technische universität
dortmund

Final report of project group 580

Fußballspielende humanoide Roboter

Mark Breddemann, Sebastian Engels
Timo Etzold, Jan Gehlhaar
Till Hartmann, Stefan Kinzel
Lukas Pfahler, Stefan Rötner
Philipp Seifert, Piotr Szczotka

Friday 17th April, 2015

Advisor:
Prof. Dr.-Ing. Uwe Schwiegelshohn
Dr. Lars Hildebrand
M. Sc. Matthias Hofmann
Dipl.-Inf. Oliver Urbann

Technische Universität Dortmund
Institut für Roboterforschung
http://www.irf.tu-dortmund.de

Contents

Introduction

written by: Sebastian Engels

In the project group "soccer playing humanoid robots" the subjects science, technology, sports and entertainment are combined. Furthermore, there is the possibility to get a personal insight into the science of autonomous robotics. Due to the different tasks that are imposed on the project group, this gets an overview about the development of a larger project. These include, for example, the infrastructure, the communication within a team, as well as the analysis of problems as a team.

Various tournaments in the standard platform league are performed with the Nao robots by the company Aldebaran Robotics. Exclusively identical robots are used to play football against each other by all participants. The two biggest tournaments are the German Open and the international RoboCup.

As result of the standardized platform, the focus of the league is on the used Algorithms. The results of research come into practical use and can be directly exchanged with other participants. This allows a better insights into the topics and achieve better results.

The objectives of the project group are diverse. Above all, the previous run is to be improved, as well as new opportunities for modeling the behavior be developed. These issues are discussed below in more detail.

Walk Optimization

written by: Sebastian Engels

In the last years a robust walking movement was developed, which used the different sensors, e.g. accelerometer, gyrometer or foot pressure sensors to stabilize the Naos. However, the parameterization for a good walk is strongly dependent on the robot (wear rate, temperature of the engine, engines, accurate dynamic robot model) and from the flooring material (thickness of the carpet). Therefore, the robot must be calibrated regularly and the parameterization of the walk has to be adjusted. This procedure is very time consuming for 11 robots. Hence, it is the goal of this project group to develop an automatic parameterization/calibration, to determine automatically the optimized values for the robots and the flooring. During the optimization, the robot should move as during a normal game on the field. Thus, the robot should accelerate or decelerate and run in different directions. This is to ensure, that the values are not only valid for certain velocities or movement directions. Therefore, it is necessary to develop an algorithm which reviewed at any time, whether the walk of the robot has improved and reacts accordingly.

Learning Behaviour

written by: Stefan Kinzel

The behaviour of the Nao robots is currently described in the form of deterministic, hierarchic automatons. These automatons are described in a language called CABSL, which is a C++-based replacement for the former used language XABSL. One goal of this project group is to find a better way to determine the symbols used in CABSL, using learning methods. Other approaches, like Extended Behaviour Networks (EBN) or revisable knowledge bases, based on Conditional Knowledge Representation (OCF), should be implemented as a more dynamic replacement of the current (static) CABSL-automatons. For evaluation of these new implementations, a valid test environment is needed. A simulator, able to simulate test games, could be an adequate test bed, so another goal of this project group is to adapt the existing simulation framework, being able to simulate test games with different behaviour implementations. If possible, these simulations should be executed on a computer cluster.

In future there will be the possibility of a coaching-robot, which is placed at the border of the field. It can be used to get an overview over the game situation. This project group tries to elaborate some ideas for using this robot.

Another consideration is the participation in the simulation league. Here the behaviour part of the framework becomes the only crucial part. A participation in this league should be taken into account.

Structure of the report

written by: Sebastian Engels

This report can be divided into two main parts. First, in part I the motion control is introduced. For this the walking parameters and the logging of sensor values are described in chapter 1. Subsequently, different evaluation methods and the framework for parameter optimization are presented. After this, in chapter **??** two criteria of stability are shown. The first one uses the x-acceleration sensor to determine an upper limit for the velocity of the robot. The second criterion uses the foot pressure sensors to detect unstable runs. Furthermore, a proper strong kick, the so called *LongKick* is presented in chapter 3.

In the second main part II the behavior control of the robot is discused. First, a possibility of modeling the world states and classification of game situations using fuzzy logic is introduced in chapter 5. Subsequently in chapter 6 a software framework for developing artificial intelligences that participate in the rcss2d soccer simulation league is presented. Another approach for modeling agent behavior, the extended behavior networks, are discused in chapter 7. At the end of the behavior-part a new approach for the path-planning is shown in chapter 8.

Finally the results of the project group are summarized in chapter III.

Part I

Motion

Chapter 1

Evaluation and Improvement of the Walking Engine

1.1 Objectives

written by: Stefan Kinzel

The stable walk of a *Nao* robot depends on several factors. The already developed walking engine uses various sensors of the robot and tries to generate a movement for a stable walk. In the parametrization of this movement, both the properties of the robot, such as the wear-out of the joints, manufacturing tolerances, and even dynamic factors such as the temperature of the motor and gear, as well as the nature of the substrate play a role. The goal of the walk optimization is to find a method by which the walk parameters can be adapted to the current situation. The robot should be able to evaluate his walk independently and on this basis generate a better set of walk parameters. So in the end each *Nao* robot finds its own set of parameters, adequate to its specific condition. In best case, this leads to a higher walking speed of the whole team, less drops and a more stable, reliable, walk.

1.1.1 Walking Parameters

To parametrize the walk there are 77 walking engine parameters. Some of these parameters are no longer used or have (almost) no influence on the walk. To test the optimization, various parameters have been adjusted, which have an obvious influence on the movement of the robot during the walk. One thing that has to be determined later is which of these parameters should automatically be adapted by the walk optimization procedure.

Step height

With the parameter *stepHeight* the maximum distance between the sole of the foot and the ground during a step can be set. The default value is 0.02, which corresponds to a step height of 2 cm. During our tests we set a step height between 0.05 (5 cm) and 0.005 (0.5

cm), which had a clearly visible influence on the quality of the walk. Using a step height of 0.01 and lower causes the robot to walk slower and slightly more unstable. With a step height bigger than 0.02, the walk became more and more unstable, probably caused by faster joint movements in the Z-direction. This controllable instability was used to verify the evaluation criteria.

Step duration

The time used for one footstep can be influenced by setting the parameter *stepDuration*. In the default settings, a step duration of 0.5 (0.5 s) is set, so the *Nao* walks with two steps per second (2 hz). With a step duration of more than the pre-set 0.5 s the stability of the walk decreases, caused by a longer period, where the robot only stands on one foot. A shorter duration also caused an increase of instability, so 0.5 s seems to be an (almost) optimal setting. The step duration has to be considered during the evaluation of the sensor values, especially during the interpretation of the frequency spectrum.

ArmFactorLeft, ArmFactorRight

The *Nao* robot uses its arms to balance acceleration forces occurring during the walk. With the parameters *armFactorLeft* and *armFactorRight* the degree of the movement can be set. The default factor is 0.005 for both sides. Setting this value to a much higher or lower value (e.g. 0.5 or 0.0001) has a visible influence on the movement of the arms, but no remarkable influence on the stability of the walk.

1.1.2 Sensors

The *Nao* robot has several built in sensors [1], which can be used to evaluate and stabilize the walk. The most important sensors are the accelerometer, which measures the acceleration in all three directions, the gyrometer, which measures the angular speed, as well as the force sensitive resistors in the soles of the foots, measuring the gravity forces. For the walk evaluation, we logged these sensors values, including the calculated Centre of Pressure and weight, to a file, using the CSV-Logger of the framework. The result is a CSV-file, containing the values with a frequency of 100 hz.

Accelerometer

The accelerometer of the *Nao* are capable of measuring the acceleration in all three directions [2]. Aldebaran Robotics states an accuracy of 1 % (probably based on the maximum value, about 2G), so these sensors (in theory) are good for evaluating short time movements. Long time measurements cause the error in measurement to sum up, so the values are not exact enough any more. In practice, these sensors show an offset (error in measurement) and a variable portion of the gravitational force, which has to be removed from recorded values on evaluation. To remove the gravitation forces from the sensor data, the

exact orientation of the sensor has to be calculated. As a basis for this calculation, the angles of the joints can be used.

Gyrometer

The gyrometers are able to measure the angular speed of the robot in all three axes [2]. For the gyrometer, Aldebaran Robotics state an accuracy of 5 % (probably based on the maximum value, about 500 degrees per second), which is quite big. This inaccuracy makes the sensor quite useless for evaluating small and short time movements.

Force Sensitive Resistors

In the sole of each foot there are 4 force sensitive resistors (FSR) [3]. These FSR work like the keys in some older keyboards: The more pressure is exerted, the lower the resistor value gets. All 8 resistors are capable of measuring a force of 25 N. The current walking engine uses these resistors to calculate both the current weight on each foot and the centre of pressure (CoP).

1.2 Recording and analyzing of the sensor data

written by: Sebastian Engels

To develop an algorithm for an automatic improvement of the walk, it is necessary to evaluate the running performance of the robot. A pure visual assessment can not be done here, since the differences usually are very low and are not necessarily perceived. In addition, the robots should independently assess their movement behavior. Therefore, it was decided to a review based on sensor data. To assess the walk the *Nao* can use four pressure sensors per foot, two gyro sensors and two acceleration sensors. The gyro and acceleration sensors measure the lateral and forward tilt/acceleration. For the development of evaluation or stability criteria of the run, the sensor values of the robot were logged while running in advance. During the running the current sensor values will be saved with a frequency of 100Hz.

In order to have enough data available for the development of an algorithm, several runs with different robots and floorings and velocities were logged. Unfortunately, the sensor values are sometimes very flawed, so, depending on the stability criterion, a subsequent filtering / editing is necessary.

1.3 Walk Evaluation

written by: Stefan Kinzel

1.3.1 Approach

The optimization of the walking parameters can be divided into two parts. First the current walk has to be evaluated. In order to evaluate the current walk, there has to be a quality criterion, which can be easily calculated by the robot itself. Using this criterion, one can implement an optimization algorithm, using a gradient descent for example.

1.3.2 Statistic methods

In order to get some initial data for the walk evaluation, we create log files, containing the values of all sensors, that might be interesting for walk evaluation: the gyrometer, accelerometer, FSRs, the calculated CoP coordinates and the calculated weight. These log files showed the values, recorded during walks for approx. 25 seconds with different speed. Walking with average speed causes the robot to walk more stable, so we wanted to compare slow and very fast walks. We also repeated this recording on several robots (old and new ones), so we can ensure not just finding an anomaly of one robots sensors.

The first idea for evaluating these data was to use simple statistic methods, such as mean values and variances of the sensor values. There was no visible difference between slow and fast walks or older and newer robots.

The next idea was to compare different sensor values in order to find a correlation between them. A calculated coefficient of correlation showed no difference between the walks. With plotting the values we found an anomaly: As visible in figure 1.1 there appeared three cluster when plotting the x and y value of the accelerometer, that almost disappeared in a faster walk (figure 1.2). We found no algorithm that was able to evaluate this automatically. The k-means-algorithm, for example, was able to find these clusters, but the distribution showed no remarkable difference between slow and fast walks.

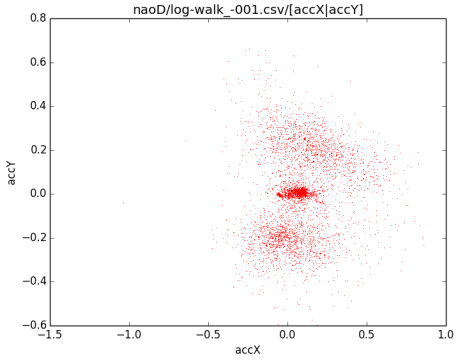


Figure 1.1: walking with 0.1 cm per second

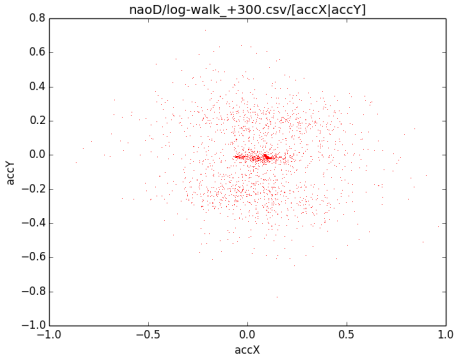


Figure 1.2: walking with 30 cm per second

Fourier Transformation

Our next idea was to use a (fast) Fourier Transformation (FFT) [4] and so take a look on the frequency spectrum. The expectation was to find an disturbing vibration, which

causes the *Nao* to fall down. As visible in figure 1.3, there was a remarkable vibration at two hertz. But this vibration seems to be caused by the walk itself, as the robot walks with a rate of two steps per second. We then implemented a two hertz bandstop filter, which is able to eliminate this vibration in sensor values (as can be seen in figure 1.4). The next vibration with a remarkable amplitude can be found at 4 hertz. A visible observation showed, that vibrations with higher frequencies do not affect the stability of the walk, so we made no further attempt for eliminating these vibrations. Nonetheless this can be an option for further experiments.

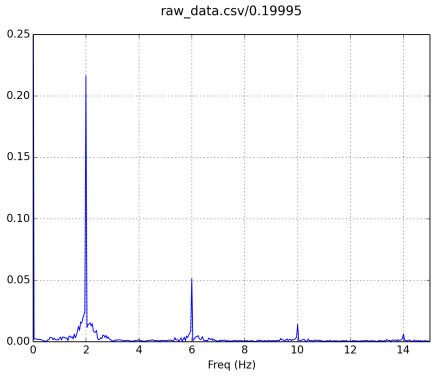


Figure 1.3: frequency spectrum of the acceleration values (x-direction)

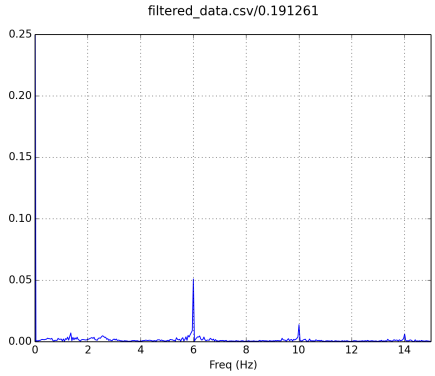


Figure 1.4: same values as in Figure 1.3, now filtered with a 2 hertz bandstop filter

Principal Component Analysis

The Principal Component Analysis (PCA) is a statistic method, which is able to determine correlations between sensor values. As a last attempt we applied the PCA to the recorded sensor values, using the tool RapidMiner, but with a disappointing result. We were not able to find other correlations, so this was the last attempt using simple statistical methods. Obviously none of these simple methods are useful to evaluate the walk, so we did no further experiments here.

1.4 Framework for walk parameter optimization

written by: Mark Breddemann

A goal was the optimization of walk parameters using a simple criterion. For this, the *Nao* should walk for a few seconds after pressing onto the chest button and record all sensor data values while walking. Afterwards, these recorded data should be analyzed and the past walk is assessed. Then the walk parameters are optimized using the gradient method based on the determined criterion. For this, the assessment of the current test-walk is compared with the one of the previous walk, to determine the direction of the optimization. After several runs, the walk paramerter value at which the *Nao*s walk is best should be found.

It was considered important to modularize the optimization process, because it is likely that there exists several assessment criteria and there are many optimizable parameters.

It is, that some parameters are optimized using different criteria or methods than other parameters. Also, some parameter optimization may need some preprocessing of the input data.

1.4.1 Integration into the *Nao* framework

For the connection between the existing *Nao* software framework and the new code for the walk parameter optimization, a *WalkCalibrator* module was created. It integrates into the *Nao* framework by providing the representation *MotionRequest* to control the test walk and the representation *WalkingEngineParams* to modify walk parameters while the framework is running. Also, the *WalkCalibrator* uses the module *KeyStates* for monitoring the *Nao*'s chest button and the module *SensorData* for collecting the sensor data.

Module *WalkingEngineParams*

The walking parameters are saved to the file *walkingParams.cfg*, just like the original provider of the *WalkingEngineParams*, so that the new implementation of this provider is compatible to the old one. Additionally, the optimization framework creates a backup of the walk parameters file between every step of the optimization, so interim results are not lost and the whole optimization process can be traced afterwards.

Module *MotionRequest*

The original *MotionRequest* provider was replaced during development, to create a simple way for testing the implemented optimization: Currently, the test is triggered by the chest button of the Nao. After pressing, this module causes the *Nao* to walk straight forward, until a specific amount of sensor values are collected from the *SensorData* module. Normally, 2500 sensor values are collected. At a framework frequency of 100Hz, this means that the *Nao* walks for 25 seconds.

After developing the optimization framework, the original *MotionRequest* provider has to be reintegrated. It is not acceptable to let the *Nao* simply walk forward for optimization during a game. Instead, the sensor data needs to be collected by another way during normal operation. For example, the collecting could be triggered when the *Nao* walks to a specific point of the field, while the collecting stops when the *Nao* shoots the ball.

1.4.2 Modules of the optimization framework

To improve extensibility and reusability of the written code, the optimization framework was divided into different modules. Due to the modularization it is possible to write a new optimization module for a specific parameter with few lines of code, provided that only existing criteria and preprocessing methods are used.

***WalkCalibrator*-Module**

The *WalkCalibrator* Module integrates the optimization framework into the *Nao* framework. It provides the *update* methods for the *WalkingEngineParams* and the *MotionRequest* modules and implements the functionality described in chapter 1.4.1 (collecting sensor data and walking straight forward). Implemented optimization modules (C++ classes) need to be registered in this main module. All registered optimization modules are called after a test run.

SimpleGradientMethod

This module implements a gradient method for optimizing a value [5]. When creating the module it is defined, whether a value should be minimized or maximized. Also, the initial step length for optimization is defined. The C++ implementation has a method called *getNewValue* which uses the old value of the walk parameter and the value of the walk criterion. Based on the history of the method calls and its parameters, a new value of the walk parameter is returned.

This module can be used by the walk parameter optimization modules to adjust and optimize the parameter. By modularization the actual optimization method, it can be reused for several walk parameters. Also, it is easy to implement further methods for finding the optimal value for a parameter in the future.

***BaseCalibrationModule*-Module**

This class offers methods for the processing of recorded sensor data, that can be further used by the optimization classes. These methods work with the data of the sensors and can be divided into categories: Preprocessing and criteria for the walk evaluation.
The implemented methods for the evaluation of the walk are:

- mean value
- quadratic mean value
- calculation of the average length of a sequence[1]

As for the preprocessing, the following methods were implemented:[2]:

- addition of all sensor values
- complementary filters
- variable low pass filter

[1] More in chapter 1.4.3
[2] Further information an the development of this methods can be found in chapter 1.4.2.

- more general, variable first order filters
- 2 hertz bandstop filter

Optimization Classes

With the help of the *BaseCalibrationModule*-Class' functionality and a pre-built gradient method, it is now simple to create new classes for optimization. Currently the following classes are implemented:

- ArmMovementCalibrator
- FootPitchCalibrator
- StepDurationCalibrator
- StepHeightCalibrator

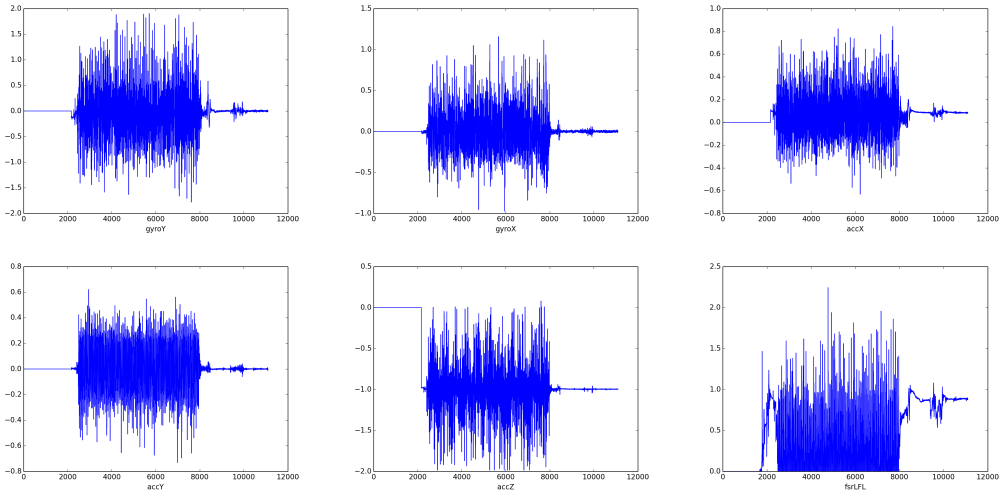


Figure 1.5: Recorded sensor data with parameters armFactorLeft=0.1, armFactorRight=0.1

The *ArmMovementCalibrator* was the first optimization class to be implemented. It turned out, that this had very little effect on the quality of the walk. Visually there was no observable difference for a walk and with the sensor data and the current evaluation criteria[3] a distinction between the different walks was not possible. The recorded sensor data with two different values of *armFactorLeft* and *armFactorRight* can be seen in figures 1.5 and 1.6. Therefore, until a proper criterion is found, an optimization cannot be conducted.

