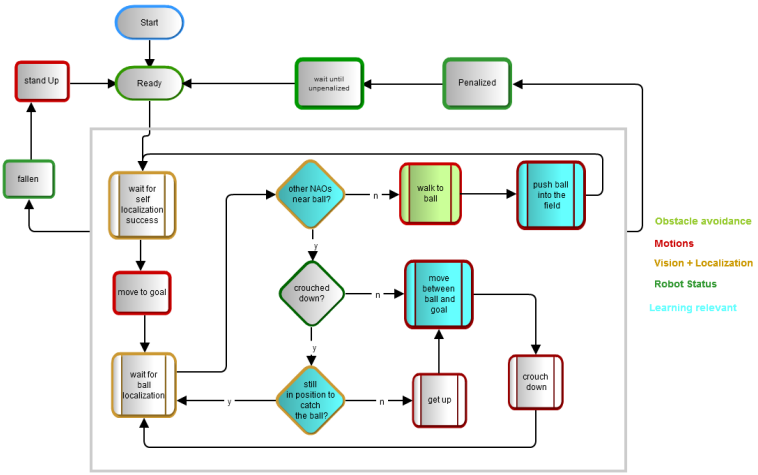

Figure 5: Behaviour of goalie

4.2.1 Main Module

NAOSoccer: The main module of the program. Initializes all other objects of the program. It provides a method to create the state machine for the field player or the goalie, and executes the state machine routine. This routine continously checks the whether the robot has fallen down or has been penalized. The next state and action is determined by the mdp but penalization and falling overrites all other states.

4.2.2 Basic Data

- *Singleton*: A template that classes can implement in order to be a Singleton.
- *RobotStatus*: This class serves as contact point for all other components. It stores information about any relevent data, including sensor data, ball and goal visibility and location, penalized status and much more. It provides getters and setters for this data. It reacts on sensor data provided by the *StatusMonitor*, e.g. if the robot seems to have fallen down, judged by it's pose, it will set a boolean to fallen, so other components of the program can react accordingly. It does not execute any functions by itself.
- *Leds*: Controls the turning on and turning off of the Foot Leds.
- *constantValues*: Header file for global constants.
- *StatusMonitor*: Runs in a separate thread and monitors sensor data from the robot, including the robot pose, sonar values or whether or not the

chest button and bumpers has been pressed. It updates the *RobotStatus* class continuously.

4.2.3 Vision

- *Locator*: Manages vision and object localization in a separate thread. It computes distances and angles to the ball and the goal, and provides input to the particle filter. The Module provides wrappers for the walking methods in order to keep the robots movements in synch with the Particle Filter. It updates the *RobotStatus* with information about the balls and goals position and angle and their visibility.

- *ParticleFilter*: Implements a particle filter for localization of the robot. The Particle Filter is able to provide the Robots estimated absolute position and angle. Additionally the field gets divided into parts by a static grid. For every part of the grid the Particle Filter provides the probability of the Robot being in that part.

4.2.4 Motions

- *Kick*: Executes kick commands. The robot can kick either with the left or the right leg. A dynamic kick method is implemented, where the strength of the kick is determined by the given parameters.

- *GoalieMotions*: Controls the motions the goalie does. It can move from standing into a crouching position being as broad as possible, and stand back up.

- *Walk*: Controls any walking or stepping motions that might be needed. Obstacle avoidance is integrated into this class. Instead of calling the standard motion routines of the robot, it calls the routines from the *Locator* class, which provides wrappers for these, in order to be able to update the particle filter.

- *StandardMotions*: Contains methods to execute standard motions of the robot, which both robots might need. These include the standup Motion after having fallen, or moving into a stable position. Additionally, the stiffness can be turned on and off with this class.

4.2.5 Communication

- *socketServer*: Starts a communication server and waits to be queried by the other robot. This class is run in a separate thread. The information provided are: his own location, ball position.

- *socketClient*: Starts a communication channel with a server from a partner robot and query it about his location or ball location. This class uses Singleton pattern.

4.2.6 NAOMDP

- *MarkovDecisionProcess*: The base of the MDP. It is completely generic and an arbitrary number of actions, states and action-state transitions can be inserted.
- *MarkovAction*: A class from which all defined actions are inherited.
- *MarkovActionStateTransition*: Defines a transition from an action to a state. A probability with which the state is reached must be set. In our program, these probabilities are always 1, causing the MDP to basically be a simple statemachine, which can be extended to an MDP at any time. If a transition is defined, it must directly be added to the mdp.
- *MarkovState*: A State in the MDP.
- *QAction*: Class to define rewards for actions. Used for the implementation of QLearning. A Q-Action must always be directly added to the MDP, along with a state that it belongs to. The rewards are defined when the state and action are added to the mdp.
- *Random*: Random Number generator.

States of the MDP

- *Initial*:The NAO is in initial state
- *Penalized*: The NAO is penalized
- *Fallen*: The NAO has fallen down
- *GoalieGoodPos*: The Goalie NAO is in a good position relative to the ball
- *GoalieBadPos*: The Goalie NAO is in a bad position relative to the ball
- *seeBall_far1* to *seeBall_far4*: The NAO sees the ball in a distance, as actions are executed he moves closer (far4 being the closest)
- *atBall*: The NAO is close to the ball

Actions of the MDP

- *GoalieAdjustPos*: The Goalie NAO sidesteps until its angle to the ball is in an acceptable threshold. In the *RobotStatus* class, the steps taken to the left or right are counted. If the NAO cannot see the ball, he sidesteps back to the middle of the goal. If it is in the middle and still does not see the ball, we assume that the ball is somewhere behind him in or next to the goal.
- *GoalieStayPos*: The Goalie NAO is content with it's angle to the ball and stays in this position until that changes, or until he does not see the ball anymore.

- *KickAction*: The NAO kicks. Which foot to use is determined by checking the angle to the ball. If the angle to the ball is too big, short side steps are executed before the kick.
- *Wait_Localize*: The NAO waits until it has the ball localized. If the NAO is the field player, he turns 180 degrees while looking for the ball.
- *Wait_Unpenalize*: Do nothing until the chest button is pressed again.
- *WalkAct1*: Approach the ball
- *WalkAct2*: Look at the goal and step sideways in a circle around the ball until the angle is acceptable for shooting. If in this process the distance to the ball gets too short, a backward step is executed. After three sidesteps, the NAO looks at the ball and updates the distance to the ball.
- *WalkAct3*: Adjust the position relative to the ball by stepping to the left or right
- *WalkAct4*: Do small steps forward until close enough to shoot the ball.
- *Standup_Act*: NAO stands up after he has fallen. Depending on the robot pose, it executes different motions for standing up from the front or from the back. Should it lie on its back, it first moves into the *Back* position.