In order to find better evaluation criterion, parameters were written for the other optimizer modules, whose (strong) variations have a strong and obvious impact on the walk of a *Nao*. With this, new evaluation methods can be directly tested and different walks can be categorized. Simple criteria as the variance of the acceleration in the x-direction and

[3]Variance/Quadratic mean value of the acceleration sensor for the x-axis and the gyro-sensor in y-direction

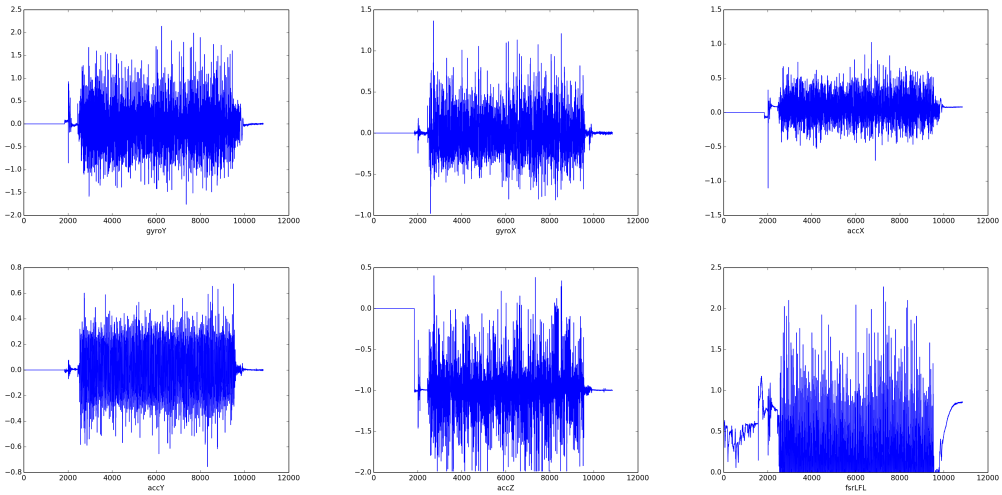


Figure 1.6: Recorded sensor data with parameters armFactorLeft=0.9, armFactorRight=0.9

of the gyro sensor in y-direction already allow for a rough classification of a walk into the categories "good" and "(very) bad".

In this context different approaches were tested and quickly dismissed again, because they were obviously not suited to determine whether a test walk of the *Nao* is stable and flowing or unstable and choppy. These were amongst others:

- Upper Body Angle: The idea behind this was to observe the variance of the upper body's angle. First a complementary filter was written, to calculate the angle from the data of the acceleration and gyro sensors. However, there exists already an implementation of the Kalman filter, that offers the same functionality. Unfortunately the variance of the resulting values correlate only little with the quality of the walk.

- 2 hertz bandstop: Especially for the acceleration sensor data, oscillations of 2 hertz can be recognized, resulting from the default settings for the step duration to 500ms. The idea was, to filter out this (wanted) oscillations and consider the resulting data. This filter was applied to the data from the acceleration for the x-axis and from the gyro sensor in y-direction. Finally, the variance of the values was calculated again. However, the evaluation criterion did not improve as to the previously tested criterion without the two hertz bandstop.

1.4.3 Calculation of the Average Sequence Length

A suitable criterion the average sequence length of the X-acceleration sensor. Briefly summarized, this criterion analyzes the duration of a forward and backward oscillation of the upper body and the velocity during swinging. These two values are combined, while a longer duration and a higher velocity is rated more bad than a walk with short durations and low velocities.

Procedure of the sequence criterion

The following pseudo-code describes the calculation of the sequence criterion:

```
current_sequence_sum = 0
current_sequence_sign = POSITIVE
all_sequence_sums = LIST(empty)

// note that sensorvalues contains all
// sensor values in a chronological order

FOR value IN sensorvalues {
        value_sign = (value > 0) ? POSITIVE : NEGATIVE

        IF (value_sign != current_sequence_sign) {
                // new sequence found! save current
                // sum in list and reset current sequence
                all_sequence_sums.add(current_sequence_sum)
                current_sequence_sum = 0
                current_sequence_sign = value_sign
        }

        // sum up square sum
        current_sequence_sum += value^2
}
```

1. At first, the sensor data is split up into sequences. The boundaries of the sequences are located between two sensor values, whereas one value has a positive value and the other a negative one.
2. Taking the square value of every data value, so that there are no more negative values.
3. Summation of the squared values of every sequence. Note that every sequence has its own sum.
4. Calculation of the mean value of all sequence sums.

The calculated mean value in step four is used as the output for this criterion. When using the X-acceleration sensor, this value varies from 30 which indicates a very good walk to over 1000, when the *Nao* is nearly falling.

Origination of the sequence criterion

The sequence criterion is originated from a lot of testing and varying the process steps. The first idea was to examine the length (only the number of values, not the values itself) of the *longest* sequence. This approach proved to be useful, as it differs between a very good

and very bad walk. For further improvement, the velocity while oscillating was included. For this, the absolute values of a sequence were summed up and the *topmost* sequence sum was used as the output value. This expansion of the process appeared useful for the evaluation of the walk.

However, this criterion was susceptible for single outliers which occurred in some tests, although the walk was good. Therefore, the criterion was extended by the mean value of the sums. To uprate 'bad' sequences, the values are squared before they are summed up. This lead to the sequence criterion described above.

Prefiltering

While testing, it turned out that it would be useful to apply some prefiltering methods for further improvement of the sequence criterion.

Partially, the measured X-acceleration is caused by gravity. This is because the *Nao* does not walk with an upright upper body, but it is inclined slightly forward. This gravity-caused part must be eliminated from the error value. Theoretically the error can be calculated by the upper body angle set in the walk parameters. Due to the wearout of especially older *Nao* robots, the desired angle is different from the real body angle of the robot.

Therefore, the mean value is calculated from the sensor data. Using a longer period, the mean value of the recorded X-acceleration values approximates to the error value caused by gravity. So this mean value is subtracted from each sensor value before computing the sequence criterion.

In addition, it was determined that the criterion decides better if the X-acceleration values are filtered with a lowpass.

Problems of the sequence criterion

Unfortunately, this criterion is not sufficiently precise for walk parameter optimization. Quite unstable walking is reliably detected, but differentiation between a subjectively relatively good and a very good walk is not possible. Also, it was observed that if the criterion made a good review of the walk, the *Nao* may not tend fall down, but it is not confident, that the *Nao* actually did a good walk. For example, if the step height was set to a very low value, the *Nao* only jiggled a bit, but it did not walk. This jiggling had a good evaluation by the criterion. Therefore, this sequence calculation is more an assessment of the stability of the *Nao* not the walk quality.

1.5 Clipping

written by: Stefan Kinzel

The idea of clipping is to decrease the maximum walking speed if the sensor data indicate that the *Nao* tends to be unstable. There is an implementation in the framework, which

depends on the absolute values of the gyrometer sensor. Our idea was to improve this clipping implementation, using the sequence criterion instead of the gyrometer values. While this criterion can not determine whether a walk was good or very good, it is at least able to distinguish good from bad runs, which is sufficient for using it in a clipping implementation.

A problem with the original implementation is that it depends on recorded sensor data of a 25 s walk, but for the use in the clipping process, it must react more quickly. Therefore, this method has to be rewritten to an online algorithm. The clipping algorithm sums up the current (and squared) sensor values, waiting for a change of the sign, which marks the end of the current sequence. With comparing the value of the sequence to an (adjustable) threshold, separating good and bad walk periods, an unstable walk period can be detected. As a reaction, the clipping implementation now decreases the maximum walk speed by a value also depending on the sequence value.

As seen in experiments, newer *Nao* robots achieved a higher average speed (caused by less clipping interventions) than older robots. Based on this observation, there could be implemented a long-term clipping, reducing the robots walk speed to an optimal speed for the specific robot. This could lead to fewer interventions of the clipping and all in all to a more stable walk, as the clipping is only able to react to bad walk periods.

Chapter 2

Criteria of Stability

written by: Timo Etzold

Sensor values were logged and examined in order to improve the walking of the *Nao*. Two of these are described in the following sections: the x-acceleration (2.1) and the center of pressure (2.2).

2.1 X-Acceleration

written by: Timo Etzold

While having a look at the x-acceleration sensor data, it stuck out that the x-acceleration data values decrease if the *Nao* falls. Figure 2.1 shows the x-acceleration sensor data plotted over time, which were logged over 5000 frames, which is 50 seconds. The sensor data jump from positive to negative, because the acceleration is positive in the first half of a step and negative in the second half. While logging the robot fell three times. This

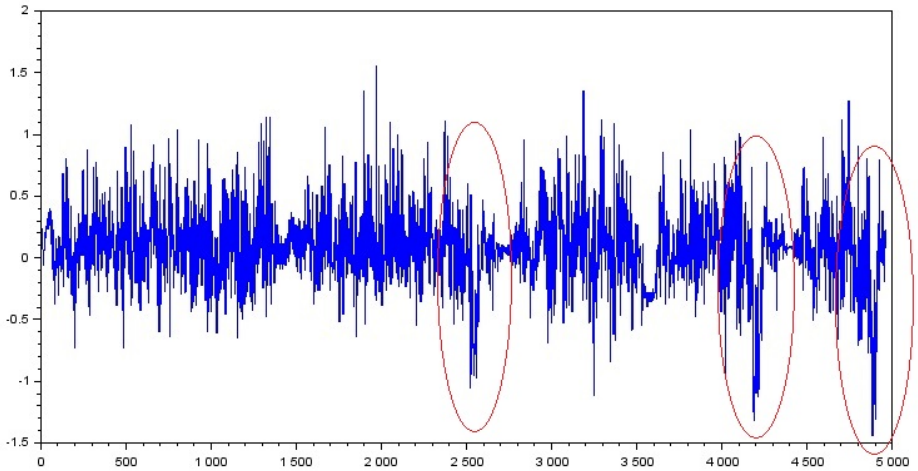


Figure 2.1: Plotted x-acceleration

is noticeable in the collected data. The first time the *Nao* fell was at about the $2500th$ frame, the second was at about the $4200th$ frame and the last fall was at about the $4900th$ frame.

The problem is to create a term from those data, which describes the quality of the walk. Because the data deviate from the regular data when the walk is poor, these deviations can be used to determine whether the walk is good or not while walking. But the values are very noisy, because the acceleration can deviate in just one step (2 steps per second) and can be regular in the next one without making the walk bad. The arithmetic mean (2.1) and the variance (2.2) is used to compensate the noisiness.

The arithmetic mean of a group of n data is calculated by [6]:

$$\bar{x}_{arithm} = \frac{1}{n} \cdot \sum_{i=1}^{n} x_i, \tag{2.1}$$

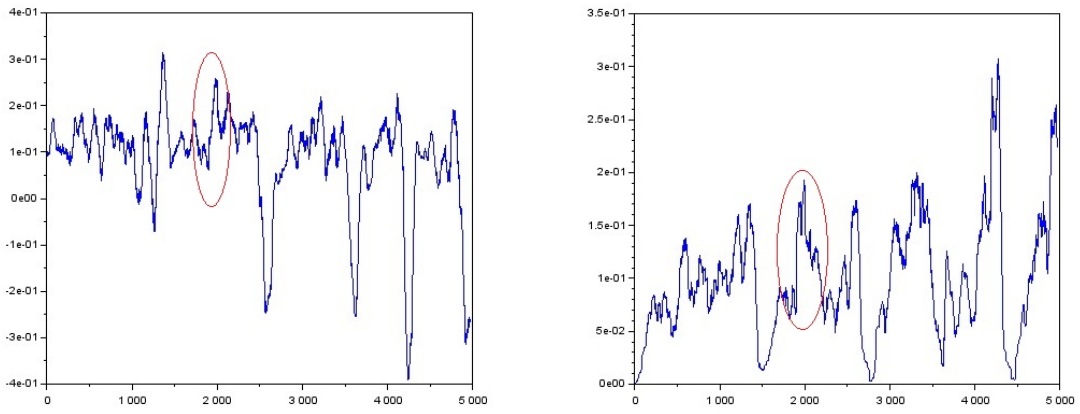
where x_i is the i-th logged sensor value. The variance of a group of n data is calculated by the term [6]:

$$s^2 = \frac{1}{n-1} \cdot \sum_{i=1}^{n} (x_i - \bar{x})^2, \tag{2.2}$$

where $\bar{x}$ is the arithmetic mean and x_i is the i-th logged sensor value.

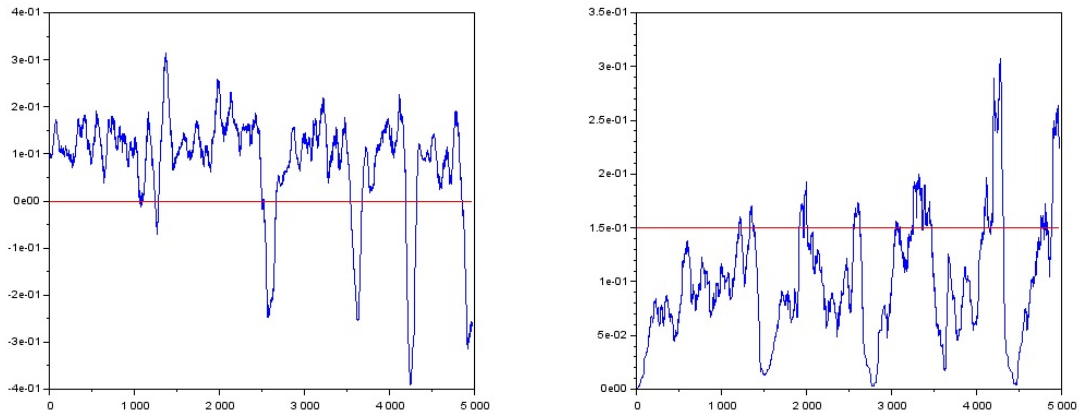
Arithmetic mean and variance are calculated in each update (100 per second) to detect the situations in which the walk is working bad. But not all sensor values from the beginning of the walk are used to determine these situations. This is done, because, if all data were used, the walk would be classified as bad too late, e.g. when the *Nao* is already falling. So the moving arithmetic mean over the last n values at time t is calculated by (modified according to [6]):

$$\bar{x}_{arithm}(t, n) = \frac{1}{n} \cdot \sum_{i=t-(n-1)}^{t} x_i, \tag{2.3}$$



(a) arithmetic mean **(b)** variance

Figure 2.2: Moving arithmetic mean and variance of the last 100 sensor values



(a) arithmetic mean; threshold 0

(b) variance; threshold 0 15

Figure 2.3: Moving arithmetic mean and variance of the last 100 sensor values with drawn in threshold value

where x_i is the i-th logged sensor value. The following term is used to calculate the moving variance over the last n values at time t (modified according to [6]):

$$s^2(t,n) = \frac{1}{n-1} \cdot \sum_{i=t-(n-1)}^{t} (x_i - \bar{x})^2, \tag{2.4}$$

where $\bar{x}$ is the arithmetic mean and x_i is the i-th logged sensor value.

Figure 2.2 shows the development of moving arithmetic means (equation 2.3 in subfigure 2.2a) and variances (equation 2.4 in subfigure 2.2b). In the case of the arithmetic mean it can be seen, that the moments of a bad walk can be differentiated from normal moments. These moments can be detected using the variance, too. In addition to this, stumbling can be seen in the variance, these are the moments in which the variance is also very high. One example of this are the marked values in the figures. The stumbling cannot be detected by only having a look at the arithmetic mean, but by having a look at the variance too the stumbling can be detected.

By testing some values for the determination of a bad walk it turned out that using one threshold for each of variance and arithmetic mean is sufficient. In the case of the arithmetic mean this is $threshold_{mean} = 0$ and in the case of the variance it is $threshold_{variance} = 0\ 15$. Figure 2.3 shows those thresholds drawn in the plotted data, which are shown in figure 2.2, as a straight line. If the arithmetic mean falls below $threshold_{mean}$ or if the variance exceeds $threshold_{variance}$ the walk has to be adjusted immediately, otherwise the *Nao* threatens to fall.

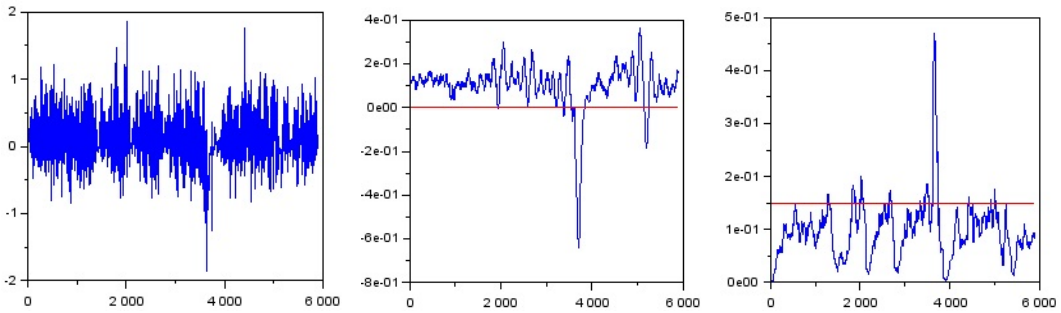
To adjust the walk the following approaches were tested:

- decreasing step height
- decreasing speed

- decreasing step height and speed

Reducing step height

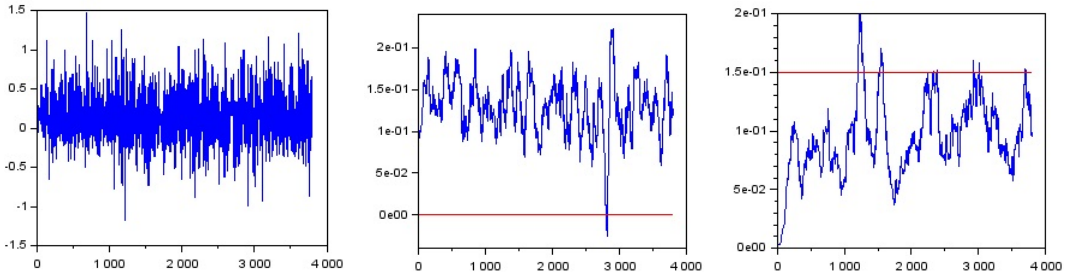
The approach of decreasing the step height is based on the idea that low step heights increase the stability of the walk. However, it appeared in experiments that the step height must not be lower than 0 1m, because otherwise the feet will drag across the floor which results in an unstable walk. Figure 2.4 shows the sensor data which occur when the step height is decreased if arithmetic mean or variance falls below the threshold value or exceeds it respectively. It can be seen that decreasing the step height has no significant influence on adjusting the walk, because an irregularity occurs around the $4000th$ frame, which is not corrected immediately. So because of this, decreasing the step height alone is insufficient for adjusting the walk.



(a) sensor values **(b)** arithmetic mean; threshold 0 **(c)** variance; threshold 0 15

Figure 2.4: Sensor values, moving arithmetic mean and variance of the last 100 sensor values with drawn in thresholds adjusting the walk by decreasing step height

Reducing speed



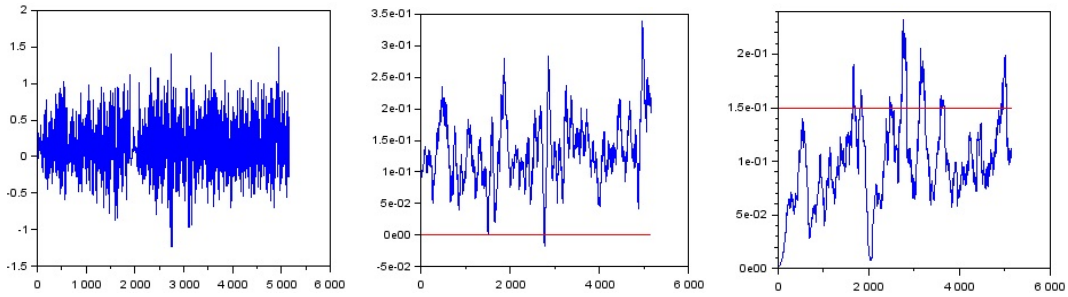
(a) sensor values **(b)** arithmetic mean; threshold 0 **(c)** variance; threshold 0 15

Figure 2.5: Sensor values, moving arithmetic mean and variance of the last 100 sensor values with drawn in thresholds adjusting the walk by decreasing speed

Reducing the speed does result in a stable walk, because the steps are smaller. Figure 2.5 shows the sensor data which were logged while decreasing the speed if the threshold values are exceeded or fallen below. The original sensor values have fewer peaks, which can be seen in figure 2.5a. The walk is also adjusted very fast when exceeding or falling below the threshold values. This can be seen in figures 2.5b and 2.5c. So decreasing the speed is a reasonable approach in making the walk more stable using the x-acceleration.

Reducing step height and speed

Decreasing the step height in combination with lower speed was also tested. Figure 2.6 shows the sensor values, moving arithmetic mean and variance which are logged while decreasing step height and speed if mean and variance are above or below their thresholds. In comparison to just decreasing the speed, no big difference can be found (see figure 2.5). Thus combining the decrease of step height and speed is not a sufficient approach to adjust the walk.



(a) sensor values **(b)** arithmetic mean; threshold 0 **(c)** variance; threshold 0 15

Figure 2.6: Sensor values, moving arithmetic mean and variance of the last 100 sensor values with drawn in thresholds adjusting the walk by decreasing step height and speed

Conclusion

As shown in the previous paragraphs, using the x-acceleration to increase the stability does make sense and works. But only decreasing the speed has a significant effect on the stability. So the x-acceleration as a stability criterion can be used in the walk optimization by implementing it into the clipping.

Progress

The arithmetic mean does not align around the same value on every robot. Since the instable moments of the walk can also be detected using the variation, the arithmetic mean will no longer be used. Instead, the body angle of the *Nao* is used to detect bad working walk. Because the *Nao* will most likely fall by having an angle greater than 0 25, the used threshold value is 0 15. Also, the threshold value of the variance has been tweaked to 0 12.

The speed change will be executed when there is a specific number of continuous frames, where the threshold is exceeded or respectively not exceeded. This is done because increasing the speed every frame where the walk is classified as stable and decreasing the speed every frame where the walk is unstable would result in permanent and too great speed changes and therefore an unstable walk. For the increase of the speed this value is 250 frames and for decreasing it is 100 frames. The difference is made, because when the walk is unstable for 100 frames (1 second) it has to be improved immediately. On the other side, increasing the speed every second would be too fast as it is not known if the higher speed results in an unstable walk.

If the angle threshold by contrast is exceeded, the speed has to be reduced immediately as with even higher values the *Nao* will fall. This is done once for every exceeding. Also,

there is a cool down of 150 frames to prevent the *Nao* from stopping when there are many peaks which exceed the threshold for short amounts of time.

The value by which the speed will be reduced when exceeding the angle is hand-tuned as 30 as this value has proven to be good in several tests. The value by which the speed is changed by having continuous stable frames is calculated as follows:

$$(varborder - \overline{accXvar}) * fPos, \tag{2.5}$$

where $varborder$ is the variance border (0 12), $\overline{accXvar}$ is the average x-acceleration variance of the last 5 frames and $fPos$ is defined as 5. The average of the last 5 frames is used to compensate the peaks, because if the speed is reduced when the current frame has the peak of the variance the speed would be reduced too much. If there are by contrast continuous unstable frames and the speed has to be reduced the value is calculated by:

$$(varborder - \overline{accXvar}) * fNeg, \tag{2.6}$$

where $varborder$ is the variance border (0 12), $\overline{accXvar}$ is the average x-acceleration variance of the last 5 frames and $fNeg$ is defined as 20. The factors $fPos$ and $fNeg$ are used to make it harder to speed up for the *Nao* while slowing down goes faster.

The *Nao* begins to walk with a speed limitation of 100. This limitation is changed using the previous described method. But if it falls, everything is reset including the limitation to 100.

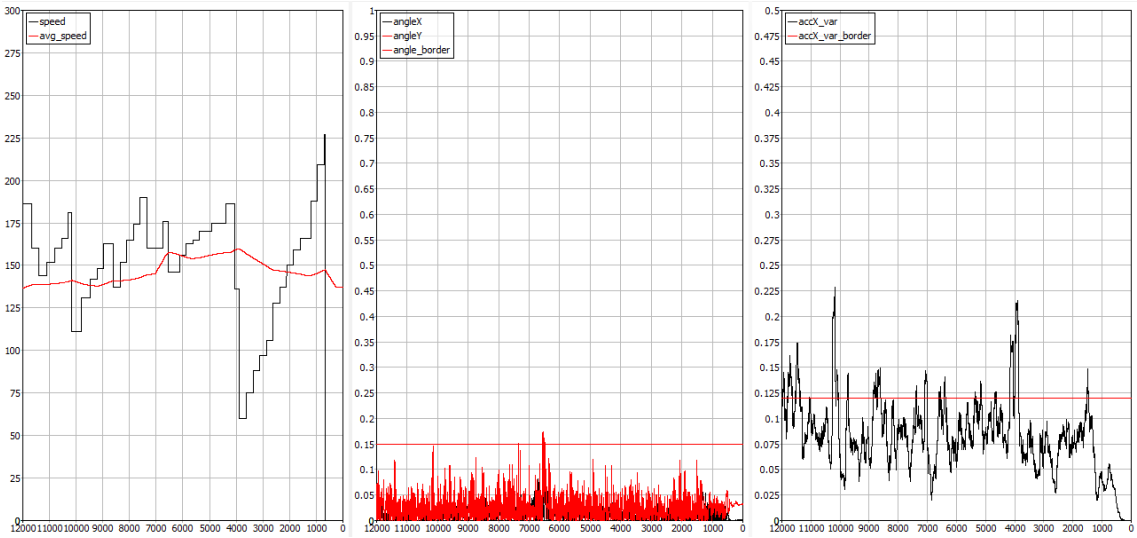


Figure 2.7: Speed, angles and variance of x-acceleration of *Nao* C

Figure 2.7 shows data collected using *Nao* C. The sub figures show the data of the last 12000 frames (2 minutes). The leftmost figure shows the speed by which the *Nao* was moving (black line) and the average over the last 2 minutes (red line). The central figure shows the angles in both x- and y-direction as well as the angle border (0 15). Between the last 6000 and 7000 is a peak, which exceeds the angle boarder. Because of this the speed was reduced, which can be seen in the leftmost figure. The rightmost figure shows the variation of the x-acceleration and the border (0 12). There are several long sequences in which the border is exceeded so the speed was reduced in these situations, e.g. near

the 10000 mark and near the 4000 mark. The latter is the longer sequence so the speed is reduced greater than near the 10000 mark. A good example for the increase of speed is directly after the last great decrease (near the $4000th$ frame). There is no peak in the figure showing the angles and no long sequence of exceeding the variance border so the speed can be increased a lot of times.

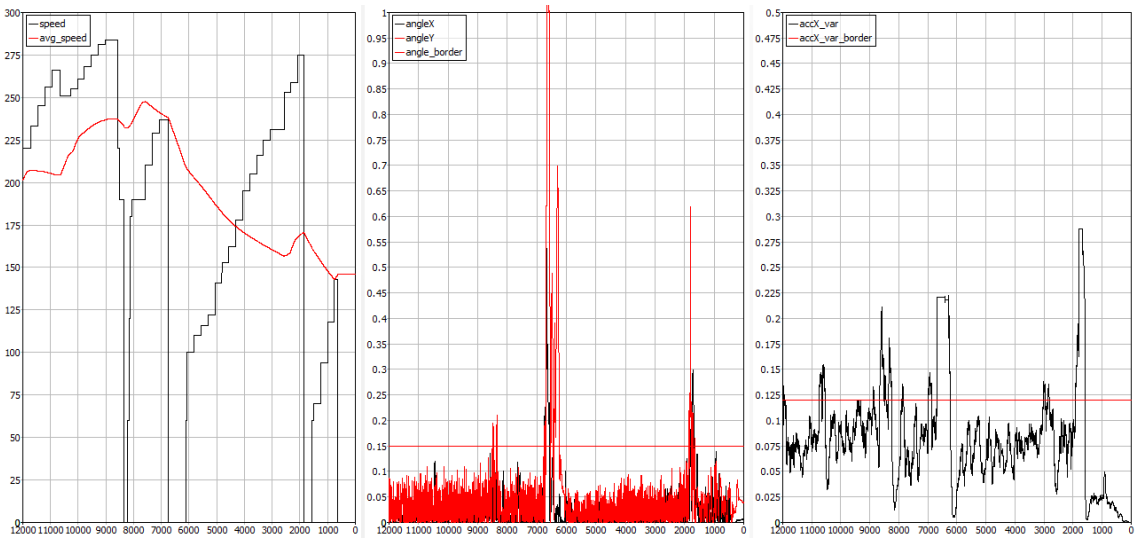


Figure 2.8: Speed, angles and variance of x-acceleration of *Nao* J

Figure 2.8 shows data collected using *Nao* J. The leftmost figure shows the speed by which the *Nao* was moving (black line) and the average over the last 2 minutes (red line) in the last 12000 frames (2 minutes). The central figure shows the angles in both x- and y-direction as well as the angle border (0 15). Between the 8000 mark and the 9000 mark there was a stumbling which resulted in a high peak, which resulted in stopping the walk of *Nao* J. Between the 6000 mark and the 7000 mark the *Nao* fell, so the speed limit was reset to 100 which can be seen in the leftmost figure. The last peak in the central figure resulted like the first in stopping the *Nao* which is also not produced by the previous described method. The rightmost figure shows the variation of the x-acceleration and the variance border (0 12). Like in figure 2.7 it can be seen that long sequences result in speed decreasing.

Outlook

In the future this could be used to make the *Nao* learn its limits. After decelerations the speed could be greater increased while the limit is far away and increased with smaller values when being near the limit. In figure 2.8 it could be seen that some values have to be tweaked as the method does not work good on newer *Nao*s. Another option would be being more carefully when walking with a speed faster than 200.

2.2 Center of Pressure

written by: Sebastian Engels

In addition to the x-acceleration it is possible to evaluate the stability of the walk using the data of the Center of Pressure (CoP). The Center of Pressure is the appearance point of the ground reaction force on the foot of the robot. The ground reaction force is the sum of all forces acting between a physical object and its supporting surface. The *Nao* uses the four foot pressure sensors to calculate the Center of Pressure. As the positions of the sensors are known, the center of pressure can be calculated after measuring the values of the sensors. So according to this calculation the x and y coordinates of the CoP of each foot are available. Since the robots mainly drop forward or backward, only the x-coordinates of the feet were considered. Furthermore, we consider the sum of the two x-coordinates. This has the advantage that only the development of one value has to consider drawing conclusions about the stability of the robot. Denote $x_{right,t}$ the x-coordinate of the right foot at time t and $x_{left,t}$ the x-coordinate of the left foot at time t the sum at time t is defined as:

$$x_t := x_{right,t} + x_{left,t}. \tag{2.7}$$

The foot pressure sensors are prone to errors, so that it can easily lead to measurement errors. If this has happened, the calculation of the CoP is faulty too. In Figure 2.9 the raw values for the CoP are shown for a period of 2100ms. It is observed that at times always comes to outliers. Since these occur only sporadically and are of short duration, may be assumed that these are erroneous measurements of foot pressure sensors at these values. In order to make statements about the walk, these measurement errors should be filtered out, so they do not affect the result. On closer examination of the CoP values reveals that they (with the exception of the outliers) do not pass or fall below the value of 0,3 or -0,3 in negative sector.

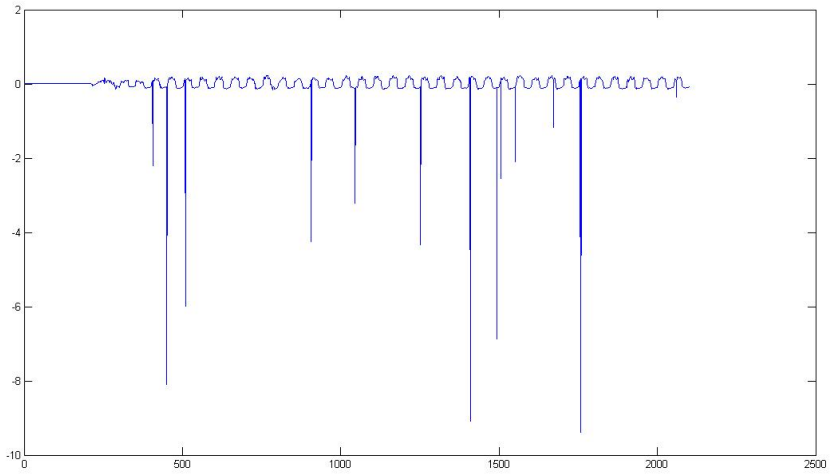


Figure 2.9: Raw values of the Center of Pressure

Thus, we ignore outliers and replace it with the previous value. The filtering of the data can therefore be described by the following function:

$$x_t := \begin{cases} x_{t-1} & if \quad |x_t| > 0{,}3 \\ x_t & otherwise \end{cases} \tag{2.8}$$

This filtering ensures that the value is valid, i.e. is in the range of values. The result after filtering is shown in Figure 2.10. It is clear that the values always fluctuate between a positive and a negative amplitude. This results from the individual steps of the robot. Depending on whether the front or rear foot is raised, the value of the CoP is positive or negative. As the amplitudes remain approximately the same, it is a good, stable, walk with a few variations.

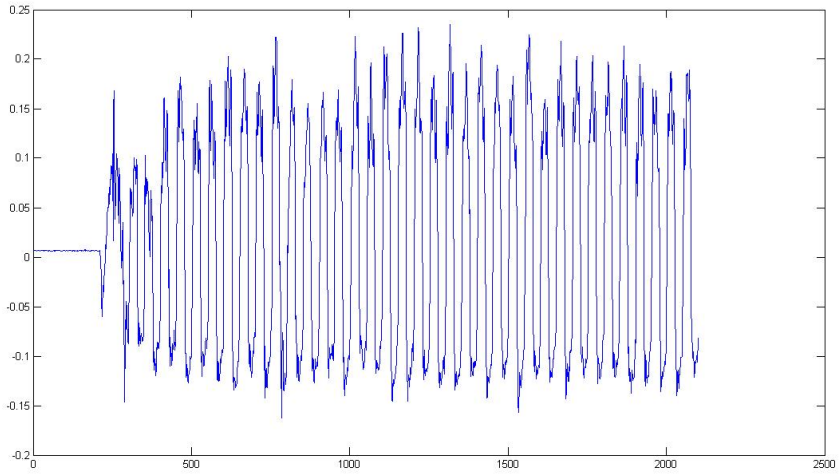


Figure 2.10: Values of the Center of Pressure after filtering

In Figure 2.11 the filtered CoP values of a walk on artificial turf are shown. In contrast to the stable walking it can be seen that the positive amplitudes vary greatly. In this case, a decrease of the positive amplitudes in the time periods $500 - 1200$ and $2800 - 3800$ is observed. The robot occurs during these periods on the spot and swayed so much that he finally fell over at times $t = 1300$ and $t = 3900$.

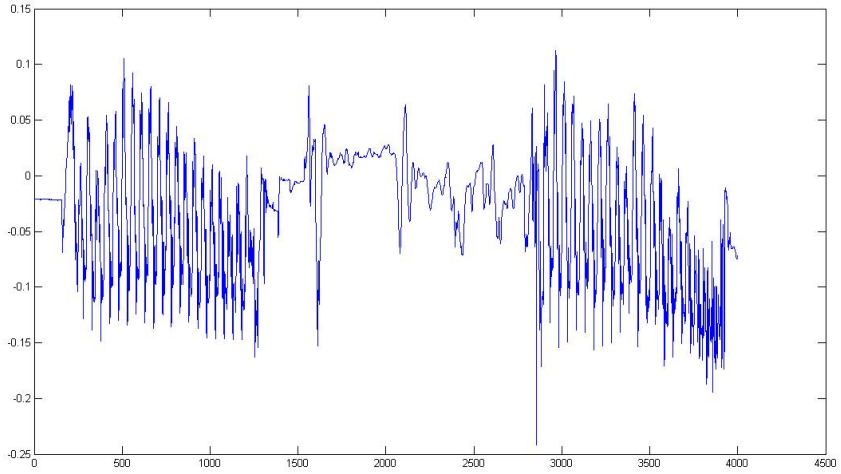


Figure 2.11: Filtered Center of Pressure values of a walk on artificial turf

The behavior of the positive amplitudes, that they are approximately the same for a stable walk and fall down for a unstable walk, could be detected in experiments with different robots and different velocities. Therefore, we use the course of positive amplitudes for the development of an evaluation criterion. For this reason, the current CoP values are filtered again, since only the positive amplitudes should be considered. A value x_t at time t is one of the relevant values $x_{rel,t}$, if the following holds:

$$x_{rel,t} := x_t \qquad if \quad x_t > 0. \tag{2.9}$$

To smoothen the smaller fluctuation of the amplitudes the moving average is calculated as in Section 2.1. Thus, the current value is influenced by older values. Good results can be achieved when 60 values have been considered for the moving average $\bar{x}_{aver,t}$ at time t:

$$\bar{x}_{aver,t} = \frac{1}{60} \cdot \sum_{i=t-60}^{t} x_{rel,i}. \tag{2.10}$$

In figure 2.12 the sequence of the moving average is shown with respect to the CoP values of figure 2.10. The increase in values up to the point 200 can be interpreted as a acceleration phase. Then the period of 300-500, a brief drop in the average is recognizable. At this time the robot had problems and got into wavering. From the time 500 the robot was stable again, which manifests itself in an almost straight course with a few fluctuations.

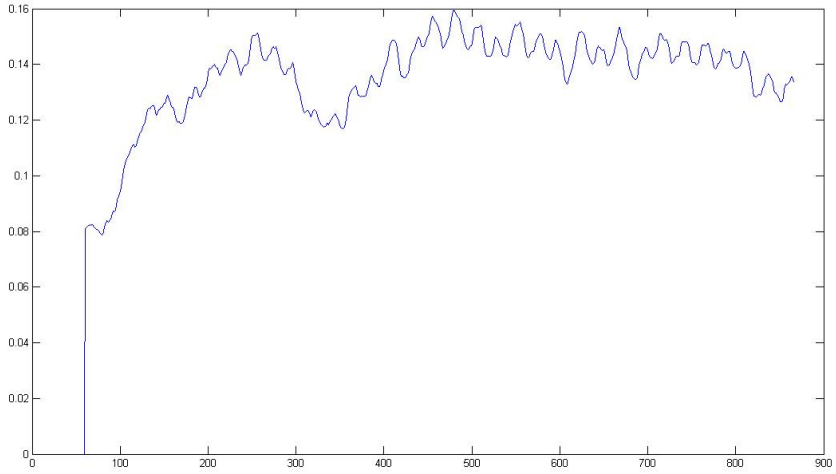


Figure 2.12: Moving average of the Center of Pressure values

The aim of the evaluation criterion is to keep the fluctuations as low as possible and to change the walk parameters at strengthen fluctuations (such as the time 300) to improve the walk. To detect these time points for changes the variance, as described in chapter 2.1, is used. The variance ${s^2}_t$ at time t in this case indicates the changes of the moving average $\bar{x}_{aver,t}$. So far, the variance of the last 30 values is used to indicates the changes:

$$ {s^2}_t = \frac{1}{30} \cdot \sum_{i=t-30}^{t} (\bar{x}_{aver,i} - \bar{x})^2. \tag{2.11} $$

In Figure 2.13 is the profile of the variance during a walk shown. When the variance increases over a certain limit value, the running parameters should be changed.

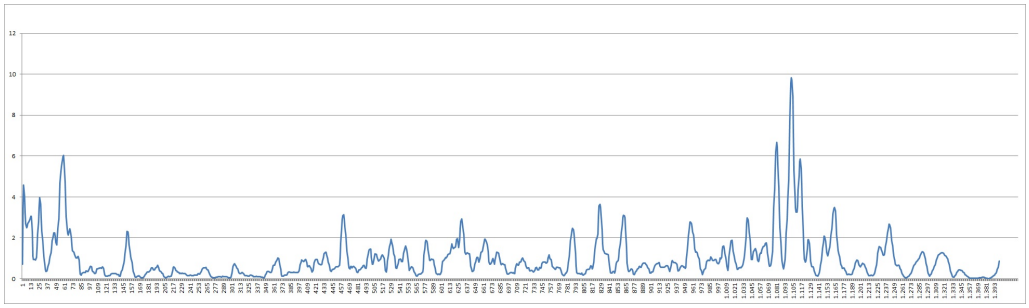


Figure 2.13: Profile of the variance during a walk

In runs with different robots, it was found that the variance values of the individual robots varied greatly. Thus, it was not possible to determine a general limit for the variance. Therefore, the limit should have been determined individual for each robot. For this purpose, self-learning approaches or automatic processes are also conceivable, but were not further investigated.

Furthermore, no changes in the course of the variance could be detected in experimental changes of the step height while running. For these reasons, it was decided to refrain from using the CoP as a stability criterion.

Chapter 3

Development of the LongKick

3.1 The existing Legacy Kick

written by: Stefan Kinzel

Initially the kick motion consisted of a *SpecialAction*. This is a static specification of joint angles and a static timing. The *Nao* performs a predefined movement without any possibility of parametrization, because *SpecialAction*s cannot be modified at runtime. The disadvantage is obviously the missing dynamics, such as a static kick direction, no reaction to changes in the environment or a bad CoP-Management. Also, there is no sensor control which leads to a missing compensation of instability.

An instant kick motion was developed earlier. This instant kick can be triggered during a walk phase and has the same duration as a normal step. This kick can be parametrized, of course within some bounds, e.g. the maximum kick angle. Due to the timing the instant kick integrates into a walk, so there has to be no change in timing and the forward projection of the ZMP. It can be seen as a special kind of step. Disadvantageous is the low power of this kick: there is no time for a strike-out-movement, so the kicking foot has not enough kinetic energy for a long range kick. Its range is just about two or three meters. Nonetheless, this instant kick is perfectly suitable for dribbling or in situations, where an instant reaction is needed.

In addition to this two kick movements (and, in the future, as a replacement for the slow and static kick motion) the task was to create a new kick, in dependence on the motion of the old special action kick.

3.2 Development Process

written by: Stefan Kinzel

This section describes the overall development of the new kick motion. First we explain some general considerations made in the beginning of the development (section 3.2.1). Secondly we describe the initial state of the framework before we started to implement the motion (section 3.2.2). We then implemented a raw draft for this kick (section 3.2.3), which suffered some problems we describe in section 3.2.4.

3.2.1 General considerations before implementation

written by: Mark Breddemann

There are several ways to specify the movement of the *Nao*s joints. The most intuitive implementation of the movement is setting the absolute degree values for each joint by time. The framework implements such a method by *SpecialActions*, in which joint angles can be specified at certain timestamps relative to the beginning of the *SpecialAction*. Between the given timepoints, the joint angles are linearly interpolated. Using this way, full control of the *Nao*'s kinematic is provided. This method has the disadvantage that the angles are completely static. Hence the *LongKick*-module should be dynamic, as it should be able to include the position of the ball and the kick target into its motion. Of course it is implementable to calculate the joint angles and set them dynamically, but the positions of the ball, the *Nao* and the kick target are given in world coordinates and therefore a translation would be needed to calculate the joint angles.