4.3 Program Structure

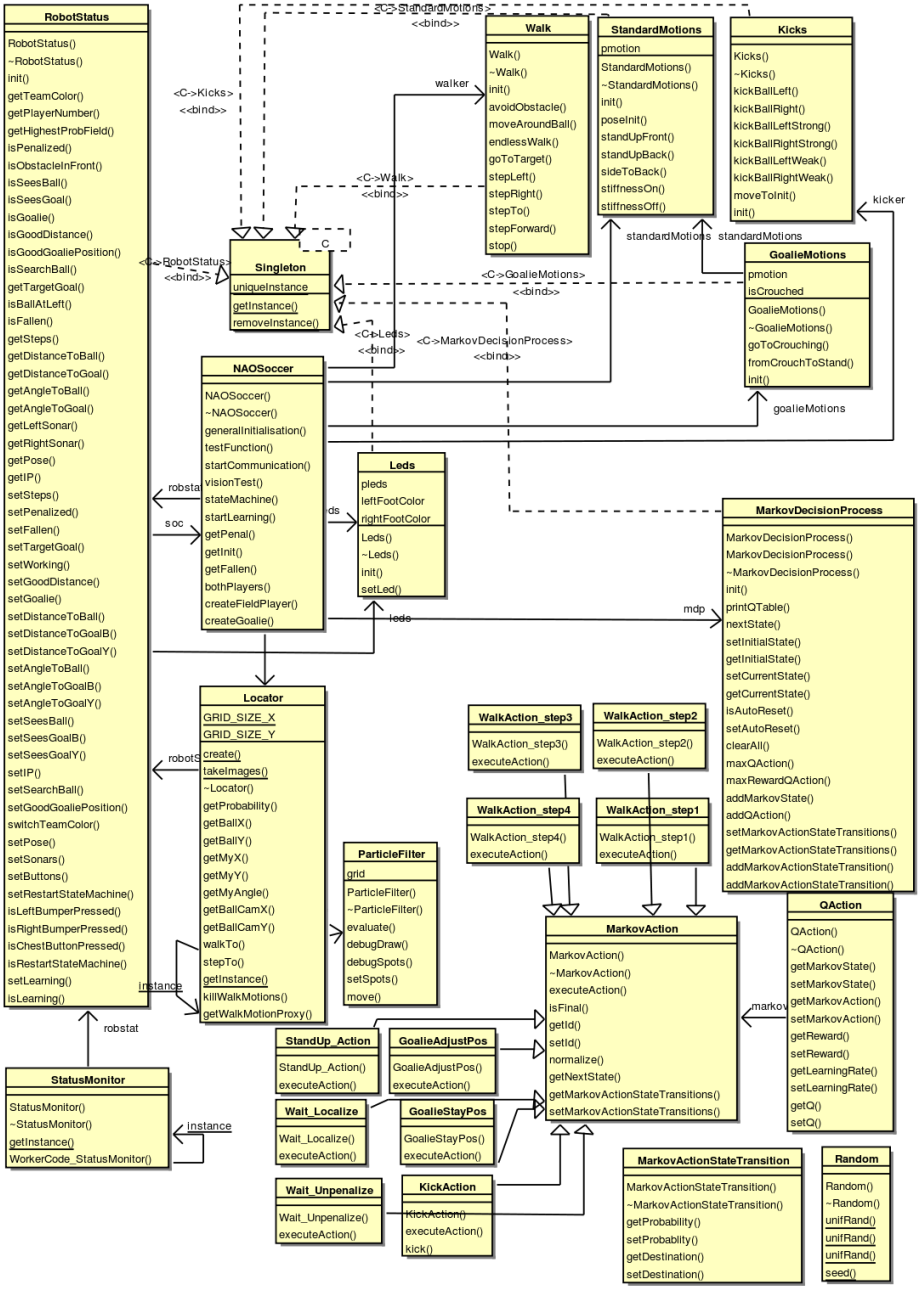

Figure 6: UML Diagram without private members and functions

In figure 6 the UML diagram showing the connectivity of the different classes is depicted. Private members and functions are not displayed, allowing for a better overview. Each class will be described below.

4.4 Motions

The motions of the robot are for a part standard motions from the SDK, for example walking. Standard motions, like going into the init pose were taken from the provided examples in the SDK. Any movements that make the robot change its position or its angle are wrapped by the *Locator* in order to update the particle filter as well.
For special motions, like for example kicking and the crouching position of the goalie, the export function from Choregraphe was used. This means, that the real robot was moved into certain positions, and the pose was saved in a Choregraphe box. From here, the code describing the angles of the joints can be directly exported into C++. This is a standard function of Choregraphe.
In order to create complex motions that consist of more than one pose, key poses of these animations were saved. Within Choregraphe, these snapshots were executed one after the other and, if necessary, adapted until the real robot was able to execute the motion flawlessly. Since the pose boxes in Choregraphe use the standard AngleInterpolationBezier method in order to move into different poses, the transitions between poses are smooth and the robot computes by itself which joints have to be moved in order to move into the desired pose. So the difficulty is to find poses that the robot can reach from its previous pose without losing its balance.
In the AngleInterpolationBezier method, 3 lists have to be provided: One with the names of the joints, and corresponding to those, the desired angles and times. If the motion addresses 20 joints, the names list has 20 entries, but the times and angle lists are two-dimensional, the second dimension being the number of steps needed for the pose.
With the times list, it is possible to make the NAO move faster or slower. This was used to design the strong and the weak kick, it is the same motion but executed with different speed.
All animations are saved in .txt files to be included to the code wherever they are needed. This allows a clearer code without long text blocks of joint values.

4.4.1 Obstacle Avoidance

Obstacle Avoidance is done using both sonar senors [1]. The range of the sonar sensor is up to 0.7 meters. The main problem is the inaccuracy of these sensors. For this reason and also for computational reasons the algorithm tries be as simple as possible. Because the soccer field does not have too many obstacles, the algorithm performs quite well. It treats two possible cases : when the robot is in the field without ball, and when the robot is in the field and has the ball.
Obstacle avoidance without ball - in this case, if the sensors detect an obstacle in front at a distance smaller than 0.3 meters the robot will enter in a procedure

to pass the obstacle. In this procedure it will change its direction and move into that direction for a very small distance. It will repeat this procedure until it will not have any obstacle in front anymore. After that it will localize the ball again and it will move to it.
Obstacle avoidance with ball - in this case, if the sensors detect an obstacle localized between the ball and the goal which can interfer with its shot, it will move around the ball until the obstacle is not detected anymore. After this it will push the ball to pass the obstacle.

4.4.2 SDK update: Problems with Motions

With the SDK switch, some problems with earlier recorded motions arose due to incompatibility of the two Choregraphe versions. Some motions did not work as intended anymore, causing the NAO to do weird movements or even to fall. The solution to this problem was to re-record these motions with the new Choregraphe. This happened to at least the "moving to init-Pose" motions, and possibly to other motions as well which has not been discovered.