This leads to the next approach of implementing the movement: Setting the world coordinates for the foot position at several timestamps and interpolating the positions in the meantime by using b-splines. The walk of the *Nao* is implemented this way, as it is relatively easy to implement deviations of the normal going-straight-forward walk, like turning while walking. This is done by modifying the targeted foot positions on its way, especially the end point of the step. The interpolation with B-Splines has the advantage, that it leads into smoother movement, as it is able to avoid abrupt changes of the joints angles. The *Nao* Framework includes a so called *inverse kinematic engine*, which calculates the given foots world coordinates into joint angles which will be applied.

As the position of the ball, the *Nao* and the kick target is given in world coordinates, it is appropriate to implement also the *LongKick* by using only world coordinates and let the inverse kinematic engine calculate the joint angles. On top, there already exists a method that calculates the ideal traverse of the foot through the ball to kick the ball to the correct direction. This method was implemented as part of the instant kick and uses world coordinates.

3.2.2 Initial position before implementation

written by: Stefan Kinzel

We use the existing *InstantKick*module for our implementation. The Code for this kick is located in the `SwingLegController`, which is responsible for the movement of the feet.

The *InstantKick* consists of different b-splines, which are created using C++-macros like this:

```
START_POLYGON(2, polygonStart, _kickStart, theFreeLegPhaseParams.footPitch);

POINT_XY(polygonStart.x, polygonStart.y,
        theFreeLegPhaseParams.heightPolygon[0], kickStart.z);

POINT_XY(kickStart.x, kickStart.y,
        theFreeLegPhaseParams.heightPolygon[1], kickStart.z);
```

```
END_POLYGON(output, kickStartLen, 2);
```

In the beginning the start point (`polygonStart` in this example), the end point (`_kickStart`) as well as the length of the b-spline (`2`) and the foot pitch (`theFreeLegPhaseParams.footPitch`) have to be specified (`START_POLYGON`-macro). The `POINT_XY`-macro creates an intermediate point in the b-spline. With the `END_POLYGON`-macro, the b-spline is calculated with a specified length (`kickStartLen`) and written into an output variable (`output`). A point which is added twice will be exactly reached by the resulting spline curve while other points are just weak waypoints.

There are 4 parameters for the kick motion:

`polygonStart` Initially, the foot is placed at the `polygonStart`. Here all movements of the foot have to start.

`kickStart` The `kickStart` is the point, where the actual kick motion starts. This point is located in the air, so the foot has to be lifted to this point.

`kickStop` This point is the end of the actual kick motion. If the foot is placed here, it should have hit the ball, so it has to be in or behind the ball (from the robots view)

`polygonEnd` The `polygonEnd` is the point, where the foot has to be placed after the movement. The next step expects the foot at this position, so the current step (or kick motion) has to ensure, that the foot is placed here at the end

These points are calculated by the `initKick`-method, which is called before every kick. The kick motion itself is divided into three parts:

1. from `polygonStart` to `kickStart`
2. from `kickStart` to `kickStop`
3. from `kickStop` to `polygonEnd`

The time used for this *InstantKick* is exactly the same as for a normal step, so it can be placed in a normal walk without further problems. We used this *InstantKick* as a starting point for the development of our *longKick*.

3.2.3 Basic concepts of the LongKick

written by: Stefan Kinzel

The *LongKick* differs from the *InstantKick* in some ways. First of all, a longer period of time is scheduled for the *LongKick*. While the *InstantKick* is executed within one step duration, the *LongKick* has a duration of about 2.5 seconds. Because of the long single support phase, the ZMP has to be moved to the supporting foot. This also causes the robot to lean towards the supporting foot.

To achieve this, a different set of parameters is used. The usual parameters, which are stored in the `walkingParams.cfg`, are replaced by the parameters from the `freeLegParams.cfg`, so every parameter can be redefined for the *LongKick*.

The first step of developing the *LongKick* was the creation of three b-splines, similar to the ones of the *InstantKick*. To speed up the movement and to increase the power of the kick, we tested some other partitions, e.g. the division into five partitions, but the original partition turned out to be the best. Figure 3.1 shows the movement of the *Naos* feet during a walk to the ball and a following kick. The `kickStart` and `kickStop` are well visible. But before we reached this smooth movement, we ran into some problems which are described in the next section.

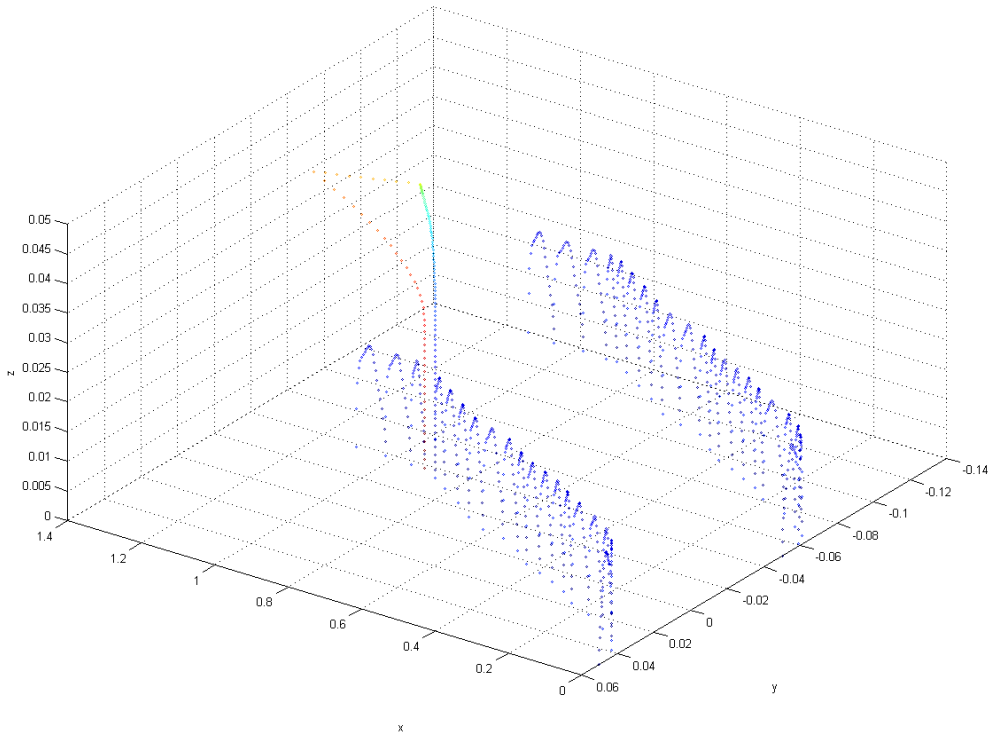


Figure 3.1: Plot of the foot movement of a *Nao*

3.2.4 Problems while implementation the LongKick

written by: Mark Breddemann

In this section the main problems which occurred while implementing the *LongKick* are described.

Trembling

It was clearly visible that the *Nao* began to tremble especially with the leg it was standing on, as soon the *Nao* leaned to one side and moved the other foot up from the ground.

The cause of this trembling is the sensor control engine of the *Nao* framework, which takes the values of the *Nao*'s acceleration sensor and its gyroscope to evaluate deviations from the intended (world) position and tries to compensate them by moving to the opposite direction.

The settings of the sensor control engine are optimized for walking, but it turns out that they are quite bad for slow and smooth movement like the stabilization phase at the beginning of the *LongKick*. However, turning the sensor control engine completely off while the *LongKick* is running lead to instability, as the engine is able to compensate symptoms of aging joints and especially older *Nao*s tended to fall down while kicking more often.

To overcome this, we enabled the sensor control engine, but reduced the influence of the motion to a very low level. Concrete, the absolute value *accXAlpha* is now lowered from -0.3 to -0.05. It seems logical that a very low value seems good, due to the slow and smooth movement during the stabilization phase of the *LongKick* and therefore no quick response of the sensor control engine is necessary, unlike the walk.

Applying new parameters during the LongKick

For the *LongKick* several configurable parameters during the calculation of the movement need to be changed. For example, the normal step length while walking is 250ms per step or a duration of 500ms for the repeatable movement of both foots. Of course, 250ms is not far enough time for a complete kick with a stabilization phase, a strike-out movement, the kick and again a stabilization phase. The time for the kick is increased to 2500ms or a complete step duration of five seconds by increasing the *stepDuration* parameter to 5. Another Example is the movement of the arms, which needed to be much greater than the normal walk to balance the *Nao* during the kick. Also, the kick requires a much longer time where both feet are standing on the ground than during the walk, to stabilize the *Nao* before the kick.

This distinction was done by using the *FreeLegParams* during the *LongKick* instead of the normal *WalkingParams*. This lead into several problems, as the whole calculation of the movement is organized in a pipeline:

1. At first the action of the next phase is planned, eg. one duration of walking (moving both feet) or kicking.
2. Then, the required time for each phase is reserved by creating the movement frames.
3. Afterwards, the handover points between the phases are calculated.
4. Only then the actual movement is inserted into the reversed frames.
5. Then, the sensor control engine applies and modifies the intended movement.

There are several stages missing in this enumeration, but is clear that this is a quite complex procedure. Unfortunately, the extend of the step duration during the *LongKick*

was not recognized in all stages which lead into very big problems during development of the new kick. For example, the feet handover point at the end of the kick was located to far forward, as the corresponding calculation thought the *Nao* walks for five seconds. Another example was the calculation of the arm movement, which simply did not apply the parameters of the *LongKick* and used the normal walk parameters instead.

In summary, many mostly small adjustments of the code were needed so that the new parameters are correctly applied and properly dealt with. Fixing all these problems took most of the time during development of the *LongKick* as the erroneous code was often hard to locate.

Abortion of the kick

As seen above, the whole movement process is quite complex and especially pipelined. It is possible, that the stage where the actual kick movement is calculated wants to abort the kick process, for example because the ball is too far away. Then, the kick phase has already been planned and the needed frames were reserved. This cannot be undone, so instead of a kick movement, only a normal step is inserted then. This step takes 2500ms, but due to the side movement of the upper body, the *Nao* is quite stable during this much too long step.

Adjust the behavior engine to the kick engine

To avoid the problem above, the behavior engine must only trigger the *LongKick* when it is possible to actually kick. This was done by modifying the behavior parameters, for example lowering the *safeOpeningAngle*.

Unreachable positions of the inverse kinematic engine

When developing the *LongKick* we used a fixed *MotionRequest* and *BallModel*, as seen in chapter 3.3. At the time a acceptable *LongKick* was implemented, we did some "real world" tests in the simulator, where especially the *BallModel* was very different. As the *LongKick* reacts on the ball position to kick it to the right direction, dynamic trajectories of the shooting foot are calculated depending on these parameters. Sometimes this lead into a feet trajectory which could not be reached by the *Nao*.

This behavior was fixed by doing many tests and modifying especially the parameters *ballXMax*, *ballYMax*, *ballXMin* and *ballYMin*. With these adjusted parameters the *LongKick* was aborted, rather than invalid trajectories were calculated.

Instability in particular at older Naos

Before kicking, the *Nao* moves to one side for shifting its center of pressure over the standing foot, so it is able to move the kicking foot up from the ground without falling down. As

the joints of the older *Nao*'s are worn out, the COP is not always moved exactly over the standing foot which leads into instability. This was solved by adjusting the target position of the COP by modifying the parameters *polygonRight* and *polygonLeft* towards leaning more aside. This solution hat got the disadvantage, that the parameter value must be specified for each *Nao*, as the joints wearout varies from robot to robot.

Other problems

There were several other problems, like thrown assertions because the *standType* in the *MotionRequest* was not reset after an abortion of the kick or a very strange behavior of the inverse kinematic engine when the joints movement was too fast. We fixed these problems, but when looking back, we can say it was very time-consuming to fix all these errors and it took by far the most time in the development process of the *LongKick*.

3.3 Triggering of the LongKick

written by: Stefan Kinzel

In the simulator the *LongKick* can be triggered as follows:

```
set representation:MotionRequest
        motion = walk;
        standType = leftSingleSupport;
        specialActionRequest = {specialAction = stand; mirror = false;};
        walkRequest = {
                request = { rotation = 0; translation = {x = 10; y = 0;}; };
                requestType = speed;
        };
        kickRequest = { kickTarget = {x = 0; y = 0; }; };
        kickDirection = 0;
        kickTime = 10;

set representation:BallModel
        lastPerception = { position = {x = -210.641; y = -410.166;};
        velocity = {x = 0; y = 0;}; };
        estimate = { position = {x = 150; y = 50;};
        velocity = {x = 0; y = 0;}; };
        ballLost = false;
        timeWhenLastSeen = 59736;
        timeWhenLastSeenByTeamMate = 0;
```

The first command triggers the *LongKick* with the left foot. The second command fixes the ball model, so the *Nao* "sees" the ball on the correct position to execute the kick.

The `kicktime` must be a positive number to trigger the *LongKick*. If the `kickTime` is negative, a *InstantKick* is triggered. For the execution of a *LongKick*, the ball has to be

at a kickable position. For this, the perfect ball position is 15 cm in front of the robot (`x = 150`) and 5 cm to the left (`y = 50`)[1].

The behaviour can trigger the *LongKick* using the same constraints, e.g.:

```
float kickDirection =
        std::max(std::min((float)atan2(target.y,target.x),pi_2),-pi_2);
localMotionRequest.kickDirection = kickDirection;
float kickTime = 10.f;
localMotionRequest.kickTime = kickTime;
if (theBallModelAfterPreview.estimate.position.y < 0) {
        localMotionRequest.standType = rightSingleSupport;
} else {
        localMotionRequest.standType = leftSingleSupport;
}
```

This is a snippet from the `kick.h` we used to trigger the kick in our tests. Primarily, this code was used to trigger the *InstantKick*. As written before, the main constraint is the positive `kickTime`. The value of the `kickTime` has no influence on the kick itself, it is just used to decide, whether the *InstantKick* or the *LongKick* should be executed.

Depending on the position of the ball, the right foot has to be chosen. If this is done wrong, the kick will be aborted (also mentioned in section 3.2.4). The if-statement in the example does this the right way.

3.4 Result

written by: Stefan Kinzel

In the end, the resulting kick motion was obviously more powerful than the *InstantKick*. The *InstantKick* barely shoots the ball farther than three meters. Under optimal circumstances, the developed *LongKick* shoots the ball about seven meters, with further enhancements the distance can be improved. Also, the new kick is more parametrizable than the old *SpecialAction* motion. The kick direction can be specified and many parameters can be tuned in the `freeLegParams.cfg` file. This includes the maximum angles of the kick movement, the arm movement, the ZMP curve during the kick and some other, mostly more unimportant settings. These parameters not only influence the kick motion itself, but many other modules like the arm movement controller or the ZMP calculation. So the kick motion can be fine tuned for each robot.

Due to the occurring problems, not all of which were resolved, we don't think this kick is ready for use in a real match. Some problems have to be solved, but we had not enough time to do this within this project group. This could be a task for the next project group.

[1] For a kick with the left foot. If the kick should be executed with the right foot, this must be `y = -50`, of course.

Part II

Behavior

Chapter 4

Introduction

written by: Stefan Rötner

In the existing framework [7] the behavior module controlling the agents is realized using a deterministic state automaton. One of our main goals is to assess alternative approaches, such as an implementation of the BDI model for intelligent agents as foundation of behavior control.

In addition more approaches from the field of logics should be utilized. We will focus on logic programs (with regard to the huge amount of uncertainty we are facing, especially under the answer set semantics), fuzzy logic and Extended Behavioral Networks. We will present the 2D simulation league as a playground to experiment with new approaches to the realization of the robots' AI. As part of the fuzzy logic chapter we will also present first mechanisms of game situation detection that might be relevant for a coach robot, which is going to be introduced in the very near future.

This chapter will provide a description of the theoretical foundations of the selected approaches and present the practical implementations.

Chapter 5

Modelling World States with Fuzzy Logic

written by: Jan Gehlhaar

In this section a new approach for modeling world states using fuzzy logic is introduced, which was implemented and evaluated by the project group. Starting with the motivation and a brief introduction to the concepts of fuzzy logic, the details of the new approach and the evaluation are presented. Finally, this section closes with an outlook for possible enhancements and future use cases for the developed system.

5.1 Motivation

Every behavioral system needs some sort of modeling of the current state of the environment, on which to the system has to act or react. Currently the *Nao* framework exists only of symbols with a low level of abstraction, like whether the ball is seen or not or if a player is nearest to the ball.

The goal of this project group is to bring the abstraction of environmental symbols to a new level, by introducing a symbolic representation of a current game situation. The values for this new symbol range from a defensive to an offensive situation and are defined as follows:

- `Critical Defence` A critical defence situation occurs, when the opponent team is either about to score a goal or is in an advantageous position, so that scoring a goal soon is very likely.
- `Defence` This is an ordinary defensive situation, in which the opponent team has possession of the ball and is approaching the goal of the own team.
- `Open Game` An open game depicts a situation, which can result in an attack either by the own team or by the opponent team. An example of this might be a situation, in which no player is currently near the ball.

- `Attack` Analogous to the defence situation, the attack situation describes that the own team is now in ball possession and is advancing on the opponent's goal.
- `Critical Attack` Again analogous to the critical defence, in a critical attack situation the own team is either in striking distance to score a goal, or is in such an advantage that this will so be possible.

As most of these values are very vague, traditional computational approaches to determine those situations, like thresholding or some sort of linear interpolation algorithms, are not really suitable. Therefore, this project group evaluated a new approach for inferring those values by using fuzzy logic. Fuzzy logic offers some advantages over other techniques, like the possibility to express simple if-then rules for inferring game situations, like "If the opponent is near the ball, but I am not, then this is a defensive situation." as shown in the following equation

$$d_{ball}(opponent) = near \wedge d_{ball}(me) = \neg near \Rightarrow situation = defence, \tag{5.1}$$

where d denotes the distance to the ball. Another beneficial characteristic of fuzzy logic is a good interpolation behavior, which counteracts flickering or rapid jumps of the calculated values and results in a stable interpretation of the current game situation.

5.2 Introduction to Fuzzy Logic

The theorem of fuzzy logic was first introduced by Lofti A. Zadeh[8] back in 1965. The main difference to the conventional set theorem is that elements are not restricted to either belonging to a set or not, but can have any degree of membership to a fuzzy set. This allows for sets, that - in contrast to conventional sets - have no distinct boundaries. This becomes even more obvious, when taking a closer look on the membership functions of the two set types.

The membership functions of traditional sets are mapping from the super set into the set of 0,1, while the membership functions map from the superset into the *interval* [0,1]. For both function types the properties of "an element belongs to the set" and "an element does not belong to the set" are encoded by zero and one respectively. In allowing soft boundaries for fuzzy sets, a whole series of everyday problems can now be modeled more precisely. For instance, the fact that a human is tall can be more adequately describe by a bézier curve than with a simple step function as shown in figure 5.1.

Like with traditional sets, it is possible to describe propositional formula with the typical logic operators, like conjunction, disjunction, negation, and implication. Here, however, the rule applies also, that the values for those logical expressions are not solely restricted to the value of zero and one, but can accept any value between those two. In the last decades a bunch of different semantics for the logical operators have been proposed, with which the expressions can be evaluated. Further discussion of the different semantics can be found in [9, 10].

The following example demonstrates how fuzzy logic can be used to model a real world problem: Assuming a car manufacturer wants to develop a new adaptive cruise control

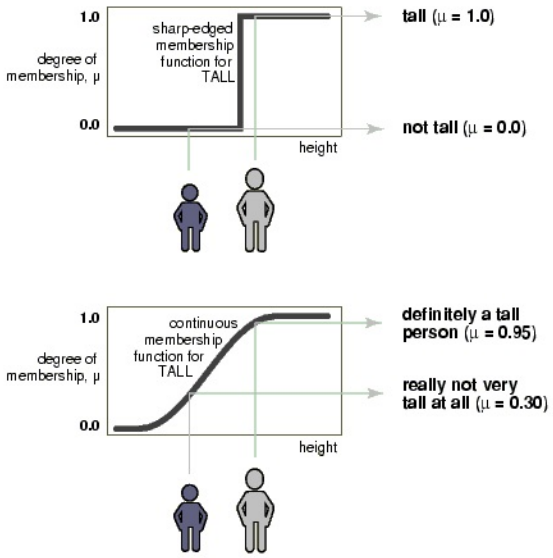


Figure 5.1: Membership functions
(Source: http://radio.feld.cvut.cz/matlab/toolbox/fuzzy/fuzzy_ta.gif)

system, which alters the current velocity of a car according to the current traffic conditions on the road and the mood of the driver. The system should ensure maximal speed without causing a state of panic in the driver. For this the current car density for a finite section of a highway and the adrenalin level of the driver as an indicator for the driver's mood will serve as input parameters for the system. The car density super set will be partitioned into the three fuzzy sets "dense traffic", "normal traffic" and "light traffic". Additionally, the adrenalin super set will be divided into the fuzzy sets "bored", "relaxed" and "tense". The output of the system will be the change in speed, which consists of the fuzzy sets "decelerate", "hold speed" and "accelerate".

With all those sets defined, simple if-then-rules can be defined, which together will describe the behavior of the cruise control system:

$$\begin{aligned} traffic = light \wedge mood = relaxed &\Rightarrow speed = accelerate \\ traffic = dense \wedge mood = tense &\Rightarrow speed = decelerate \\ traffic = dense \wedge mood = bored &\Rightarrow speed = accelerate \\ &... \end{aligned} \tag{5.2}$$

In order for the system to decide, whether to go faster or slower, all rules must be evaluated for every decision. This starts by transforming the crisp input values into fuzzy values for the atomic expressions like $traffic = dense$. This step is called *fuzzification* and is performed by evaluating the membership functions of each fuzzy set according to the input value. After that, the inference step takes place by evaluating each fuzzy rule. Finally, in order to get a distinctive value as an output parameter of the cruise control, that ought to result in an alteration of the current speed, the calculated fuzzy values for all rules must be aggregated and converted back into a crisp value. This step is called *defuzzification* and results in a single value, that can be used for speeding up or slowing down the car.

In the last decades there have also been several proposals for the semantics of the defuzzification, but those will not be described in more detail in this report. Interested reader can find further information on the semantics in [9].

5.3 Current Approach

The first step for the new approach is to develop a system that can perform all necessary tasks, associated with the processing of fuzzy logic. While there already exist some implementations for this purpose, a completely new system was build. One reason is, that the existing systems are all focused on very special use case scenarios and therefore have limited capabilities. Another reason for creating a new library is to make sure to have grasped all concepts of fuzzy systems. But the main reason is the need for an infrastructure, with which the necessary fuzzy symbols and rules can be efficiently developed and evaluated. This called for a tool set with a rich gui support and although it would have been possible to archive the needed interoperability between those tools sets and an already existing fuzzy system, working with a newly developed fuzzy library goes all lot smoother.

For developing a rich gui experience on the Windows platform, the programming language C-Sharp and the possibilities of Microsoft's Windows Presentation Foundation were chosen. For the processing and execution of fuzzy logic a library, called *FuzzyController*, was developed. The *FuzzyController* is capable of evaluation fuzzy rules according to several strategies that were presented in [9]. The library was also developed in a test driven manner and shows currently a test coverage of over 95 percent.

5.3.1 Functioning of the *Fuzzy Controller*

The *FuzzyController* can be seen as a black box processing system for fuzzy logic. It receives a set of numeric values, processes those values according to a specified strategy against a set of fuzzy rules and finally computes a numeric result, which represents a defuzzified value. The controller must be initialized with the desired evaluation strategies and the definition of the fuzzy logic, which includes fuzzy rules and the definition of the fuzzy symbols, which are used by those rules. Additionally, a mapping, defining which numeric variables are to be expected as the controllers input and are assigned to which fuzzy variable, is needed for the definition. The inference logic itself (i.e. the fuzzy rules) is composed of iterations. Each iteration has a set of fuzzy input variables, a set of fuzzy rules and a set of fuzzy output variables, that receive their values by evaluation the given rules.

The first iteration always represents the fuzzification of the numeric values into fuzzy values and therefore no rule gets applied in this step. The result of every iteration is called a "scope", which is as set of current values for the fuzzy variables, which will then be used for the next iteration. With each subsequent iteration a new scope is created, holding the new values for the fuzzy variables, which were calculated using the current subset of fuzzy rules. Values that are not used be the rules of the current iteration also get included into the scope. After the last iteration the fuzzy variables, which have been marked for

defuzzification in the definition of the fuzzy logic, are converted back into numeric values, according to the specified defuzzification strategy.

Although the *FuzzyController* can be though of as a black box processor of fuzzy logic, the scope concept offers the possibility to read the internal states of the inference mechanism by simply including the values of each scope into the controller results. This can be effectively used for debugging purposes and improvements of the fuzzy logic with a tool, which was also developed by the project group, the *FuzzyLab*.

5.3.2 FuzzyLab - An Editing and Evaluation Program for Fuzzy Logic

The FuzzyLab application serves two points. First, it enables users to define the fuzzy logic for the controller in a WYSIWYG[1]-fashion. Secondly, it can be used to debug and evaluate this logic, by allowing to see inside the controller and observe the inference mechanism on close.

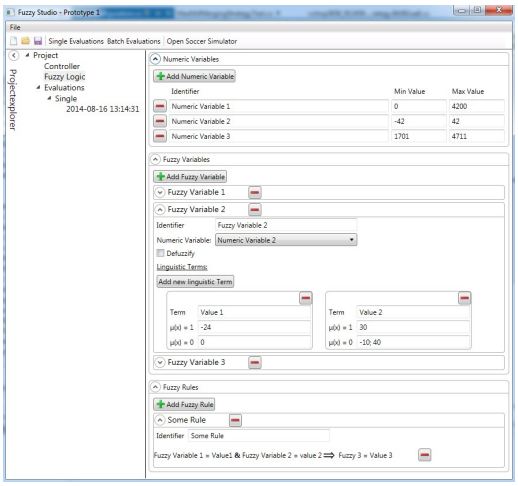


Figure 5.2: Graphical user interface of the FuzzyLab application

Figure 5.3: The situation editor PlugIn

The FuzzyLab application is designed as a plug-in system for defining and evaluating fuzzy logic. All required fuzzy variables, values and rules can be defined and then evaluated, by manually providing numeric input values for the controller and then examining the output results of the controller via the user interface. Besides manually entering the input values, it is possible to provide these values with user defined plug-ins, that can be mounted into the application. For instance it is possible to write a plug-in, that provides these values from real time sensory data, that are received from the robots via their wlan interfaces.

One plug-in, which was also developed by the project group, is the Situation-Editor plug-in, as shown in figure 5.3. This plug-in lets the user define new game situations by simply moving the players and the ball on the field. The Situation-Editor can also send numeric values of a selected situation to the fuzzy controller.

[1] What you see is what you get

5.4 Modeling game situations with fuzzy logic

One main challenge when modeling the game situations of a soccer game as described above, is to get the size of the state space under control, because every new introduced input variable results in an increasing number of new combination of all input variables. Therefore, if the fuzzy logic ought to cover all possible scenarios in a soccer game, one ends up with an exponentially growing number of rules for each additional input variable. In order to shrink the number of fuzzy rules to a manageable magnitude, the fuzzy rules were organized in a hierarchical manner. The idea was to extract a set of new fuzzy values from the initial input values, that have a slightly higher level of abstraction. Then those abstract variables get used in the next stage to result into a new set of values with an even higher abstraction level and so forth. Although this can be applied in a very granular fashion, the first version of the fuzzy logic uses only two stages for inferring the game situation. In the first stage the game situation in the immediate vicinity of the ball, the own goal and the opponent goal are calculated. After that, the three separate game situation are combined to infer an over all game situation. Figure 5.4 illustrates the different stages and levels of abstraction. Please note, that $XX_Center_Distance_To_X X$ denotes the median distance rather than the average distance.

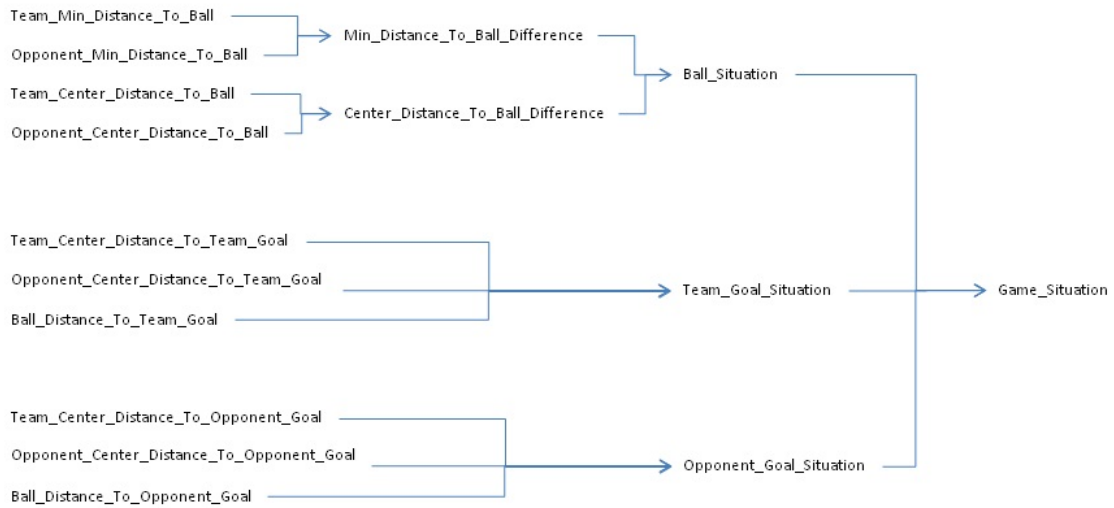


Figure 5.4: Hierarchical fuzzy logic.

5.4.1 Evaluation of the current approach

In order to effectively evaluate the suitability of the fuzzy logic approach, some samples, labeled with ground truth data, are needed. For this the Soccer-Simulation plug-in was used to create one hundred sample game situations, which were either reconstructed from video footage of actual games, create by deliberately choosing positions for the players or by randomly assigning player positions. The members of the project group then annotated those sample situations with a small Java program, which was specifically developed for this purpose. In order to see how stable the annotations will be, all situations were labels by each member twice, resulting in over 2000 labels. It turns out that the labels of all

users, beside some minor spikes, which turned out to be labeling errors by the users, were in fact quite consistent for each sample, and therefore are suitable for the evaluation of the fuzzy controller's performance.

Although the first version of the fuzzy logic is a very simple one, it shows some promising results, when evaluated against the ground truth data. 50% of the situations that were assessed by the controller lay within the standard deviation of the annotation by the project group participants. While randomly picking a value would have only resulted in a roughly 24% chance of lying inside the standard deviations, this shows that the current approach, although the fuzzy logic is a very basic one, has great potential for correctly detecting game situations. Furthermore, the calculated situation values showed a standard deviation of 15.5% from the average ground truth situation values and a 16% standard deviation from the median values.

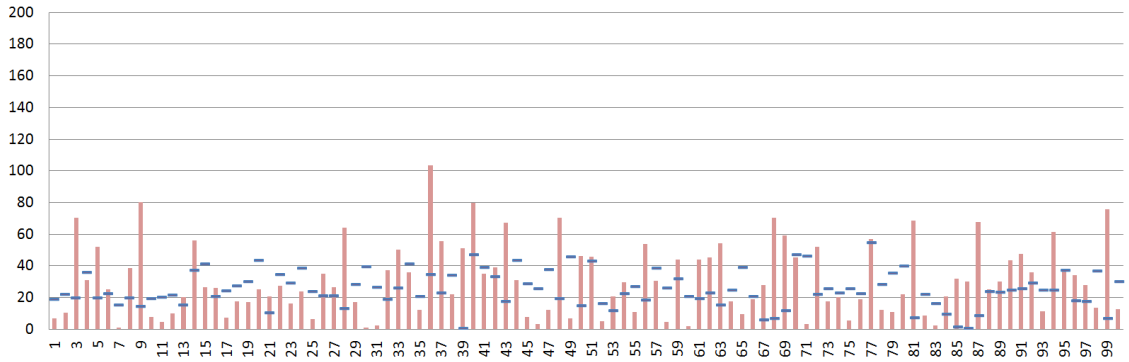


Figure 5.5: Results of the evaluation.

Fig 5.5 shows the results of the evaluation. The red bars represent the difference of the values that are calculated by the fuzzy controller to the annotated ground truth values, while the blue stripes depict the standard deviations of the ground truth data.
Technical note: The values in the diagram are scaled to the range between 0 and 200, because 200 represents the maximum difference between two game situation values. This is due to the fact, that the fuzzy logic uses the value of -100 to represent a most critical defence, while a most critical attack situation results in a value of 100.

5.4.2 Further improvements

As stated before, the current version of the fuzzy logic despite its simplicity shows a lot potential and many aspects can be further improved. First of all it should be mentioned that, with 100 game situations, the data quantity that is available for the evaluation of the fuzzy logic is rather small. One improvement can be to increase the number of game situations to obtain a broader spectrum of possible game states. A closer look at the current fuzzy logic reveals another room for improvement. When counting the number of rules that lead to a certain game situation, it strikes out that there is a higher number of rules that describe an "open game" situation. Therefore, the fuzzy logic is always biased towards this game state. One way to archive a more impartial assessment of the game situation by the controller, is to distribute all rules equally across all five game states. But

there is also another way to get a more impartial response from the fuzzy logic, which leads to the next possible improvement.

As of now, each fuzzy rule is treated equally according to its importance for the inference mechanism. But as in real life there are situations in which some rules are not applicable, because they can not possibly provide a correct result. For instance there is a rule, that describes a defensive situation, when a opponent player is much nearer to the ball than a player of the own team. But this implication is not of equal use in every game scenario. It is obvious, that this rule becomes increasingly important for the current game situation when the distance to either one of the goals decreases. While the modeling of this correlation can also be achieved with the current inference system, it is much more natural to be able to express the importance of a rule, by another set of fuzzy expressions.

Finally, a new and revised modeling approach can be used, instead of the two stage approach, which was initially tested by this project group.

5.5 Future Work

As shown in the previous sections the fuzzy logic approach has a good chance of reliably detecting game situation. The next steps will therefore be to incorporate the fuzzy inference system in the *Nao* framework. Additionally, the fuzzy logic will be evaluated according to robustness to signal noises. Those signal noises can be simulated in a controlled manner by randomly moving the objects on the field inside a given radius for each input game situation. Then those radii can get continuously increased until the precision of the fuzzy controller falls below a certain threshold. Those radii can then be used as a quality measurement of the object localization by the *Nao* recognition and localization system. Furthermore, it can also be investigated, whether fuzzy logic can also be used to infer roles for each robot in a given game situation. Leading to specific actions for a role, which the robot can execute. This would result in a complete fuzzy driven behavior for *Nao* robots.

Chapter 6

Simulation League

6.1 Robot Soccer Simulation League

6.1.1 Introduction and Motivation

written by: Lukas Pfahler

Several different RoboCup Soccer Simulation leagues exist, but we will focus on the 2D Simulation League *rcss2d*. As its name already implies, rcss2d models a soccer match in two dimensions. Therefore, both players and ball are embedded in the euclidean plane. A rcss2d match consists of up to 22 players, 11 per team, where only one player per team may act as a goalkeeper. A player can move and turn, kick the ball and tackle or even foul other players. Each player has a limited view range and has to react to its surrounding environment based on incomplete and noisy information. A match is divided into 6000 cycles, i.e. 3000 per half, where one half usually is about 5 minutes long, so the entire match takes about 10 minutes. In addition to the players, there may also be a so called online coach per team which may coordinate the players and update the teams strategies.

We thought about enrolling in the 3d-simulation league, where one has to steer the simulated robots not by simply issuing dash-like commands, but by setting angles and positions for all joints of a simulated robot model. However, since one group of students of our project group is already focusing on optimizing the movements of robots, we feel that focusing on motion in another setting is redundant and keeps us from achieving results in the core area of our interest: improving the behavior.

Thus we chose the 2D Simulation League (rcss2d) described above. It allows contestants to develop any kind of artificial intelligence, for instance a system that controls all 11 players simultaneously with shared, central knowledge and a centralized planner. However, we see the simulation league as a possibility to test ideas for future artificial intelligence in robot soccer and we want to lay groundwork for future seasons of the RoboCup. Thus, we constrain ourselves to a model close to the real soccer-playing robots and try to draw as many parallels between our simulated soccer players and the robots as possible, both from a theoretical and from a software developmental point of view. Our goal is to develop an artificial intelligence that can be adapted to the real robots and replace the current state automata.

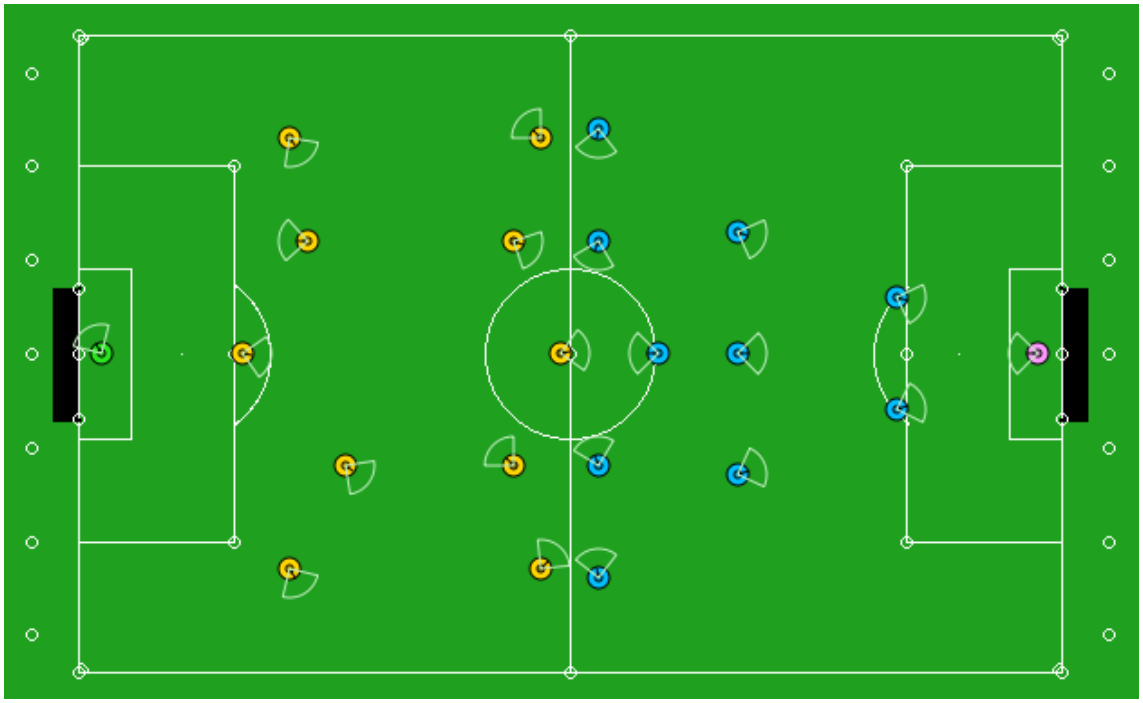


Figure 6.1: The RCSS Monitor visualizes the current state of a simulation game.

We see players as autonomous, individual agents, that is we have 11 instances of our artificial intelligence running independently of one another. They don't have any kind of implicitly shared knowledge, the only mean to explicitly share knowledge is by communication (see below). This implies, that each agent has a different idea about the status of its environment and the game and agents may or may not agree on that status. Furthermore, each agent independently decides on a next action based on its idea of the environment. The obvious challenge is to get these independently acting agents to play as a team, i.e. plan actions that are not only intelligent from an individual perspective but from a team perspective. For instance, running towards the ball surely is a good idea for an individual player, but having 11 players run for the ball will not result in a good outcome.

From a software-developmental point of view, we try to build a framework that is close to the *Nao*framework used on our real soccer-playing robots. In our *Nao*framework there are Modules that process sensor inputs and combine gathered data into so-called symbols that model the current state of the game or the environment of our robot. The artificial intelligence or *behavior* then uses these symbols to act within the environment. Currently, this is done using an automate system, that changes its status according to the changes of the environment and simultaneously starting actions, like walking or kicking, to change the state of the robot within the environment. These actions are called *special actions* and these special actions are then executed. A special action essentially describes a sequence of commands that change the positions and angles of the robot's joints. We will present a system that has essentially the same components for simulated robot soccer.

6.1.2 Robot Soccer Simulation Server Protocol

written by: Till Hartmann

The *RCSSServer* is the component which handles the state of the match, i.e. has an internal model of the field, the ball, the players and simple physics (movement speed, acceleration, sight, ...). It is a highly asynchronous system with two independent loops, where one loop issues perceptions every 150 milliseconds (by default) and another accepts commands sent by a client once every 100 milliseconds.

Note that the information the server sends to its clients is noisy by default[1], that means the client must not assume the information it receives from the server to be accurate.

Commands

Some atomic commands which may be sent by a client are

Action	Parameters	Description
`dash`	`force`	moves a player in the direction he is facing with given `force`.
`turn`	`degreeMoment`	turns a player around itself at a maximum of `degreeMoment` degree.
`kick`	`force, dir`	kicks the ball in the specified direction `dir` with the given force `force`.
`say`	`msg`	Shouts `msg` across the field. Nearby players can hear this message.

For a complete list of available commands, see `http://sourceforge.net/projects/sserver/files/rcssmanual`.

Sensors

Each client receives sensor data from the server, which we will call *perceptions*. These perceptions can be categorized either into *aural*, *visual* or *body* perceptions, where aural perceptions are messages from other players on the field, from a coach or from the referee, visual perceptions contain information on *which* object was seen (by the specific player) *where*[2] and body perceptions offer information such as how much stamina[3] a player has got (at that moment). For a complete overview of possible perceptions, see `http://sourceforge.net/projects/sserver/files/rcssmanual/`.

[1] as with most server related settings, noise may be disabled by the user

[2] usually *relative* distance and direction

[3] if a player runs out of stamina, he will be unable to perform certain actions until enough stamina is recovered

Perception type	Possible subtypes	Information
visual	`ball`, `player`, `flag`	at least *direction* and *distance*; if close enough also contains *dirChange* and *distChange*. If the object seen is a player, might also contain *bodyFacingDir* and *headFacingDir*
aural	`self`, `player_n`, `enemy_n`, `online_coach_[left|right]`, `referee`	messages from the respective sender. The messages themselves are restricted to a fixed set of characters and a maximum length. Messages 'shouted' across the field are only 'heard' by players in the vicinity of the sender. For *online_coach_left* or *-right* these *may* but do not necessarily follow a certain protocol.
body		statistical data, such as *kickCount, dashCount, ...* aswell as *stamina*, *viewMode* and *speed* (-amount and -direction). Messages by the referee usually initiate special routines, such as *kickoff*, *foul_[l/r]*, *freekick_[l/r]* etc.