4.5 Communication

The communication between NAOs is implemented using a TCP/IP client server architecture. In picture 7 it can be seen how the information is exchanged between robots. When each robot starts, it creates a new instance of the *socketServer* class which will be run in a separate thread. This server uses the *Locator* class to update its knowledge about its own status. When a robot needs some information from the other robot it calls the instance of the *socketClient* class. This class will create a socket which binds to the server of the other robot. The information which are exchanged are : robot position and ball position.

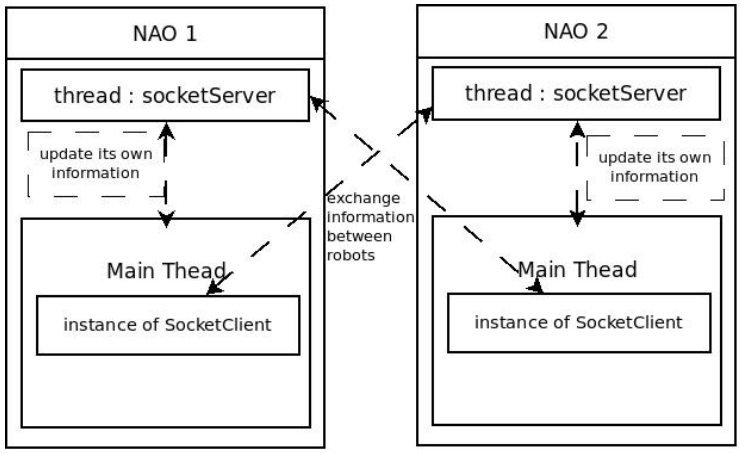

Figure 7: Communication between robots

4.6 Vision (Locator)

The Locator Module is used to find the Robots position on the soccer field, and to determine where the ball and the goals are. This of course is a very important module for being able to play soccer and thus it has been worked on during the whole semester. During the second block of the semester, a course changing decision was made because the already implemented code for the Locator Module was found to be ineffective. Due to this decision two approaches have to be described. The first approach is labelled as "Experimental" and is not used in the final product anymore. The second approach is used in the final product and is working much better than the experimental approach.

4.6.1 Camera Parameters

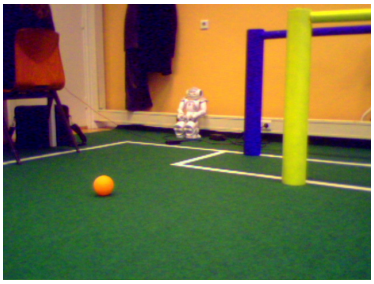

Figure 8: Example camera shot on the soccer field in Maastricht

Due to the fact that our group had three different testing environments, the soccer fields in Aachen and Maastricht and the Webots simulator, it made sense to have a set of camera parameters for every environment. These camera parameters where used to achieve the same camera images regarding brightness, saturation, contrast, white balance, etc. in every environment. An example camera shot can be seen in Figure 8. The parameters made it possible to have stricter color filter algorithms, which is necessary because some colors are very similar and hard to distinguish. These camera parameters need to be applied after every camera change. The camera parameters are defined via constant variables so it has to be defined in which environment the program is supposed to run at compile time.
Problems occurred in the Webots simulator because changing camera parameters there leads to a crash of the simulator. To overcome this problem, the camera parameters are not changed when the program runs in Webots, but in order to still have the same colors the colors of the simulated environment where changed directly in Webots. This fact lets the colors of screenshots made in Webots appear unnatural, but the color correlate to what the NAO camera would see in Aachen or Maastricht.

4.6.2 Experimental Ball Detection

For ball detection, Hough transformation method is used. This method was chosen because it is tolerant of gaps in feature boundary descriptions and is relatively unaffected by image noise and also the ball is presumed to be a circle on a 2D image. OpenCV has a function which supports the implementation of circle (ball) detection using Hough method, the *cvHoughCircles* function. Before detecting the shape (circle), a Color classification has to be done to allow fast identification of approximate areas of objects such as goals and balls. The full resolution image can be used to accurately determine the location of each of these objects in the frame. In order to filter out colors which do not match the orange color of the ball, the image gets converted into HSV color space. In this color space it is relatively easy to filter orange due to its specific hue. The result is a black and white image with white for pixels which are near orange and black for pixels which differ too much from orange. Smoothing is applied to prevent a lot of false circles from being detected. After the smoothing process is done, cvHoughCircle is applied to detect circle shape that represents the ball. The results are the 2D coordinates of the circles center in the picture and the circles radius. The next step is to determine the 3D coordinates of the ball. This would have been done with linear algebra methods, but due to the decision to drop the 3D analysis this was never implemented. This already implemented way of detecting the ball was removed from the final product and replaced by a more simple approach.

4.6.3 Experimental Goal Detection

The process of goal detection is more complex than ball detection. The goal has a very specific structure, which is hard to find in a picture. To identify the goal, it has a specific color, which can be filtered from the image like the orange color of the ball. The result is again a black and white image, where only pixels which are in the goal color are white. The goal basically can be represented through three connected lines. This is why line detection was used to identify the goal. Due to the fact that the lines of the goal are relatively thick, several pixel lines will be found for every goal line. To find the best three lines which represent the goal we search for the two lines, which have the biggest area under the assumption that they span a quadrangle. These two lines represent the two posts. The line which represents the bar will not be searched because it can be calculated from the two past lines. After the two lines have been found, the four endpoints will be used for further computation.

4.6.4 Experimental 2D to 3D coordinate transformation

Due to the fact, that we know the real dimensions of the goal, we could have computed the 3D coordinates of the goal relative to the camera if we would have known the four edges of the goal as 2D coordinates. As a matter of fact, the NAO camera is not perfect and thus delivers a distorted image. To accurately calculate the 3D coordinates, we have to calculate the undistorted

2D coordinates of the four edges. To achieve this, the camera matrix has to be calculated. This is done with a chess board presented to the NAOs camera. The NAO searches for edges in the chessboard and calculates the camera matrix for the 2D coordinates of the chess board edges. This is possible due to the fact that all squares of the chess board are equal in size. This functionality is included in OpenCV, the function call is cvFindChessboardCorners. Once the camera matrix is computed, it can be stored and reused after termination of the program. With the camera matrix, the measured 2D coordinates of the goal corners and the 3D dimensions of the real goal, the 3D coordinates of the Goal relative to the camera can be computed. OpenCV provides this functionality with the cvFindExtrinsicCameraParams2 function call. Experiments with this method have shown, that the whole process, from detecting the goal to finding it's 3D coordinates, works on the NAO, but is very slow. On average the process took five seconds. It was decided that this time is too long for a single frame so this method will not be used to find the NAOs location relative to the goal.

4.6.5 Threaded Locator Module

The final Locator Module is a Threaded Module. This means, that an independent thread is used for this Module. This allows independency from the other action the robot does, so it is possible to always analyze the camera image in the background and update the State Machine with the knowledge of the world which is gained through the vision of the robot. For updating the State Machines knowledge, the *RobotStatus* is used. The Locator Module will tell the *RobotStatus* the gained information after every iteration of the Locator Modules algorithms. These information include whether the robot sees the ball or the goals, and if it sees them, the distance and the angle to them are calculated and shared with the *RobotStatus*. Iteration of the algorithms means the process of getting a camera image and completely analyzing it. To analyze an image, the following steps will be made:

- Field extraction to be able to ignore unimportant and potentially confusing parts of the image.
- Ball detection to determine if the ball is seen and to find the balls distance and angle relative to the robots main body.
- Goal detection to determine if a goal is seen and to find the goals distance and angle.
- If an item is not seen: search for that item.
- If an item is seen: focus on that item.
- At every tenth iteration: detect field lines or field line intersections and provide the detected intersections or line spots to the particle filter.
- At every tenth iteration: Iterate and evaluate the particle filter.

What to search for is indicated in the *RobotStatus* and checked at every beginning of an iteration in the Locator Module. The item to be searched for is either the ball or the target goal (yellow or blue). Checking *RobotStatus* is also used to determine if the robot is fallen or on penalty, which results in suppression of all movements (search for and focus item).
The particle filter algorithms are only called every ten iterations because they take longer than any other algorithm. This is no disadvantage because compared to the ball, the robot moves very slow and its position updates much slower than the balls position. The field line detection provides two ways of operation, which are both described later on, as well as the reason for having two ways.

4.6.6 Field Extraction

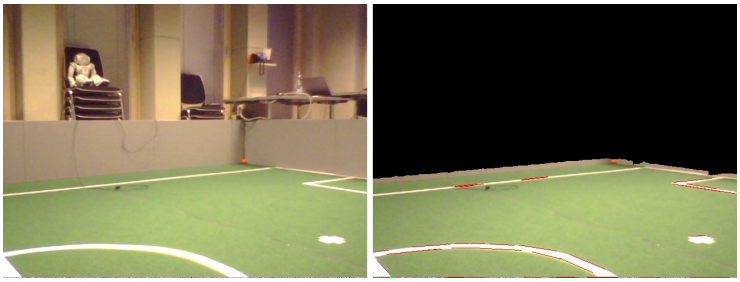

Figure 9: Result of the field extraction algorithm

An often reoccurring problem is caused through objects which the NAO camera sees in the background but which do not belong to the soccer field environment. An example is an orange wall in the background, which happens to have a similar color like the ball. This can cause serious problems in the final ball extraction algorithm and thus needs to be filtered out. To filter this out, we had to decide what is of importance to our vision algorithm and what is not. We decided to write an algorithm which cuts everything away from the top of the camera image to the bottom, and stop cutting away if the green soccer field is recognized. Cutting away means painting black. An example output of the algorithm can be seen in Figure 9.
This also includes cutting away the goal poles if the camera sees them above the green field. We still need the lowest part of the goal poles for our final goal detection algorithm, so we stop cutting away not at the beginning of the green field, but a few pixels before that. This results in the goal poles being slightly visible.
During testing of the algorithm, it became clear that it had some problems. The biggest problem was deciding what belongs to the green field. This was problematic because the field appears less green if it is far away from the camera. This resulted in an algorithm, which cuts away part of the field, even if they are useful for further algorithms. Loosen up what is considered to be the green field

solved this problem but introduced new problems. An example is the shadow of the blue goal pole on the gray wall in Aachen, which was considered to be the green field. This lead to an algorithm which did not cut away enough. There is no real solution for this problem but the current version works quite well in Maastricht. The reason for this may be that the chosen color thresholds are overfitted to the lightning conditions in Maastricht.
Despite the problems this algorithm has, one very simple but effective improvement is limiting the camera angle to a maximum horizon. This means that the camera is only allowed to look as high as it is absolutely possible. To determine how high the camera may look we chose an angle which makes the robots camera see from one corner of the soccer field to the away corner most far away from the camera, such that the far away corner is seen at the top of the camera image. It may now happen that the ball is not seeable because it rolled out of the field, but if this happens we are not supposed to run for the ball anyway.

4.6.7 Angle and Distance Calculations

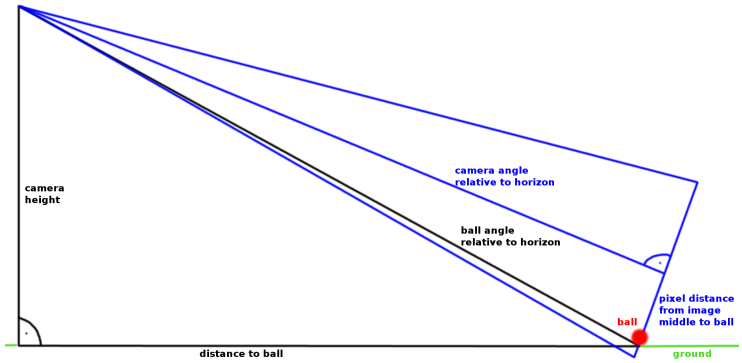

Figure 10: Geometrical dependencies of distance calculations in Locator Module

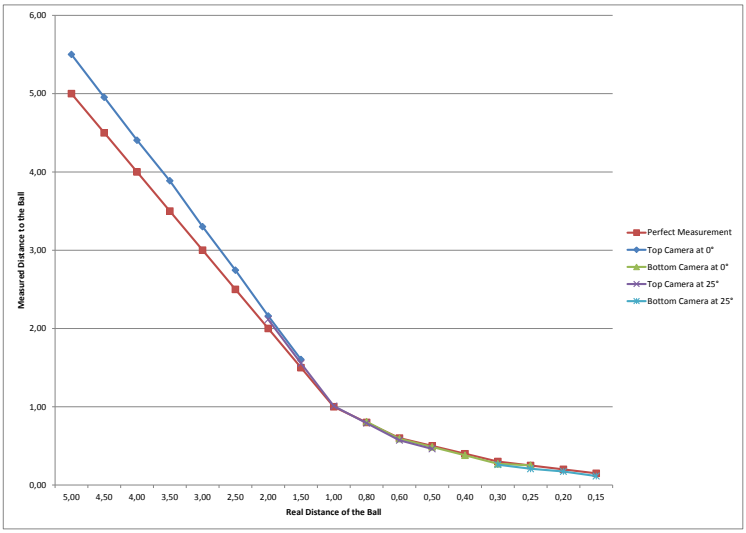

Figure 11: Analysis of the Distance Calculation Accuracy

Every information we get from the image can be easily linked to a position in the camera image which are the 2D coordinates of a pixel in the camera image. To get useful information of these pixel coordinates, we have to transform them into something more useful. In all cases in our further algorithms, we need to know where the pixel we find interesting is in the real world, relative to our robot. If we only consider pixel coordinates of an object which lies on the ground, which is the only thing we need for all further algorithms, we can calculate the angle and distance to that object with methods of linear algebra. Calculating the angle to an object relative to the camera is very easy because we know the field of view we get from the NAOs camera in degrees, so we simply can say things like "objects which are at the upper corner of the image are x degrees above the middle of the image" where x would be half of the cameras vertical field of view. Given this fact we can compute horizontal and vertical angles of any pixel coordinate relative to the middle of the camera image. If we add the angle of the NAOs head relative to the main body to a horizontal angle, we get the objects horizontal angle relative to the robot. This is for example used to find the angle to the ball.
The second information about objects like the ball is the distance from the robots feet to the object. In figure 10 we see the geometric dependencies of how such a calculation looks like. We know the pixel distance from the middle of the image to the ball and therefore also the vertical angle to the ball. What we need to know is the angle to the ball relative to the horizon. We can compute