Communication

On a technical side note, the UDP datagrams sent by the server are so called *S-Expressions*, which basically represent nested lists (i.e. trees). These need to be parsed and interpreted by the client.

6.2 Our SimLeague Agent

The framework is split into two main sections: server communication and artificial intelligence. We will give a short overview of its structure and some basic API usage.

6.2.1 `libsimleagueagent`

Communication from and to an RCSSServer-instance is handled by `libsimleagueagent`. It is responsible for sending commands, reading and parsing messages and converting them to so called perceptions. The core class in libsimleagueagent is `TwoDAgent`, which allows connecting to a server and offers methods to send commands to the server and to listen for perceptions received from the server.

```
TwoDAgent agent = new TwoDAgent("YourTeamName");
agent.setAI(new YourAi());
```

```
agent.init(); // sets up connection to server
agent.run(); // starts perception-receiving-thread, etc
```

In this case `YourAi` is a class implementing the `Ai` interface[4]. We provide `AbstractPerceptionProcessor` which serves as a kind of multiplexer for perceptions - if a new perception arrives, it is handed off to one of its many specialized methods - and can be used as a starting point for a custom Ai. For example, if a visual perception with information about the ball arrives, it will be forwarded to the `onBallSeen` method, which sensible `Ai` implementations override.

6.2.2 Agent

written by: Lukas Pfahler

Concept

As mentioned above, we want to develop a system that has as many parallels to the *Nao*-robots and the *Nao*-framework as possible.

Like the *Nao*robots, our simulated players have to deal with noisy visual perceptions. Of course, noisy perceptions immediately implies the problem of uncertain knowledge. Starting from noisy input data we try to extract knowledge about the state of the environment, for instance we try to compute a players position based on its visual perceptions. These extracted pieces of information are obviously not perfect due to the noise. We discussed only playing on servers, where the noise is deactivated, however we decided that a key aspect or key challenge of robot soccer is dealing with uncertainty and thus it is a challenge we also want to tackle. However, in contrast to the real robots, we have information about the noise model used by the server to add noise to the incoming perceptions, which we can use to rate the quality of perceptions, thus allowing us to extract better information by preferably using 'good' input data. We believe that a similar approach could also be used on real robots.

The same problems basically also apply to the the output side of our behavior: The action commands we issue are not executed exactly but are randomly modified. For instance, kick commands have a random noise added to the angle of the shot, influenced by how well positioned the player is for the kick. However, we admit that the problems of uncertain motion are much smaller in the 2d-simulation league than in real robots.

Like the *Nao*framework, our agent framework is twofold – there is one part for motion and one part for behavior. The motion part essentially consists of a thread executing *special actions* from a queue. Special actions, an expression we borrowed from *Nao*framework, combine atomic commands into higher level actions, like looking for the ball or moving to a position, that respond reactively to perceptions.

The Planner Thread models the environment of the soccer field and computes plans – sequences of special actions – in an infinite loop. The world is modeled using simple logical

[4] which extends `PerceptionProcessor`

symbols and actions describe changes to these symbols, thus allowing us to reason about theoretical future world states.

The class `AIProcessor` implements this two-fold structure and is thus the key component of our Artificial Intelligence.

Special Action Thread

The special action thread manages and executes a queue of higher level actions implemented by extending the abstract `AbstractSpecialAction` class. Once the artificial intelligence is running, there is always one special action that is currently being executed, stored in `currentAction`. The `AIProcessor` collects all received perceptions and forwards them to the currently active special action and also stores them in a `HashMap` called `memory`, for easy look-up for a limited amount of time currently set to 600ms. The special action thread is running infinitely, always taking the first action out of the queue, initializing it and executing it. Whenever the queue is empty it automatically inserts an `IdleAction` that will be executed until the planner thread submits a new plan.

`AbstractSpecialAction` Every high level action or special action must extend the abstract class `AbstractSpecialAction`. Its most important method is the `performAction()` method which defines the behavior. Within this method, you have to control the agent by adding atomic commands like dash or turn. The method should be abortable; the class has a field aborted that can be set to true by calling abort(), which will typically happen when a new plan is issued that countermands the old plan. Typically, implementations of special actions should look somewhat like the following example:

```
public void performAction()
{
    while (!aborted)
    {
        // dash forward
        agent.addAction(Control.dash(100));
        // you must not submit more commands than 1 per 100ms
        sleep(SERVER_TICK_MS);
    }
}
```

Special actions have two means to react to perceptions, first via the memory `HashMap` storing the newest `VisualPerception` younger than 600ms for every ObjectName. Second, since `AbstractSpecialAction extends AbstractPerceptionProcessor`, you can override event handler methods such as `onBallSeen()` or `onPlayerSeen()`. Actions should also override the method `updateTo(AbstractSpecialAction newAction)` that is called whenever a new plan starts with an action of the same class as the currently executed action. When it is possible to update the action already running in a manner that fits the new plan, this method should do so and return true. For example this could be the case

when the current action is 'Move to a Position' and the new plan just differs in the actual position. If it is not possible, the method should return false. In that case the special action will be aborted by the planning thread.

When implementing special actions that react to perceptions, one has to keep in mind that new perceptions will be received with a lag. There are some pitfalls to avoid:

- Do not expect to have a new perception ready immediately after sending an action. You might want to lock until you receive a new perception via the event handling methods. See the `LookForBall` special action for ideas on how to do that.
- Do not react to the same perception twice. For example, when you try to run towards the ball but find that it is not right in front of you, you turn according to the angle you got with the last perception. After issuing the turn command there will not be a new perception available immediately – if you check the memory again you will still find the old one; however you must not issue another turn command based on the same perception. To get around this problm, you could for example store the server cycle of the last perception you used and only react if there is a perception newer than that.

Planner Thread
written by: Till Hartmann

The planner thread is responsible to generate new higher level plans. Unlike special actions, which are purely reactional, planning involves 'thinking' ahead and reasoning about possible futures.

We choose to view the problem of finding a good plan as a search problem: Beginning with the current state of the world or *situation*, there is a set of possible actions an agent can take, each of those leading to subsequent *hypothetical* situations. In this search tree, we choose the path of action that leads to the most desirable situation, e.g. one that results in a goal. The depth of those search trees is limited to a given maximum depth, currently 5. The evaluation of situations in the leaves of a tree is done heuristically. We continually update our current belief about the status of the world by processing perceptions. These perceptions are used to describe a situation for the planner using logics; symbols representing the perceptions are stored in a simple belief base we call `Situation`. A new plan is always computed based on the most recent situation.

Situations, `Symbols` and Uncertainty In our case, situations are of a predominately hypothetical nature. They describe the *expected* world model after executing a planned action but are also used to describe the current world model using a common set of logical symbols. This allows a STRIPS-like approach to situation modeling: If the preconditions of an action are fulfilled in a given situation, we can take an action which has an effect on the status of the world and thus leads to a different, future situation.

A Situation is a set of `Symbol`s, our interface for simple logical symbols. Symbols can be of different types: boolean, scalar or positions. Name and Value.

As we operate in a hypothetical, highly dynamic world, we can never be sure if a value is actually still 'valid' after a certain amount of time. Therefore, all classes extending `UncertainValue` need to be able to handle uncertainty. For example, an agent might see an enemy agent e at some position p at time t denoted by $p_e^{(t)}$ and knows that this enemy can move at most v units per time step. If this agent is not seen in the next $t+n$ time steps, he could roughly be anywhere in a circle around $p_e^{(t)}$ with radius $n \cdot v$. This combination of position and uncertainty is handled in `UncertainVector`. If the uncertainty grows too large, i.e. the circle in which the enemy's position might be is too large, it probably neither makes sense to use this specific information while planning nor to plan at all – you would rather need to start planning from scratch with fresh perceptions. For your own sub-classes of `UncertainValue` you must supply a suitable method to increase the uncertainty after a given number of server cycles.

Also note that we make the *closed world assumption*, that is, if a certain boolean symbol is not present in a given situation, we assume this symbol's value to be `false`[5].

Action The actions to get from one situation to a preceding situation are described by extending the `Action` class. When an action can be taken in a given Situation its `isApplicable()` method returns true. Applying the `execute(s)` method then yields the next hypothetical situation we expect to arrive after completing the action. Each action in plan space has a corresponding special action. The other way round, an Action models the expected effects of special actions on the symbols of our situations. Thus, an Action has to implement the `translateToSpecialAction()` method and return a corresponding special action.

Heuristic The main task for any heuristic is to evaluate a situation: that means, it is a function $h : \text{Situation} \to \mathbb{R}$ which assigns a value to a situation such that a situation s_1 which is deemed better than another Situation s_2 also gets a larger value $h(s_1) =: v_1$. A heuristic should be easy and fast to compute. Oftentimes one chooses the heuristic to be a linear combination of indicator functions that describe whether certain properties of a situation are fulfilled. This has two advantages: It is easy to compute and interpretable and you can automatically optimize it by adjusting the weights of the linear combination. A naive heuristic which favors scoring goals and ball possession would hence look similar to the following function:

$$h(s) = 2 \cdot \text{s.goalScored} + 1 \cdot \text{s.ballInTeamPossession}$$

Tree Search Now we have presented all the components necessary to describe the actual planning algorithm. As noted earlier, in a tree with nodes corresponding to situations and arcs corresponding to actions, we search for the leaf with highest heuristic value. We limit the depth of the search tree to 5, a depth for which trees can be analyzed in a reasonably fast manner. See figure 6.2 for a toy example search tee.

[5] see `checkBoolean()` code

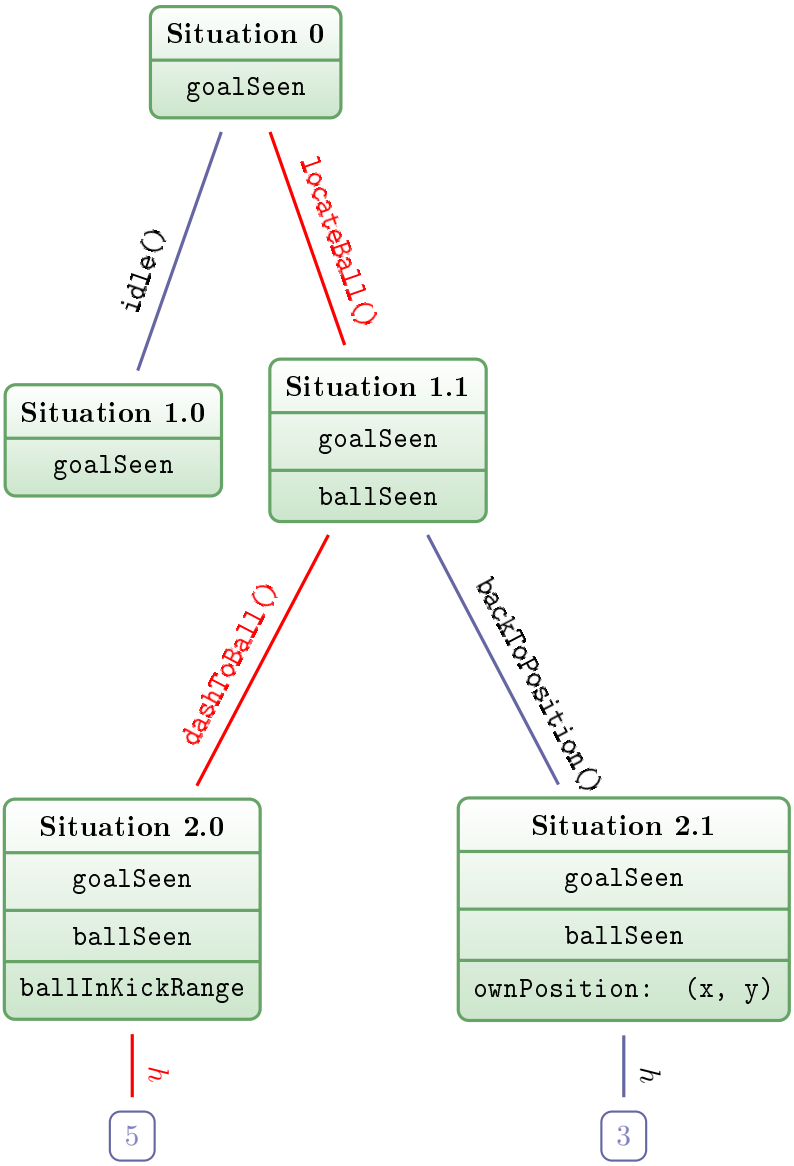


Figure 6.2: Based on the current Situation – Situation 0 – we derive possible future situations depending on the actions taken. In this example, the best plan according to the heuristic h is to look for the ball, then dash to the ball as it eventually allows us to kick the ball – possibly even score a goal. Please note that this figure only shows a subset of a search tree. Also note that situation 0 contains symbols derived from perceptions of the environment, i.e. contains 'non-hypothetical' symbols.

Artificial Intelligence

written by: Philipp Seifert

The following section gives an overview over our developed artificial intelligence, meaning what our heuristic looks like, what our special and plan actions do and what roles our agents are divided in.

Symbols The symbols we use to describe a purely hypothetical situation can be divided in roughly three categories: Boolean symbols, number symbols and position symbols. While the first category usually describes very basic symbols, such as `BallSeen` or `GoalScored` which can be either *True* or *False*, the latter two describe more comprehensive symbols, such as `EnemyXPosition` (where $X \in [1, 11]$) or `BallDistance`. Note that for example `EnemyXPosition` *does not* model the 'true' position of enemy X but rather its hypothetical (future) position. Together with the uncertainty parameter described in 6.2.2, this allows us to 'predict' an area where enemy X might very likely be located in the future. However, it might still be possible that `EnemyXPosition` does indeed model the 'true' position, if and only if the agent has recently *perceived* (i.e. seen) enemy X.

Special Actions Following are our atomic special actions.

Special Action	Description
DashToBall	This action reads the current ball position and distance from its memory and simply lets the agent move towards it at maximum power. If we get closer to the ball, we lower the dash speed to prevent stepping over the ball. We made several cases where we distinguish between different distances to the ball. That is so we don't already lower our speed at a greater distance.
DashToPosition	We get different coordinates of the game field and objects from the *SimpleLocalizationPerception* that we can use to dash to a specific given vector. By computing the distance and direction from our current position to this vector we convert this information into dash and turn commands.
Dribble	Simply kicks the ball a short distance towards a given direction if we are in kick range, otherwise we dash towards the ball. This behaviour imitates dribbling. Due to some noise when we are dashing we always check if we are in the correct direction to the ball first.
Goalie	An action specifically designed for the goal keeper. It simply performs a catch command if the ball is in range.
Idle	Does nothing.
KickBall	Takes a direction to kick and a distance to cover with the shot as parameters. E.g. if we want to cover a distance of 20 we would need to kick with a power of 40. But this is only true for a perfect kick. There are a few restrictions for a kick that lower the actual traveled distance, but which are calculated back in this action.
KickBallToEnemyGoal	Automatically reads the direction of the enemy goal from the memory and shoots with maximum power.
LookForBall	Tries to find the ball by doing two operations: first we turn only our neck and wait for following perceptions if we could locate the ball by doing this. If we weren't successful we turn our body and check if we could find the ball now, turning towards it.

Plan Actions As described in paragraph 6.2.2 every plan checks for a given situation if it is applicable, computes a hypothetical following situation and translates to a SpecialAction. Following is an overview of our plan actions.

Plan Action	**Applicable when**	**Situation changes**	**Translation**
DashToBall	This plan is applicable when the agent has seen the ball and the ball is in the agent's role's dash range (see *Roles* for further details).	The agent is in kick range of the ball.	Translates to a *DashToBall* special action.
DashToPosition	Always.	Sets the agent's position to the specified position.	Translate to a *DasToPosition* special action.
Dribble	This plan is applicable when 1. we are in kick range, 2. we know our own position, 3. we know the position of the enemy goal (because currently we will always try to dribble towards it) and 4. if we can find dribbling direction where we wouldn't directly dribble through an enemy.	None.	Translates to a *Dribble* special action.
KickToEnemyGoal	This plan is applicable when the agent is in kick range and knows the position of the enemy goal.	A goal is (probably) scored and the agent is no longer in kick range.	Translates to a *KickBallToEnemyGoal* special action.
KickToPosition	This plan is applicable when the agent is in kick range and knows its our own position.	The agent is no longer in kick range and the ball's new position is the specified position.	Translates to a *KickBall* special action.
LookForBall	This plan is applicable if the agent does not have knowledge of the balls position.	The agent has seen the ball.	Translates to a *LookForBall* special action.
PassToPlayer	This plan is applicable if the agent is in kick range, knows its own position and passing the ball to a given player is possible (there is no enemy standing in kick direction).	The agent is no longer in kick range but a teammate now has got the ball.	Translates to a *KickBall* special action.

Roles In soccer there are different roles in the team, what we used as an example to model our team. Each role can have its unique actions, heuristic and parameters, e.g. a defender may have other priorities than a goal scorer. Therefore every agent's AiProcessor must return an implementation of the *Role* interface, which has three methods *getDashMaxDistance()*, *getActionGenerator()* and *getHeuristic()*. An abstract implementation is given by *NaoRole*, which implements the two latter methods and we focus on first.

An *ActionGenerator* produces a list of actions suitable for the current situation, a standard implementation is given by *NaoActionGenerator*, which simply adds every currently modeled action to the list. Each role can have its own ActionGenerator, for example the goal keepers generator only generates an action that is meant to be executed by this role.

Also available for each role is a unique implementation of a heuristic, so situation evaluation may differ between the agents. A standard implementation is given by *NaoHeuristic*.

As said before the NaoRole is only abstract. The last method *getDashMaxDistance()* of the Role interface is implemented by different classes that represent typical soccer roles: *GoalKeeperRole*, *FullbackRole*, *MidfielderRole* and *StrikerRole*. This method is meant for controlling how far an agent may move to prevent e.g. a defender to move too far away from its initial position. Currently for a defender the method returns a value of 20, meaning the defender will only move towards the ball if it is away 20 distance units at maximum. At this time all of our roles actually only differ in different return values for this method (except the goal keeper).

Heuristic *written by: Lukas Pfahler and Till Hartmann*

As mentioned above, a simple way to obtain a sensible heuristic is using a linear combination of indicator functions. An indicator function returns 1 when its corresponding condition is fulfilled and 0 otherwise. A generalization could be to use functions that return values in the interval [0,1]. We propose the use of the following indicator functions:

- Whether we are in possession or kick-range of the ball or not
- Whether we are in goal kick range or not
- Whether we are closer to the ball than the enemy or not
- Whether we just scored a goal or not
- Whether we are in a good defensive position when the enemy has ball possession, or not

Obviously, the goal scored indicator function should have a very high weights whereas the defensive position functions should have smaller value.

6.2.3 Conclusion

We have put a lot of work into building a strong framework as a foundation for simulated multi agent robot soccer. While there has been some success this last year for the aspect of artificial intelligence, there is still a lot to be done based on our works.

Most importantly, improvements have to happen on the following frontiers:

- We have implemented a few basic special actions. However they do not work very reliable and suffer from synchronization issues. Most of the actions implemented so far are mostly for offensive play, future work should also strengthen the defensive play.
- Up to now, communication within the team has been mostly neglected. However, this can easily be achieved by using `say` commands. Note that spreading information in this way requires knowledge of the exact `say` semantics, which involves distance from the speaker etc. Communication should also be restricted to the necessary minimum, especially since the amount of `say` commands is heavily restricted to prohibit abuse. Needless to say, one should also consider which information to communicate at all.
- We have introduced a rudimentary set of symbols and modeled our special actions using this set. A more detailed model of the environment, particularly with respect to uncertainty, would certainly improve the performance of the planning component.
- Right now our heuristic mostly rewards scoring goals. A set of more sophisticated, role-dependent heuristics is essential to enhancing our Ai. When choosing a linear combination based heuristic, the weights should be adapted and optimized continuously during the game based on, for example, the success of past decisions.
- Even more rudimentary aspects of robot soccer simulation need more work: Most interactions with the referee still need to be implemented. Referee decisions lead to special game situations like kickoffs, free kicks, corner kicks, etc. which need special handling routines.

Criticism

While working on this project, initial enthusiasm quickly turned to frustration, as we struggled to get a grip on how *rcss2d*-architecture and -protocol are meant to work. The documentation of *rcss2d* server and protocol available on the internet seem to be outdated or not actively maintained and lack detail and depth, particularly regarding timing and synchronization between server and clients. It is our belief that the design choices with regard to abstractions from real world soccer are somewhat cumbersome. Also, unnecessary hurdles are placed in a new team's path to participating in *rcss2d*-matches. For example, halftime changeover does not make the simulation more interesting or demanding ai-wise but merely adds another avoidable layer of complexity to the implementation. Other details which only make implementation more difficult but do not further scientific insights into artificial intelligence and planning in a real-time, dynamic, continuous, competitive and

multi-agent environment are: Different wind-directions and -intensities, different player types, fouls, tackles.

We would advise simplifying and reducing the simulation complexity, thus allowing teams to focus on their research topic instead of wasting time working around major quirks of the *rcss2d* simulation. A step in this direction – at least in a *non*-multi-agent-system – is the so called *online coach* mode which allows controlling each player with an omniscient artificial intelligence. Similar simplifications should be applied to the multi agent setting, e.g. by adding a sensible synchronization mechanism[6] and perhaps even allow absolute positioning.