this because the NAOs SDK provides functionality to get the camera angle relative to the horizon. The NAOs SDK also provides the cameras height above the ground which makes it possible to calculate the distance to the ball with trigonometric functions. An analysis of the quality of the results is shown in Figure 11. The measurements where done with a fixed angle for the Head Yaw and thus the object may be out of sight for certain distances. The figure shows that the calculated distance is not perfect. Objects which are far away from the camera tend to be seen even farther away. Objects which are very near appear to be seen quite precise, but if an object is very near even errors of a centimeter can be fatal.
The results are not perfect because the values provided by the NAOs SDK are a bit imprecise and thus the distance and angle to the ball we calculate get imprecise. Another reason is that if the object is near the center of the camera image, even small error result in big errors in the distance calculation. As described later on, when focusing on an item it is centered in the camera image. A better solution would be to look a bit above the item.

4.6.8 Final Ball Detection

The final ball detection algorithm is simpler than the experimental one. They have in common that the image is color filtered to find orange pixels, but the final algorithm simply calculates the average 2D coordinates of all orange pixels in the camera image. The final algorithm also calculates the average position during the color filtering, so storing the filtered image is unnecessary which reduces calculation time and memory usage. From the average coordinates the distance and angle is calculated and the result is provided to the *RobotStatus*. The ball is considered to be seen if there are more than three orange pixels visible in the camera image. Despite being very simple, this algorithm showed to be very good for ball detection.

4.6.9 Final Goal Detection

Like the final ball detection, the goal detection is very simple. Again, the whole algorithm is done during color filtering. What is different is that it is searching for two colors: blue and yellow. Both colors do not get mixed in the analysis of the pixels, so there are two results, one for each color. For goal detection the result is not the average of the correct colored pixels, but the lowest one, to find the lowest part of a goal pole. This is the spot where the goal pole touches the ground, which is a requirement for the distance calculation.
Of course there are two goal poles for each goal. To take this into account, the result can be multiple pixels. To decide if the algorithm has found a lower pixel on the first goal pole or the first pixel of the second goal pole, a distance threshold of the horizontal pixel distance is checked.
The final result of the algorithm are two lists of pixels, one for every goal color. A list of pixels contains one, two or zero pixels depending on if we see one, two or zero poles of the goal. Of course due to noise, it is not impossible that more

than two poles are recognized. To filter out this noise, the distance and angle to the goal is always calculated from the average position of the pixels in the list.
For the two pole case this means that the state machine gets told where the robot sees the middle of the goal. For the one pole case the robot gets told the exact position of the pole he sees. Due to the fact, that in field extraction most of the goal is cut away, there is no way to determine if the robot sees the left or the right pole if only one pole is visible. To determine this, the robot has to calculate the probability for both poles given the estimated position and angle of the robot provided by the particle filter. The pole with the higher probability is likely the pole the robot is currently seeing.

4.6.10 Line Intersection Detection

After finding that orienting by looking for the goal is too slow, the new approach was to detect field lines and orient with intersections of the lines. These intersections can be edges of the field or T-crossings of lines. Experiments with the NAO showed that the field lines are very bright and clearly to see with the NAOs camera. Based on this knowledge a black and white image is generated with white where a field line was detected and black for everything else. Detecting the field lines is done by a simple threshold for every pixel which has a high brightness and a low saturation value.
Experiments showed that problems occurred if there are white objects like walls in the background. These match the threshold, too. To avoid this problem, only white pixels were used which have a field-green pixel nearby. The idea was dropped later on, because it showed that under non-perfect lightning conditions, the green field appears more gray then green if it is too far away from the camera. This resulted in a situation where field lines were only recognized if they were near the NAO. This problem was solved by introducing the Field Extraction algorithm.
After creating the black and white image where lines can be seen, the Hough line detection algorithm is performed with cvHoughLines2. This gives a start and end point for every detected line. It is important to separate the detected lines from the field lines. A field line is a very thick line and thus there a a lot of thin lines detected within a single field line. To avoid this problem, before detecting the lines an edge detection algorithm is applied to the black and white image. The result are two parallel thin lines for every thick field line. The line detection algorithm now finds a lot less lines which is beneficial for further analysis. Experiments showed that it is no problem to get two parallel lines per field line.
To do something useful with the field lines, we had to detect edges and intersections of field lines and find their 2D coordinates. This is done by searching for intersections of the detected lines. The implemented algorithm checks for every possible pair of lines whether their angle differentiates for more than 15°. If so, it is checked if they have an intersection with linear algebra methods. If an intersection is found, a new point is generated. Generated points get stored

in a list of points. Due to the fact that on every field line edge/intersection there might be multiple detected line intersections, if a point is found but there already is a point nearby, the two points get merged into a single point. This results in a list which does not contain all found points, but multiple average points.
Experiments showed that this methods work quite well and only needs 0.5 seconds on average, what is much better than the goal recognition. Figure 12 shows

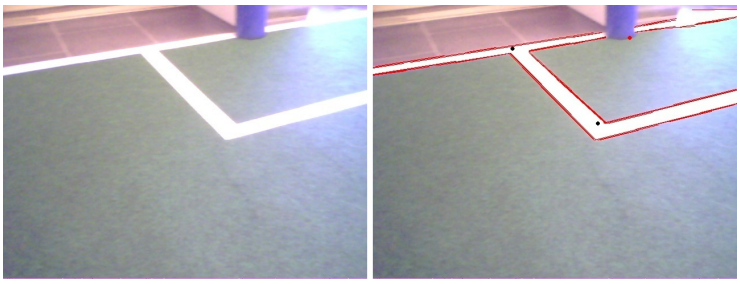

Figure 12: Result of the line detection algorithm

how a result of the line detection algorithm can look like. The left image is the input, the right is the output. The black points are field line edges/intersections, the red points are goal poles (see "Final Goal Detection"). The red lines are the lines which the algorithm detected.

4.6.11 Simplified Line Detection

During testing of the line intersection extraction it showed that the algorithm is very prone to noise. This means, that it happened a lot that line intersection where not recognized as desired, or false line intersections where recognized. For the particle filter this led to a bad performance and due to our way of implementation of the particle filter a much simpler field line recognition algorithm could be used in the final product. Which algorithm is used is completely dynamic and is decided when the particle filter is constructed which made comparing the two algorithms possible.
In the line intersection algorithm the results are point which represent the 2D coordinates in the camera image of the line intersections. The simple line detection algorithm just searches for every pixel which is white and assigns a point representing the 2D coordinates of the white pixel. This results in a long list of points which all lay on the lines. To reduce calculation time of the particle filter, not all points are kept. The points are reduced by their position which means that the camera image is split by a fine equal grid and if a part of that grid contains white pixels the middle of this part gets a point. The grid size is chosen such that each part of the grid has a size of three times three pixels.
As one can imagine this filed line detection algorithm is much faster, a drawback

is that the number of resulting points is higher, which results in a slower particle filter evaluation. In summary the faster algorithm and slower evaluation are about as fast as with the old algorithm, but they are a lot less prone to noise.

4.6.12 Searching and Focusing on an Item

Searching an item means moving the head around until the searched item is found. To do this, in the first iteration of the Locator Module where the Item is out of sight it gets decided whether to start looking left or right. If the item was last seen in the left half of the camera image the robot will start searching left and the other way around. Searching left means moving the head left a few degrees with every Locator Module iteration. The robot is not able to turn the head 360° so if a threshold is reached, the turning direction is changed. This process will never stop if the searched item is not in the robots field of view, so the robot has to turn for itself in certain situation. This process is part of the state machine and thus not discussed here. Do to the fact that the searched item may be out of sight of only one camera, it is useful to swap cameras during searching. This is done with every iteration so with every second iteration the same camera is used.
After an item is found, the camera should focus the item in the middle of the camera image to reduce the risk of losing it again if the robot moves. To do this the vertical and horizontal angles of the item relative to the middle of the camera image are calculated and if they are bigger then a threshold the robots head is moved to reduce the angles. It may happen that it would be beneficial to change the camera. An example is a ball which is focused with the upper camera and is rolling towards the robot. At a certain point the camera will be unable to follow the ball because the robots chin is touching his chest. The item focus algorithm realizes such events and changes the camera just before these events happen. To not lose the item out of sight when the camera is switched, the head is quickly moved to focus the item with the new camera.

4.7 Localization

The Localization is done in the ParticleFilter Module, which is part of the Locator Module. The ParticleFilter Module contains a particle filter for localization of the robot. It tries to determine the robots 2D coordinates and the robots rotation in the top-down view of the field. It is a completely independent Module which is only embedded in the Locator Module. This allows dynamic deleting and reconstructing of the Module, which is useful to reset the particle filter if the robot falls over or is penalized.

4.7.1 Using a Particle Filter for Localization

Particle filters [2] are a very useful tool for localization in knows environments, which we have for our soccer field. The basic idea is to have a top-down map of the area we are in and in that map we randomly distribute so called particles.

These particles are able to determine what the robots sensors would measure if the robot would be in that position. If a real measurement is available, all particles can compare the real measurement with what they would measure. The result is a value which is high for particles with a low difference of the measurement and the other way around. After every evaluation of the particles, evolution is applied. This means that particles with a low value will "die" and be respawn near particles with a higher value. After some evaluation clusters of particles will appear which represent the most likely positions of the robot.

4.7.2 Particle Filter Implementation

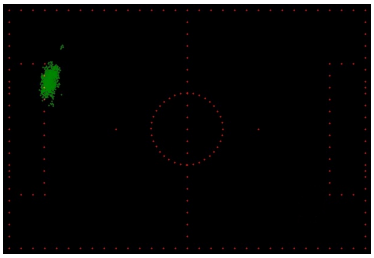

Figure 13: Debug output of the particle filter

In our implementation of the particle filter the only sensor input we use is what we get from the camera image. As described before, we get a list of points which represent either field line crossings or the field lines themselves. From the 2D camera image coordinates we have for each point we can compute the distance and angle relative to the robot with the already described algorithms. Also have the distance and angle to the goal poles if we see any, these will be used, too. Due to the rareness of the goal poles they get a higher weighting for the evaluation of the particles. To wrap up, we now have a list of distances and angles to points the camera sees.
To compare this with what the particles would see, we need a representation of the soccer field in a top-down view. Depending on which type of localization is used (line crossings or full lines) in the particle filters constructor a list of points is created which represent all possible points. The list contains 2D coordinates for every possible point when seen from above. An example can be seen in Figure 13.
When evaluating, for every particle it will be checked for every point we have on the soccer field, if the particle is able to see that point given the same viewing angle as the robots camera. If the point can be seen, the distance and angle from the particle to the point is calculated. The result is a list of points the particle can see. For every point in the list which the real robot camera provides, the point from the previously calculated list of visible points is searched, which has the lowest error sum when both distance and angle are compared. This results in an error for every point in the list which the real robot camera provides. The

inverted sum of these errors is used as the value for the current particle.
These value function leads to the following facts:

- If the robots camera does not see a point which actually is there but cannot be seen due to a bad camera image, the value of the particle will not get worse. This is positive because the robots camera misses some points from time to time.
- If the robot camera does see a point which actually is not there in reality due to noise, the particle value will get worse. This appears to be a bad thing, but the value will get worse for every particle so the particle filter is not influenced a lot through noise.

After every particle has been evaluated, evolution takes place. The evolution rule is very simple: Every particle which has a value lower than average will be put at the location of a particle which has a value above average. After being placed on the other particles location some random noise is added to the coordinates and the rotation value of the particle.
Of course the robot can move and thus, movement has to be simulated in the particle filter. For that purpose, wrapper functions for the robots movement functions have been added to the Locator Module. These wrapper functions call the actual movement function and additionally call a motion function with the same attributes within the particle filter. Within the particle filter, every particle will be moved like the robot would move given the submitted attributes. On top of the movement some noise for the coordinates and the rotation value is added to compensate the inaccurate motion model of the robot.
Even if the robot is not moved, every time before the particle evaluation is made, the particles are moved with all arguments set to zero. This will only add a bit of noise to the particles. This improves the average value of the particle filter because particles get the chance to eventually meet the robots actual position better. If the noise made their value worse, they are likely to "die" and respawn at a better location and angle.
Figure 13 shows a debug output of a possible result of the particle filter. This output was generated on the actual robot on the soccer field in Aachen, the particle filter was set to use the full field lines.

4.7.3 Usage of the Results

There are currently two implemented usages:

- Delivering the estimated 2D coordinates and rotation for telling it the other robot via the communication protocol.
- Delivering the probability of beeing in a certain part of the field for learning purposes.