6.3 A Logic Programming Based Approach

written by: Stefan Rötner

We already noted that Robot Soccer is a typical example for a dynamic multi agent system. Oftentimes autonomous agents, especially if communication and collaboration with others are required, are modeled according to the Belief-Desire-Intention (BDI) Model. To address the goal of evaluating alternatives to the finite state automaton based AI approach utilized in the current *Nao Devils* framework, a first approach towards logic programming was made. In detail we considered a BDI agent with knowledge representation under the answer set semantics [11].

During the implementation we have come to the conclusion that answer set programming is not well suited for the implementation of low level actions like going to the ball, when the environment is changing as dynamically as in the setting of robot soccer. As a consequence we focus more efforts on the implementation of Special Actions and the planning component (resulting in the system described above) that can serve as foundation to a high level planning based on BDI agents in combination with logic programming instead of trying to model the behavior with explicit logic programs.

However the logic programming based approach is very promising with regard to very high level strategic decisions, which might be made by a coach robot or by an advanced soccer robot and with much lower frequency than induced by the fixed server cycle, which is required for successful low level actions like intercepting a ball. Because of this we will describe our approach including the underlying concepts and the used framework, with a special focus on the problems we encountered and the benefits logic programming and the BDI Model bring to the domain of robot soccer. Keep in mind that this section is not the description of a well performing AI-system but the documentation of our work as a reference for future high level planning.

6.3.1 Techniques

Before we start with the documentation of our approach we will give a short overview of the advantages that we hope to gain from using the selected techniques and explain some problems we encountered.

[6] there *is* a `synch_mode`, however, it is absolutely undocumented

BDI The BDI Model (see Figure 6.3 on the next page) is often used to model multi agent systems when collaborative action is needed, because different subgoals that need to be reached as part of an collaborative plan can easily be distributed as precise intentions to the different agents. BDI allows meta reasoning not only about the own plans but also about the beliefs and goals of other agents which might be an interesting option when it comes to predicting the behavior of enemy players. We choose the BDI model because it can be used to enable a goal directed behavior over a longer period of time in comparison to the reactive behavior of state automata.

Logic Programming Logic programming based behavior relies on rules that derive the next action from a world view that is modeled as a set of literals we consider to be true. In comparison to choosing the action based on a finite state automaton, intelligent behavior can be derived from an arbitrary combination of literals instead of relying on a predefined explicit behavior for each state. This allows the addition and deletion of rules during the game which gives rise to for example feedback based behavior optimization and meta reasoning about the own beliefs. Through the communication of rules and literals representing the current beliefs that can be integrated into the beliefbases of other agents logic programming can help to create collaborative behavior.

Non-monotonic Reasoning We decided to use answer set programming (ASP) [11] because default negation allows us to formulate behavior generating rules in the presence of uncertainty. Our Agent has do deal with uncertainty with regard to noisy sensor data and incomplete knowledge of the world as it is impossible to keep all the players and the ball in view at all times.

Problems Calculating the answer sets of an ASP-program is a computationally complex and thus time-consuming task. We used an efficient standard solver but still the generation of answer sets and the overhead for passing a logic program to an external solver are not compatible to the short tick size of the simulation server. Because of this we believe that a purely reactive behavior is needed at least for the realization of some atomic Special Actions. Before such a lightweight system to provide the building blocks for complex plans is available logic programming is not applicable. We have created the system described in the previous section to fulfill exactly this need.
Another problem of the used framework is that it lacks an appropriate planning component. However we provide a short list of existing planning components that could be used at the end of this section.

6.3.2 The BDI Model

written by: Philipp Seifert

The BDI model is a software architecture that relies on works of the philosophy professor Michael Bratman, Stanford University [12, 13, 14]. He was engaged in human decision making and studied the influence of intentions on it. Anand Rao and Michael Georgeff adapted his model for the programming of intelligent agents.

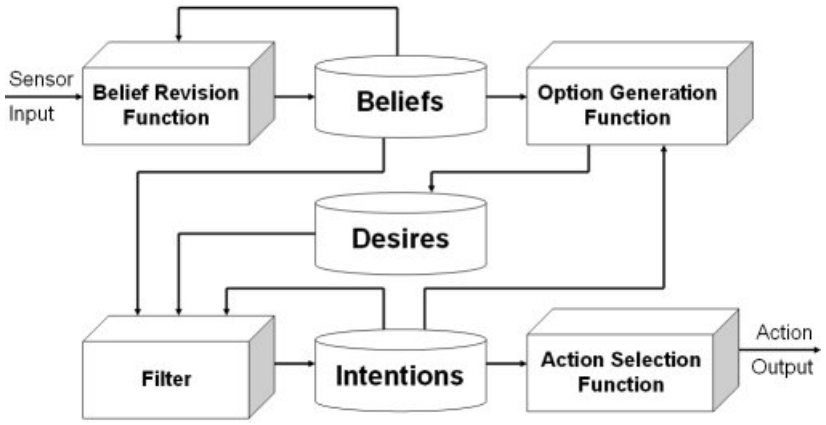


Figure 6.3: The components of the abstract BDI model.

In contrast to reactive agents, that make decisions instantly based on their perceptions, this deliberative approach allows agents to be autonomic as far as possible. In particular, the BDI model is composed of two steps. The first is *deliberation* (i.e. which goals shall be achieved) and the second is *means-end reasoning* (how can the selected goals be achieved). Therefore each agent has three mental attitudes:

- *Beliefs*: They represent the environment and contain not only the modeled view of the agent on the world, but also its internal condition and background knowledge that may be important for implications.
- *Desires*: Based on the beliefs of the agent they contain the goals an agent would like to achieve in the environment.
- *Intentions*: They represent the deliberative state of the agent – what the agent has chosen to do. In implemented systems, this means the agent has begun executing a plan [7].

Figure 6.3 shows the seven main components of a BDI agent. Just like any other agent the process starts with a perceptual sensor input. The *Belief Revision Function* takes this and the current *beliefs* as input and determines a new set of beliefs. Afterwards the *Option Generation Function* computes the options (also called desires) available to the agent, based on his beliefs and intentions. The *Filter* represents the agent's deliberation process and determines the agent's intentions on the basis of its current beliefs, desires, and intentions. Last the *Action Selection Function* selects an atomic action based on the current intentions and outputs it to the environment.

6.3.3 Angerona

written by: Stefan Rötner

In order to provide a better understanding of the implemented multi agent system it is important to describe on a conceptual level the existing frameworks we used and explain

[7] A plan is a sequence of actions that the agent can perform to achieve one or more intentions

why they were chosen.

In search of an existing framework for multi agent systems (supporting the BDI Model as well as logic based representation of the agent's knowledge) JADEX [15] and Jason [16] were considered. With regard to our claim of using non monotonic reasoning preferably under the answer set semantics our final choice was *Angerona* [17], which is currently developed at TU Dortmund. The Graphical User Interface of Angerona (see Figure 6.4) provides a simulation log and allows to track the changes in the agents' beliefbases. For a detailed description of the framework and its implementation we recommend the technical report and the manual. Thus we only give a short summary of the most relevant aspects and our adaptations:

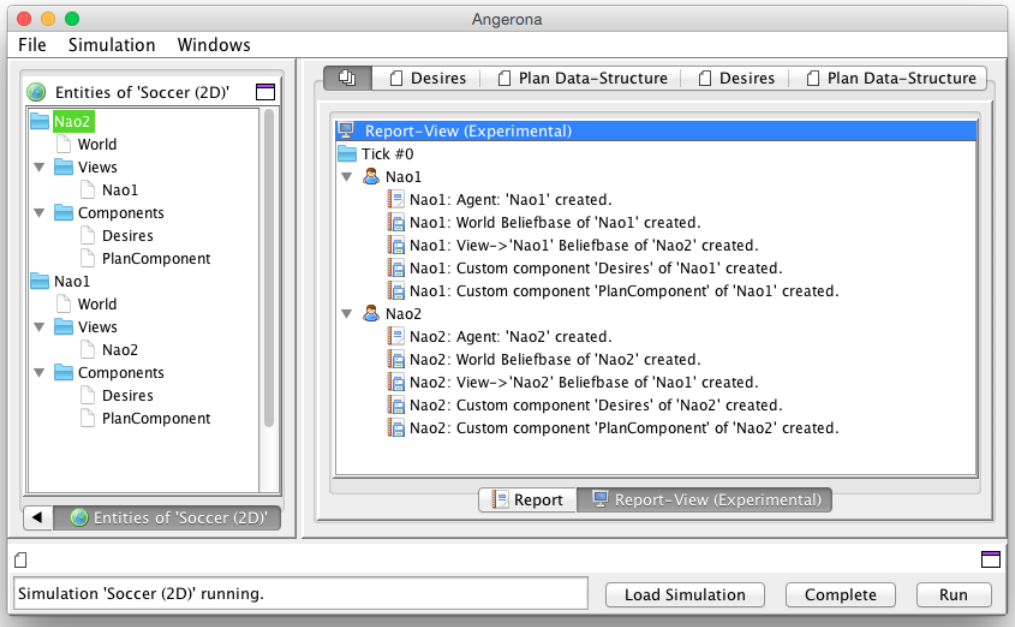


Figure 6.4: The Graphical User Interface (GUI) of Angerona logs all actions and perceptions in combination with additional information about each step in each reasoning cycle. Using the tree view on the left the epistemic components and their changes during the simulation can be displayed. The *run*-button in the bottom left corner allows a step-by-step simulation in which the simulation is paused after each reasoning cycle. In this way the GUI can be used for debugging purposes.

Agent In Angerona the epistemic [8] state of an agent and its functional component are separated. For our implementation the epistemic state consists of a belief base, the agent's desires and its intentions. The functional component is realized through a number of stateless operators which can access and manipulate the information stored in the data components of its epistemic state. The operators we used, correspond to the BDI related functions *update beliefs*, *generate options*, *intention update* and *subgoal generation*.

[8] relating to knowledge or cognition

Environment The environment from which the agents receive their perceptions can be customized by implementing a new environment behavior. Usually the environment is responsible for collecting the actions each agent executes and to distribute the changes those actions result in via perceptions to every agent. We altered the agent implementation to let each agent receive perceptions directly from the simulation server. The environment is only used for agent instantiation and communication between agents.

Communication Angerona supports communication accordingly to the FIPA standard for agent communication [18]. Human communication is imitated by distinguishing different *speech acts* like *inform*, *query* and *answer*. The semantics for each speech act are defined in a format similar to STRIPS [19] rules (i.e. rules are defined by pre- und post-condition of their execution). Information is exchanged by encapsulating the literals used for modeling the knowledge into one of the speech acts. Angerona is treating speech acts as perceptions and actions at the same time, so each query is one atomic action that can be executed at the end of one reasoning cycle. As a consequence the answer is not available during the same tick (i.e. a singular execution of the reasoning cycle). This is the reason why it might be useful not to rely entirely on speech act communication but rather use a form of communication that is more efficient for our purposes.

Belief Base The main reason for choosing Angerona as underlying agent framework was a fully implemented ASP belief base which is based on the Tweety library [20] for knowledge representation. Tweety provides parser, reasoning mechanisms and inference operators for a number of different logic languages which are all based on the same interfaces for syntactical elements like atoms, literals, negation, et cetera. Thus switching to another logic language should be possible without many adaptations.
For answer set programming the update of the belief base can be executed using different forms of revision in order to keep the belief base consistent. The reasoning is realized by passing the logic program representing the agent's beliefs to an answer set solver (at the moment we are using DLV [21]) which calculates the answer sets.

Angerona can easily be customized or extended since *Maven* [9] is used as a build system and new operators or components can be integrated using the *Java Simple Plugin Framework* [10]. Our soccer specific operators, perceptions and an environment for communication with the simulation server are grouped in a plugin we named *NaoAgent*.

6.3.4 Realization

written by: Philipp Seifert

Due to the abstractness of the general BDI Model we needed to detail the individual BDI components for the application domain. As mentioned in the beginning of the chapter the biggest problem was soccer itself, as there are countless possibilities of situations that we could encounter and therefore may need to react to. To begin with we focused on elementary actions like kick, pass and tackle and wanted to implement more complex

[9] https://maven.apache.org/
[10] https://code.google.com/p/jspf/

game plays at a later point, but before we came to that we decided to focus on a new different approach, which reasons are discussed in 6.3.1.

Soccer BDI Components

To better understand each component in following there is described a single BDI cycle, which begins with a perception and ends with an atomic action.

Input As we were working with virtual agents and not with physical robots we had no sensors in the classical way, meaning no cameras. The *SimLeagueAgent* supplies each agent with different kinds of perceptions (see 6.2 for a detailed view).
As we focused on basic behaviour intelligence most of the perceptions were ignored, because we had no necessity to react on them. For example perceptions telling us the current stamina level of the virtual agent were simply thrown away. We were mostly focused on implementing reactions on a *VisualPerception* and *NaoPerception*.

Belief Revision Function & Beliefs The Belief Revision Function is the first functional component of our BDI model and therefore the interface to communicate with the environment. When a new perception arrives its task is to determine the new epistemic state of the agent, meaning updating his *beliefs*. It is also the interface for communication with other agents, but due to problems with the delay of messages in Angerona we couldn't make use of this.
In general the new information contained in either the perception or communication are translated into a chosen knowledge representation, in our case into ASP (answer set programming), what is directly supported by Angerona. All the logical rules we wrote are therefore simple ASP rules.

As mentioned earlier we didn't respect every perception. Our first approach was to create our epistemic state based on the *VisualPerceptions* we got. For example if it told us that we have seen the ball in a distance of 40 and a direction of 30 radiance we stored these values in a separate map with unique keys.

After storing these values we invoked the revision of some rules we defined and added to the ASP program at initialization, which finally update our epistemic state. An example rule is shown in algorithm 1. It takes the stored value $BALL\_DISTANCE$ as a input and tries to revise the head. It is equivalent to the ASP rule

$$inKickRange := BALL\_DISTANCE <= 0\ 7. \tag{6.1}$$

The threshold 0.7 is the kick range of an agent in the 2D simulation league. The rule is read as 'if the distance to the ball is less than or equals the distance required to kick the ball, then add the string *inKickRange* to our epistemic state'. This string indicates that we believe that we are able to kick the ball. Algorithm **??** shows the implementation of the rule.

Algorithm 1 Rule to determine if we are in kick range

Require: double BALL_DISTANCE

```
1: Predicate head = new Predicate("inKickRange");
2: Predicate body = new Comparative(<=, BALL_DISTANCE, 0.7);
3: Rule rule = new Rule(head, body);
4: aspProgramm.addRule(rule);
```

At the end we decided to refactor these rules to the SimLeagueAgent, as it is not really the BDI models duty to evaluate them in the Belief Revision Function. Now, a NaoPerception tells us that we have to add "inKickRange" to our beliefs. We simply translate this to ASP as a fact and add it to our logical program. We adopted this procedure in our current Special Actions approach.

Option Generation Function & Desires In time five different desires were elaborated that an agent may want to achieve. These are:

- *LOOK_FOR_BALL* - The agent doesn't know the position of the ball and desires to look for it.
- *GO_TO_BALL_CLOSE* - The agent is near the ball and desires to walk to the ball.
- *GO_TO_BALL_FAR_AWAY* - The agent is far away from the ball and desires to rush to the ball.
- *DRIBBLE* - The agent desires to dribble forward.
- *GOAL_KICK* - The agent desire to score a goal.

The desires are reasoned based on the recently updated *beliefs* that the Option Generation Function got as an input. The first desire *LOOK_FOR_BALL* has highest priority. Without having information about the ball it is hard to plan anything in soccer obviously.

The two desires *GO_TO_BALL_CLOSE* and *GO_TO_BALL_FAR_AWAY* are desirable if the a predicate called *ballPosKnown* can be reasoned. The difference between them is the first one implies a lower walk speed towards the ball while the latter implies running to the ball at maximum speed. We distinguished them because we had cases where we were already close to the ball, but not in kick range, then told our agent to rush to the ball with maximum speed and the agent stepped over the ball, standing behind it. So for complexity reason we introduced a second desire where the agent would walk slowly to the ball.
Also this knowledge helped us implementing the special action *GoToBall* in our new approach.

The desire *DRIBBLE* is desirable when the predicate *inKickRange* is reasonable. The last desire *GOAL_KICK* is desirable when the predicates *inKickRange* and *goalScorable* are reasonable. The latter one is revised when the agent's distance to the goal is below a specific threshold.

Deliberation The deliberation function has the purpose to select a *desire* and mark it as a *goal*. It is part of the earlier mentioned filter process. In the classical BDI model all desires are considered equivalent and it is the programmers job to find a the best solution how it should be determined what desire is marked as a goal. For various reasons we chose a very simple way to deal with this problem: the desires are marked as goals through a fixed order $LOOK\_FOR\_BALL > GOAL\_KICK > DRIBBLE > GO\_TO\_BALL\_CLOSE > GO\_TO\_BALL\_FAR\_AWAY$.

Planner Also part of the earlier mentioned filter process is the planner component. Its purpose is to select a plan for the current *goal* from a database. The selected plan is then called an *intention*. We only managed to create a single plan for every desire, so our planner component didn't have many choices what plan to select. It was also unclear how a plan should be selected.

Our intentions consist of atomic actions. E.g. the plan for the goal $GO\_TO\_BALL\_CLOSE$ is

1. Turn to the ball, if the angle to it is not 0.
2. Go to the ball.

These atomic actions are directly interpretable by the SimLeagueAgent and therefore can directly be forwarded to the server.

Action Selection Function The Action Selection Function picks the next atomic step from the current *intention* or if there is no current intention it picks the next intention from a stack, doing nothing if there is no next intention.

6.4 Planning

written by: Stefan Rötner

One key question in creating intelligent agents is how the agent's plans are created. In the current version one of the special actions is selected as a simple plan accordingly to explicit Java code in the *sub goal generation operator* while ASP is only used to store the current beliefs and to reason about the world. In order to benefit from the possibility of creating complex plans from logic programs an appropriate planning component is needed. We did some research on how such a planning component could be realized and came up with the following list:

Know-How For plan generation Angerona provides the Know-How component. Know-How aims at enabling the Agent to reason about his own planning process by providing the information of how to fulfill his desires in *Know-How statements*. Each of these statements is a tuple of a *target* (referring to the intention that is fulfilled by the specified course of action), a number of *sub-targets* that need to be achieved in order to reach the target, and some conditions defining whether the statement should be considered at all. From all applicable statements for the current desires an *intention tree* is formed where the children of a statement refer to the applicable sub-targets. A more detailed description and the theoretical foundations can be found in [22].

With regard to the problems we are currently facing with Angerona some alternative approaches are discussed:

Jason Jason[16] is another framework for multi agent systems supporting the BDI model. Agent behavior is modeled using AgentSpeak [23] (an implementation of the *Practical Reasoning System*). Jason could be a good alternative if the time required per tick can not be decreased, but does not support non monotonic reasoning and default negation.

A-Prolog using Event Calculus As a reference for plan generation using ASP semantics there exist some implementations of A-Prolog. DLV (i.e. the solver we already use for reasoning about our beliefs) features the plan front end *DLV-k*. In [24] *Baral and Gelfond* describe an action language as extension to ASP that can be translated into standard ASP rules. *Mora et al.* define in [25] a similar language extension based on *event calculus*, with special regard to realizing the different aspects of the BDI model. This approach has been succesfully used, e.g. for implementing the Reaction Control System of the Space Shuttle [26].

6.5 Conclusion

written by: Philipp Seifert One of the main reasons why we draw off of this approach was the very slow decision making of Angerona. On a system with four CPUs, each clocked at 2,2 Ghz, the framework needed roughly 500ms for a single decision based on a perception. This was absolutely inacceptable for a real time soccer game, as the simulation server ticks are every 100ms. A simulation could be made with the previously described techniques, but it was not competitive at all.

Furthermore we saw two more critical modelling problems. As ASP is used to solve complex logical programs we needed to create some basic actions based on pure reactive behaviour first. Also the framework's planning component seemed poorly conceived. As a single BDI cycle can be part of a bigger plan to achieve a specific goal, implementing this was not very intuitive in Angerona. So at the end we decided that a better multi agent system would be modelled with a different approach.

Chapter 7

Extended Behavioral Networks

written by: Piotr Szczotka

7.1 Behavior networks in our domain

Estimating a good behavior in a given situation within a dynamic, continuous, non-deterministic and competitive scenario such as RoboCup is a challenging task. Such decisions have to be made throughout the game. Doing so is demanding for which many aspects have to be considered simultaneously. As the world changes by itself, both friendly and adversary forces are involved and most actions do not guarantee a positive outcome a sophisticated mechanism is required.

First of all, some actions can have some effects that are not deterministic. Shooting the ball towards the enemy's goal does not always guarantee scoring a point. Running towards a ball does not always guarantee obtaining the ball control. It can even result a Nao falling down. Therefore, one should follow strategy to make a decision on the basis of the best expected outcome but never perform an action which is more likely to be followed by an undesired outcome than a desired outcome. Doing nothing is sometimes the right thing to do. Nonetheless, an action which has some negative impacts is not always a bad choice.

A convenient and detailed model of a current situation is of great significance. A Nao perceives the world through sensors. Making a decision considering raw sensor values would not be an elegant solution, thus it would require a strong coupling between hardware and the deciding mechanism. The logical step is to extract a mathematical model of a current situation from the raw values. Given the mathematical model one may advance further with the abstraction. Deciding whether to shoot is easier given Boolean variables like "nearToBall" and "nearToEnemysGoal" rather than numerical values. However, describing the game situation by the means of Boolean logic may not be precise enough and cause a great information loss. The use of fuzzy logic makes sense at this point as it handles partial truths, continuous states, and uncertainties. Moreover, it provides a loose coupling from the rest of the software and hardware. A mechanism which provides a fuzzy logical model of a situation should not merely transform a single numerical value into a single fuzzy logic value, but also should be able to transform many numerical values into a single fuzzy logical value. That way it is possible to express more complex ideas i.e. how critical a

given situation is. A detailed overview on this topic can be found in this report in Chapter 5.