For the first usage, the average position and rotation over all particles is computed. Of course, this is not the cleverest thing to do, because there might be

multiple clusters of possible positions which simply get ignored in the current setup. A better way would be to cluster the particles and provide multiple possible positions and a probability for every position. Due to limited project time this is not yet implemented.
For the second usage a grid is constructed over the top-down soccer field. The grid size in grids per axis can be freely defined at compile time. For every part of the grid the sum of values for the particles in that grid is calculated. All sums get normalized be the total sum of values, which results in probabilities between zero and one for every part of the grid. It may happen that the sum of probabilities does not equal one. This is possible because there is a small probability that the robot is outside the field and thus in none of the grid parts. One minus the sum of the grid probabilities equals the probability of being outside the field.

4.8 Statemachine

The Statemachine is implemented as a Markov Decision Process. However, because each state only has one action which is executed with 100% probability, it basically is just a simple State Machine, which can be extended to a MDP at any time.

4.8.1 Functions

The *MarkovDecisionProcess* class provides several functions. Most importantly, MarkovActions, MarkovStates and MarkovActionStateTransitions can be added to the MDP at any time. Here, it is possible to add Q-values and reward functions to a state-action pair, meaning that executing the action in the state gives a certain reward.
All defined actions inherit from the *MarkovAction* class. They implement a *executeAction* method, causing the program to enter this method when the action is performed.
Another important aspect are the *MarkovActionStateTransitions*. Here, a action-state pair is added to the MDP, together with a probability with which this action results in the state. In our program, this probability is always 1, which is why we basically have just a state machine.
After adding a couple of states and actions as well as their transitions to the MDP, the MDP is run by calling *nextState()* repeatedly, until the program is stopped. This method causes the actual state to choose the action with the highest reward - in our case it is always the same action, because each state only has one action. The *executeAction* functino of the chosen action is now executed, and the resulting state is calculated, using the provided probabilities.
The class also provides a *clearAll()* function, in order to reset the whole MDP. This is needed for switching between the field player and the goalie, who have two different statemachines.
Another important function is *setCurrentState(MarkovState* state)*. As the method name implies, this manually sets the current state of the machine, and

at the next call of *nextState()* the action of this new state is executed.

4.8.2 Configuring the State Machine

The State Machine is set up in the main module of the program. The procedure is as follows:
First, several MarkovStates are added to the MDP by calling

```
MarkovState *x = mdp->addMarkovState("ID");
```

mdp is the instance of the MarkovDecisionProcess object. An arbitrary number of states can be added to the state machine this way at any time. Of course, actions need to be defined as well. The actions are derived from the *MarkovAction* class. There should be only one movement step for each action. Things like *move forward and then move to the side until the angle to the ball is acceptable* should be avoided. The reason for this is, that at any time the robot can fall down or by penalized, and if that happens, the current action must be terminated, and depending on how many movement steps are defined in one action, this could take very long. Therefore, cancel conditions are defined for the state machine - in the case of our program these conditions are weather or not the *fallen* or *penalized* boolean in the *RobotStatus* class are set to true. Additionally, defining several movement steps in only one action results in the robot having a extremely static behavior which is harder to adapt.
After defining actions, they need to be related to a state in which they can be executed. So they are added to the MDP by calling

```
mdp->addQAction(someState,newAction,0,DEFAULT_Q,LEARNING_RATE);
```

. This means, that the action *newAction* is now an action of the state *someState*, with the given learning rate and Q-Value. The last thing that is needed now is to determine in which state the robot arrives after executing an action. The command

```
mdp->addMarkovActionStateTransition(newAction, otherState, 1.0);
```

takes care of that. After executing *newAction*, the robot arrives in *otherState* with a probability of 100%. If the need arises to extend the machine to a MDP, the probability is adjusted, and other possible states can be added to *newAction* this way.
One of the states must be set as initial state. After this is done, everytime the command

```
mdp->nextState()
```

is called, the current action is executed and the next state, determined by the MarkovActionStateTransition, is chosen. If a MDP is used, the action that gives the highest reward is chosen. If the robot falls over or gets penalized, the current action is terminated after the current motion being performed is executed and the state gets set to the according state.

4.8.3 Penalization and Falling

The *penalized* and the *fallen* state are special. They cannot be reached by performing a specific action, but external influence causes the robot to enter those stages. Therefore, once the *StatusMonitor* class detects a penalization (Chest Button press) or the robot pose changes to *Back*, *Belly*, *Left* or *Right* the *RobotStatus* class sets a penalized or a fallen variable on *true*. In each action, these variables are checked continuously. If they are *true*, the execution of the action stops. Now, in the main thread, it is again checked if one of these flags is set, and if yes, the state is forced to fallen or penalized by calling.

```
mdp->setCurrentState(penalized)
```

This happens, before the *nextState()* method is called. So, when now the *nextState()* method is called, the action of the penalized state is executed.

4.8.4 States and Actions

In section 4.2.6 the states and actions of the Statemachine were listed. Their relations are as follows:

Robot Role	Original State	Action	Resulting State
Both	Penalized	wait_Unpenal	Initial
Both	Fallen	standUpAct	Initial
Field Player	Initial	wait_Localize	seeBall_far1
Field Player	seeBall_far1	WalkAct1	seeBall_far2
Field Player	seeBall_far2	WalkAct2	seeBall_far3
Field Player	seeBall_far3	WalkAct3	seeBall_far4
Field Player	seeBall_far4	WalkAct4	atBall
Field Player	atBall	KickAction	Initial
Goalie	Initial	wait_Localize	goalieBadPos
Goalie	goalieGoodPos	stayPos	goalieBadPos
Goalie	goalieBadPos	adjustPost	goalieGoodPos

4.8.5 Tests in the Simulator

Both the behaviour of goalie and field player worked rather well in the Webots Simulator. Due to bad physics in the simulator, additional noise was unintentionally introduced, e.g. the robot would turn slightly to the side while walking. However, in most cases the field player would still find its way to the ball and be able to shoot the ball in goal direction.
A huge problem of the simulator is that if the system it is running on is not a very good one, it performs very poor and lags. The NAO will then sometimes think he has fallen down, even though he has not. In that case, our state machine will execute the standUp motion causing the NAO to really fall down. In the simulator he cannot get up anymore, because its feet will slip on the ground. For the following experiments, these occurrences have not been counted.

Generally, even if the robot gets himself in a good position and did everything correctly, there is still a chance that he misses the goal, because the ball position in front of the foot is a little off. It's a round foot kicking a round ball, so not hitting the optimal spot can already result in the ball drifting off.

The results of the tests are as follows.

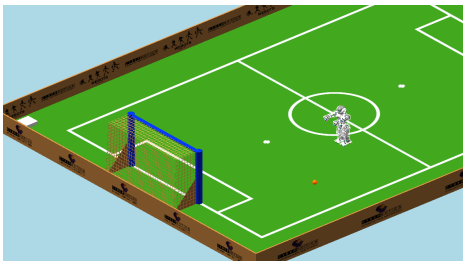

Figure 14: Webots Test 1

In 14 the NAO scores a goal in about 75% of the cases. The main reason for failing in this case is that the NAO, once he approached the ball will only recognize the left goal pole and adjust its position relative to the pole. This angle is sometimes not sufficient to score a goal, and the NAO will miss the goal by a few centimeters to the left.

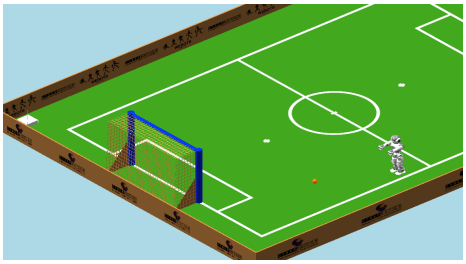

Figure 15: Webots Test 2

In 15 it is easier for the NAO to adjust its position because he does not have to adjust its angle to the goal much. In our tests, the NAO would always score a goal, as it can see both goal poles in this scenario, and hence aims for the middle of the goal.

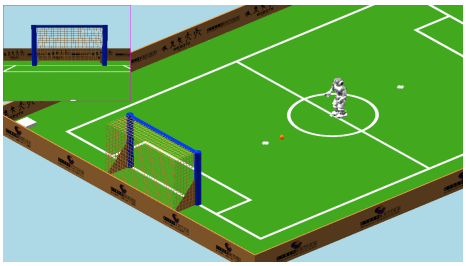

Figure 16: Webots Test 3

In 16 the easiest possible situation is depicted. The NAO is always successful in this scenario.

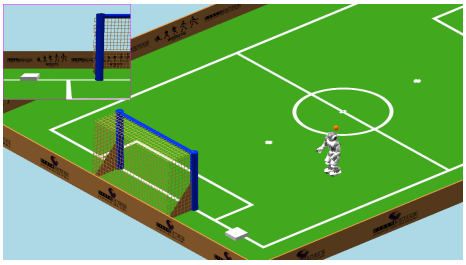

Figure 17: Webots Test 4

The scenario in 17 proved to be the biggest challenge for the NAO. Its task is to turn around while looking for the ball, then approach the ball. Then it has the goal in its back and therefore has to step in a circle around the ball until the goal becomes visible again. At first, this scenario had a success rate of zero, because the NAO would step against the ball while searching the goal.
The stepping around the ball algorithm is based on the distance to the ball, which is used as radius for the circle the NAO moves on. However, when the NAO looks for the goal, it obviously does not look at the ball anymore - so the distance to the ball changed due to inaccuracy of the stepping motion. This was unnoticed by the NAO. We approached that problem by having the NAO look at the ball after three steps, readjusting the radius. This improved the motion a little, but still the NAO would walk against the ball, probably due to inaccuracies of the angle calculation. As last resort, we implemented that the NAO takes a step backwards if the distance to the ball got shorter than 15 cm. Now, the NAO was able to move behind the ball and shoot to the goal. This had a success rate of around 50% of scoring a goal. The reason for this low rate is that in this scenario the ball is quite far away from the goal. Therefore, just a small difference in angle will cause the shot to miss.

Any tests for mirrored situations, where the NAO was at the other side of the goal showed similar results.

The goalie state machine performed very well in the simulator. It kept the ball in sight at all times, adjusting its position when the angle to the ball was dissatisfactory.
A problem was that the crouchDown Motion changes the angle of the NAO a little bit. The result was that the NAO would move to a good angle to the ball, crouch down, and then notice that the angle was not good anymore and get up again. To avoid this, we deliberately let the NAO step a little too far: 2 steps, when he moves to the right and 1 step, when he moves to the left. After this adjustment, the angle after crouching down was always acceptable.

4.8.6 Tests on the real NAO

Due to the problems with the camera described in section 4.6 the program did not perform well on a real NAO. While the ball recognition works quite well, the NAO still has problems recognizing the goal.
The switch to the new SDK caused problems with some motions. Identifying the problem and fixing it by re-recording them took away time on the field.
Due to the NAOs connecting with W-LAN, working got very slow when many groups were at the field at the same time. Often enough, a NAO would not be able to connect to the network at all, even if not many people were present.
All these issues made it quite hard to test the state machine accordingly.
What always worked in the end was that the NAO finds the ball and approaches it. It then goes through the algorithm and kicks the ball - but probably due to not recognizing the goal, it does not always shoot towards the goal. In situations like on picture 16 the NAO performs rather well. However, this is just a good looking example, because the goal is right in front of the NAO.
A rather big problem that sometimes occurred is that the calculation of the distance to the ball is off when it comes to distances lower than 20 centimeters. Therefore, when it comes to stepping close to the ball in order to prepare to shoot, we had to adjust our code in a way that if the distance gets lower than 20, the NAO guesses how far it still has to step. We set this guess to 3 steps of 2 centimeters. Of course, this leads to the NAO stepping against the ball from time to time. Since it is only a small step, the ball never rolls far. Often enough, it is still in reach for the NAO to kick it.
In order to solve this problem, a new vision aspect would have to be implemented, namely looking down, recognizing the own foot and place it correctly in front of the ball.

5 Conclusion

As a conclusion we can say, that we are very close to having the NAO perform well in a real environment. Current restrictions are:

- The Vision part, which suffers from the imperfect camera image the NAO delivers and inaccurate values delivered by the NAO SDK.
- Precise walking, which suffers from the NAOs Hardware and the floors friction. The NAO often slips over the floor instead of moving precise.
- Limited testing time on a real soccer field. Due to being forced to test in the simulator some actions of the robot may be overfitted to the simulator environment.
- Obstacle avoidance, which sometimes recognizes the NAOs own arm as obstacle. A solution would be to don't use arm movements to stabilize the walking, which would make walking even more imprecise.

Most of the restrictions do not appear in the simulator or were handled with workarounds which work in the simulator. Thus tests showed that all our algorithms perform well if the environment is fitting to the NAOs hardware.

In addition, we presented a generic, extensible way to implement Q-Learning in the NAO robot. We showed how this algorithm can be applied to specific tasks stressing the importance of carefully selecting features to minimize the number of states. These learning-enabled sub-tasks can be combined and integrated into the system without enormous performance losses. Moreover, we gave an example of how supervised learning can be made possible on the actual NAO robot by allowing a supervisor to give the NAO feedback on its actions.

References

[1] A. Ratter, B. Hengst, B. Hall, B. White, B. Vance, C. Sammut, D. Claridge, H. Nguyen, J. Ashar, and M. Pagnucco, "runswift team report 2010." Oct. 2010.

[2] S. T. et al., *Probabilistic Robotics*. The MIT press., 2005.