Furthermore, the goals that are pursued during a game have different priorities. The most important goals in a RoboCup scenario are to score a point, not to loose any points and to harm neither yourself nor others during the process. There are also some other minor goals that can be taken into consideration. Moreover, the priorities may change over time according to the current situation.

In addition, some behaviors can be performed only under given circumstances. For instance running towards the ball is only available when a robot has not fallen down. Nevertheless, it also makes sense to estimate the benefit of behaviors even when they are unavailable. Naturally an unavailable behavior will not be executed even if it is theoretically beneficial. However, a substantial potential benefit of an unavailable behavior should be a motivation to enable the behavior.

An ideal behavior would be a behavior that is likely to fulfill many goals. Although such behaviors are always considered as beneficial, they are seldom available. In order to shoot a ball towards enemy's goal one needs to control the ball and stand close enough to enemy's goal. If it is not the case one can get the ball and dribble towards enemy goal in order to fulfill those two condition. But in order to get the ball one needs to know where the ball is. So in some cases one needs to find the ball first. But if a robot has fallen down, it will not be able to search the ball until it has got up. Some actions that are required during the process do not fulfill the objective of scoring a goal directly. They merely enable other behaviors. Even though one should try to achieve a goal directly it is often required to take some additional steps which indirectly contribute towards winning the game. The process can be accelerated by luck with no effort, so one can skip some steps. However, it is more likely that the adversary will interfere the process, so one has to make a step back in order to progress further.

The sensors of a Nao are not absolutely reliable. Because of this, both the mathematical model and the fuzzy logical model are inevitably inaccurate and outdated to some extent. Moreover, the limitations of the sensors may cause rapid changes in a robot's internal model which do not occur accordingly to the environmental changes. Bad decisions and a robot changing its mind without any good reason and are the result. Sticking to a good plan, even if it is not the best plan, is in many cases better than formulating a new plan driven by some sensor noise. A robot steadily changing his mind will lack productivity in a long run.

On the one hand the modeler has a clear expectation, what a Nao should be doing in some given case. On the other hand, in the vast majority of cases, the best reaction is not always known to the modeler of the artificial intelligence. A robot should be proactive and produce clever ideas on its own, maybe even surprising its modeler by its smartness.

Furthermore, the artificial intelligence should be extendable and adjustable. First of all a possibility of optimization over parameters should be given. Secondly implementing new routines or improving some routines should have impact on the decision mechanism but should not require a complete remodeling of the behavior. Keeping the artificial intelligence

in accordance with the rest of the software and the hardware, which both may change over time, should be simple.

In addition, some actions can be performed concurrently thus they engage different resources. For instance a robot can run, look for the ball and communicate with other robots simultaneously, as the routines require legs (sometimes the whole body), cameras and means of communication respectively. Moreover, different type of resources are valued differently. Engaging some resources may require a high expected outcome. Furthermore, some resources can be lost or won throughout a single game (i.e. battery level).

Finally, one should keep in mind that the RoboCup is a game. Optimal decisions may acquire game theoretical computations like estimating the dominant strategy or computing a Nash equilibrium in mixed strategies.

Behavior networks convey the impression to be an adequate model to cope with aspects listed above. They typically consist of goals, competence modules, sensors, resources and parameters. However, there many concrete models that handle things differently. The vast majority of time has been devoted to models that have been successfully applied in the RoboCub domain - REASM Networks and Extended Behavior Networks, which can be seen as an extension of REASM Networks. Those networks have been introduced by Dorer in [27, 28]. Note that REASM is also an extended version of MASM network introduced in 1989 by Maes in [29, 30, 31]. In our project the EBN model has been extended to the so called nEBeN model.

7.2 REASM

A REASM behavior network is the most basic network which has been used in our domain. A precise and detailed definition can be found in [27]. A synopsis can be found below. The entire chapter is based on [27].

A REASM behavior network consists of goals and competence modules and parameters. In order function it needs merely a fuzzy logical model of the current state. By the means of a declared triangle norm and t-conorm fuzzy logical formulas can be computed and used by the model. Those fuzzy logical formulas are used to express conditions, relevances or executability.

Every single goal has its both static and dynamic relevance and a goal condition. The static importance is just a number between 0 and 1. The dynamic relevance varies according to the current situation and is expressed as a fuzzy logical formula. This formula can contain variables, negations, disjunctions and conjunctions. The function f unites the dynamic and the static value.

Competence modules hold the information about what can be done, under what circumstances it can be done and what the consequences are for engaging in a given behavior. The availability of an action is represented as a conjunction of signed fuzzy logical variables. The consequences are expressed as a conjunction of signed fuzzy logical variables. Additionally, one declares a probability for every single effect (every single signed fuzzy logical variable in the conjunction). Every competence module has both its availability

and utility. These two value are merged into one by a function called h. A module which is both beneficial and executable is likely to being proposed by a network.

In order to estimate whether a competence module is available one needs to compute merely one fuzzy logical formula. However, it is more difficult to estimate if the execution of a competence module is beneficial. It is the case when a module is likely to:

- fulfill a currently important goal
- enable a useful and unavailable module
- not to inhibit an important goal
- not to inhibit an available and useful competence module

In order to compute this preconditions and postcondition of competences modules and goal conditions are considered.

The utility of a given behavior, represented by an activation of the corresponding competence module, is computed in a process called activation spreading. The process is not explicitly recursive.

Nevertheless, after repeating the process several times one can see the recursive nature of it. It means among other things that if a behavior x that is likely to enable an unavailable behavior y which can enable another unavailable behavior z which is likely to fulfill an important goal, the behavior x will be considered beneficial over time. In comparison to other values the activation changes slowly during the computations as it is influenced by the past.

The action selection process proposes the best behavior if it exceeds the threshold. The threshold decreases if a robot remains idle but resets to its original value after every execution of a behavior. The threshold's original value has to set up with caution as both activity (and thus the function h) are limited. Setting the threshold too high would cause idle waiting after every execution.

One can imagine goals and competence modules as nodes in a graph. The preconditions and postconditions of competence modules and goal conditions define implicitly the directed edges of the graph. The activation flows though those edges from goals to competence modules and from competence modules to other competence modules. For instance if a competence module has an effect that is likely to partly or entirely fulfill a goal condition the competence module will receive activation from the goal. The extent of the impact depends on the probability and the goal's static and dynamic relevance.

Apart from the structure there are also many parameters that can be adjusted in order to achieve a desired behavior. These parameters are activation of modules, inhibition of modules, interia of activation, activation threshold and threshold decay. Technically speaking there are also functions that influence the behavior but are not a part of a network's definition. They are t-norm, t-conorm, sigma, h and f.

A REASM behavior network can be seen below.

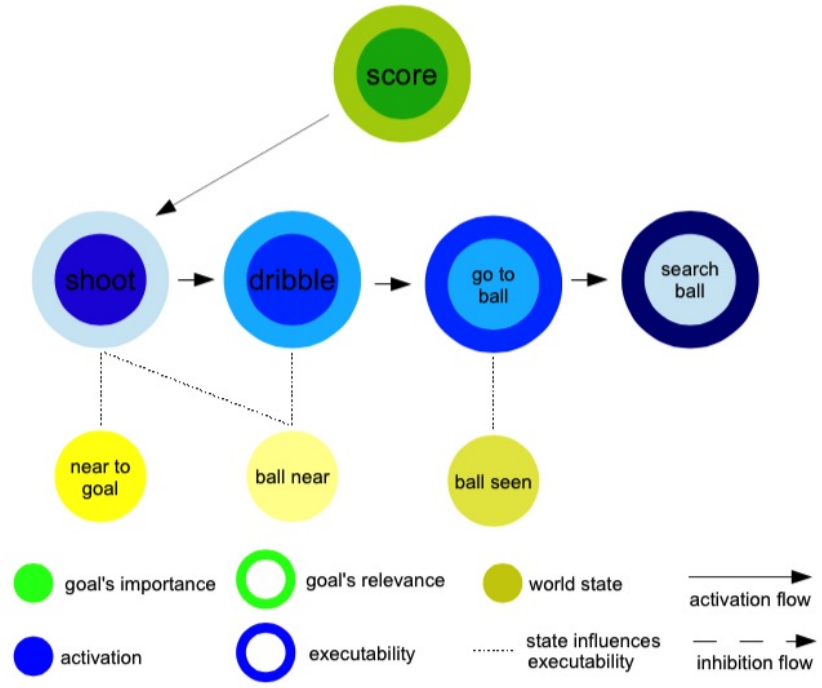


Figure 7.1: An example of a graphical representation of a REASM behavior network

7.3 Extended behavior networks

Extended behavior networks are an extension of REASM. They additionally provide the possibility to execute many behaviors if they engage different resources. The network is extended by resource nodes and competence modules are extended by definition of the resources a given behavior requires. The model has been proposed in [28]. A synopsis which is based on [28] can be found in this chapter.

Similarly to the REASM model, the EBN uses a fuzzy logical model of the current state. Additionally, it requires a state of resources. These resources are temporary engaged by the competence modules. Furthermore, they can be lost or gained during the simulation. However, in our models it was seldom the case. Every resources nodes has its amount, so that a resource can be used by many competence modules concurrently if the amount is sufficient. Competence modules that are seen as more beneficial are favored when the amount of a resource is insufficient to execute all the behaviors. [28]

The action selection mechanism has been also modified. The is no global threshold. Every resource possesses its own threshold which is decreased and reset to its original value after it has been used. Different resources can have different threshold. Some resources are engaged only if a high expected outcome is likely to occur. [28]

The action selection mechanism proposes a single behavior, many behaviors or nothing. The benefits and availability of behaviors and their overall suitability for execution are computed as in REASM model. [28]

Obviously the Naos are able to do things simultaneously so that this model was of our big interest. However, there some problems that are likely to occur while using the model.

First of all, handling doing two things when the multitasking ability is limited is rather hard. A Nao may be able to do two things simultaneously but the outcome is not always as good as if it did the things separately. Moving head while running may result decreasing the overall speed. This can be obviously overcome by defining two competence modules – like running engaging head and running without engaging head. However, doing the trick too often makes the abstraction of resources unnecessary which is only good when there is a true multitasking ability can be assumed. If this is seldom the case one can try to use the REASM model. One can always make a power set from the set of basic behaviors ignoring the empty set. Then one could delete the together non-executable behaviors and merge the valid sets into new competence modules. The postconditions of new competence modules can be defined accordingly to the expected outcome. If there are few resources and not many things that can be done concurrently, the amount of the competence modules is not likely to increase exponentially.

Second of all, the declaration of required resources of every single behavior makes the structure of the network strongly dependable on how the given behaviors are executed. It makes the modeling process difficult.

Third of all, it is more difficult to comprehend the computations while there are many thresholds involved.

7.4 nEBeN

During our project a wish to extend the REASM or EBN has arisen. Even before testing out the model in the real game play, some problems have been already identified. Those problems have been listed in the section issues. Some of those problems, but not every single of those, can be dealt with proper modeling practices. It was not indented to create a tailor-made model for our domain. However, as the experience has shown, introducing a single concept to given behavior network may open many doors.

In order to separate strategic decisions and tactic decisions or to differentiate between restrictions that originate from game's rules and restrictions that originate from technical limitations a model named nEBeN is proposed. It is strongly grounded on the EBN model. A network can delegate an action to another network. That way networks of behavior networks or even networks of networks of behavior networks (and so on) can be formed.

Definition 1 *A nEBeN is an EBN as defined in [28] except for: A competence module's behavior b is either a primitive behavior or a behavior proposed by a nEBeN during the action selection process. Pre, Post, res and a of a delegating competence module regard the delegation process and not the behavior proposed by the other network. Running into a cycle while deciding leads to an undefined behavior.*

The behaviors which are suggested by a nEBeN are only primitive behaviors and are computed in a recursive manner. These can be behaviors originating from different nEBeNs.

In most cases every network of a nEBeN should operate on the same (or at least consistent) world model and resource model.

One has to keep in mind that there are additional bridges to be crossed while delegating a decision, because every network possesses its thresholds (i.e one needs encouragement to play football, but even he is already playing football there has to be an additional encouragement to perform specific actions like moving towards ball).

The individual networks of a nEBeN can be run synchronously or asynchronously. Running the networks asynchronously may cause a potential risk of executing an outdated behavior.

By definition every individual network posses its own parameters P. It also possesses its own t-norm, t-co-norm, sigma, h and f function.

7.5 Specialized networks in nEBeN

A behavior network finds and executes a good behavior during the process of activation spreading. It is encouraged to execute a behavior that is under current circumstances likely to achieve an important goal or/and enable an unavailable action that is able to achieve a goal (or enable an action that can enable an action and so on). Moreover, behaviors that do the opposite are avoided. It is a sophisticated mechanism, thus it reduces (more or less successfully) the modeling process to declaring what goals under what circumstances are important and what behaviors under what conditions are available and what is their expected impact.

However it is difficult to model every aspect of behavior by the means of encouragement and discouragement, especially when the modeler wants the network to act under given circumstances in a given way. What if one wants to use an if-statement, randomness or a sequence of ordered behaviors (elementary or proposed by other networks)? The extension of the EBN, the nEBeN, introduced the concept of delegation. A single nEBeN/EBN provides enough means to realize so called specialized networks. The idea is to combine them with regular behavior networks via the delegation. That way sophisticated and simple ideas can be combined together. The specialized networks are listed below.

- `decision` The network delegates according to a fuzzy boolean expressions (and, or, not available – technical realization per relevance conditions)
- `ritual` A number of actions are executed via delegation in a given order.
- `dice` A behavior is chosen randomly according to a given probability distribution.
- `concrete behavior` A given action is proposed in any case. (more subtle modeling can be done later)
- `empty network` None action is proposed in any case.
- `specification` The decision to act has been made. The network does not possess its own threshold – it only specifies the details how it should be done.
- `delay` The network delays the process of delegation.

7.6 Optimization

In [27] the author reveals some hints how to optimize the parameters P. The process is difficult since the parameters are not independent from each other. In order to optimize the parameters one can let the team play against a team where the parameters are set to be static. The strength of a team can be determined by the goal difference. It has been found out that setting the parameter gamma to low will cause the robots to remain passive since the threshold is never or seldom exceeded. On the other hand making a robot too reactive will decrease the overall quality of play slightly.

A supervisor of the group has conceptualized a new optimization approach.

The problem in our case is that the expectation values for every signed fuzzy logical variable in a competence module's postcondition are undiscovered. In more common words, one does not know, how good the routines, in terms of expected benefit, are. Determining the postconditions is much harder than determining the preconditions.

In order to deal with that once could start with values that appear convenient and to continuously adjust the expectations based on the actual results whenever a behavior is executed. More recent experiences should have more impact on the mechanism. However, a naive implementation of the idea may cause some problems.

First of all, it is possible that an objectively good behavior fails during the process. Adjusting ex according to those experiences causes that the behavior is no more executed. If it is the case the competence module would not be able to redeem its good reputation.

Second of all, the expectations are vague by their nature. It is not clear how to extract their adjustment after every execution of a given competence module since it is not known and not declared when the expected effect is supposed to occur.

Third of all, the postconditions of the competence modules define in a way the dynamic nature of the world. It seems convenient that a model should act better if it has a realistic world model. But due to the limitations of the model (listed in chapter "Issues") the things are not so simple. A less realistic model could under some circumstances act better than a more realistic one. Moreover, insertion of new activation and inhibition edges during the process, because of some marginal dependencies are found, may produce a very dense network.

Nevertheless, the approach is very interesting and could severely simplify the modeling process. Moreover, it could improve the play of the robots to a large extent. Some issues could be overcome by other mechanism or simply ignored.

As a nEBeN has a hierarchical nature, one could optimize a nEBeN network by network, instead of trying to optimize the whole model at once. While optimizing a nEBeN network one could follow a buttom-up approach, since the networks of lower levels are not influenced by the higher networks. The networks of higher levels are influenced by the lower networks though.

7.7 Code

The REASM, EBN and nEBeN model have been implemented in C++. There is a guide for every implementation, so that the integration process can be simplified. The simplified UML class diagram of the REASM implementation can be seen below. The implementations EBN and nEBeN are based on the implementation of REASM. A concrete network is defined in a XML data. The program builds a given network accordingly to the definition. A nEBeN network consists of many files, as every file contains a network. One loads only the first network manually and other referenced network are loaded and build automatically. The definition of goals, competence modules and resources only defines the structure of the network implicitly, so there are some calculations required before the network is ready for simulation. The functions t-norm, t-conorm, sigma, h and f are not contained in the XML data since they are technically not a part of a network. They are implemented in code, but can be easily modified. In the nEBeN model one can define those functions differently for every network. By default, the suggested values from [27] are used. The parameter upper bound for module's activation is contained in a XML data. However, it has no influence on any computations. This value should remind the modeler not to set the activation threshold too high. If the activation threshold is set too high an inevitable waiting after every execution are the result. Since it is not known what states of the world are feasible, this value cannot be estimated by computer at this point. The network is build by a builder class (not shown in the diagram below). Although the code is relatively complex, the interface is rather simple. One provides the current state in a string like "ballNear=0.9 overheating=0.3" and gets a string which contains the proposed behavior. Working with EBN or nEBeNs requires additionally the definition of resources and declaring when they are lost or gained. The result string in that case is a string which may include many behaviors.

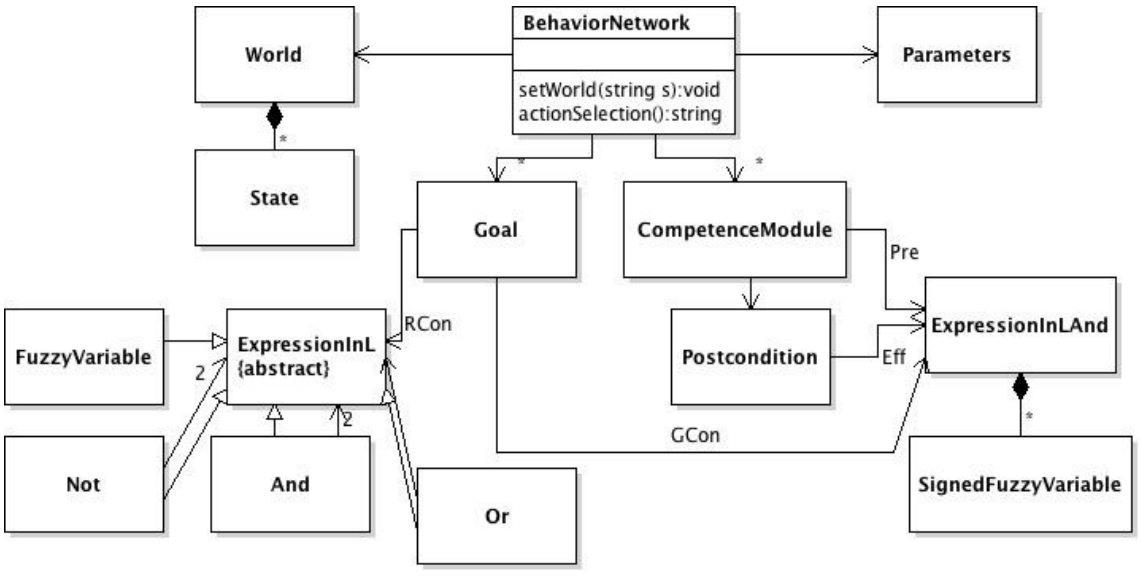


Figure 7.2: The simplified UML class diagram of the implementation of the REASM behavior network

7.8 Simulator

A simulator which is strongly based on the code has been programmed for every model. By the means of the simulator one can examine the performance of a given behavior network before running it on a robot. That way some potential problems can be identified by debugging the network step-by-step. Moreover, by the means of the simulator one can comprehend all the complex computations, so a modeler can gain a deeper understanding of the model. Additionally, the simulator estimates all the activity flow relations so that the structure which is implicitly defined by the logical terms is shown. An example of how to use it can be seen below.

Consider a following scenario. A robot wants to impress the audience with his juggling skills (especially when there are many people in the audience). He also does not want to get bitten by a bee (especially when the bee is near to him). Juggling with three balls requires two hands. Other actions can be performed one handed. The robot can also juggle with two balls. It impresses the audience less as if the robot was juggling with three balls though. He can chase the bee away but in that case he is likely to drop one ball but he can pick up the ball. The corresponding EBN behavior network can be seen below.

One can examine the behavior of the network with the simulator using commands such as:

- `load theJugglerAndTheBee.xml` loads, builds and initializes the network
- `activation spreading` performs activation spreading without proposing a behavior
- `R hands=2` declares the resources (the command can be called repeatedly if the resources are gained or lost)
- `peopleThere=0.7 beeThere=0.9 threeBallsInPossession=1` provides the network a fuzzy logic description of the current situation (the command is called repeatedly during the simulation)
- `action selection` proposes action(s) according to the current situation (returns something like "pickUpTheThirdBall chaseAwayTheBee")
- `view` shows the network

Please note that the state "threeBallsInPossession" should be always represented by a binary value and the resource "hands" should be represented by an integer value.

7.9 Integration

A mechanism that controls information flow between the rest of the code and the behavior network is required.

First of all, a network should be provided with a current world model and current resource model. The resource model is in our case straightforward since the resources are only

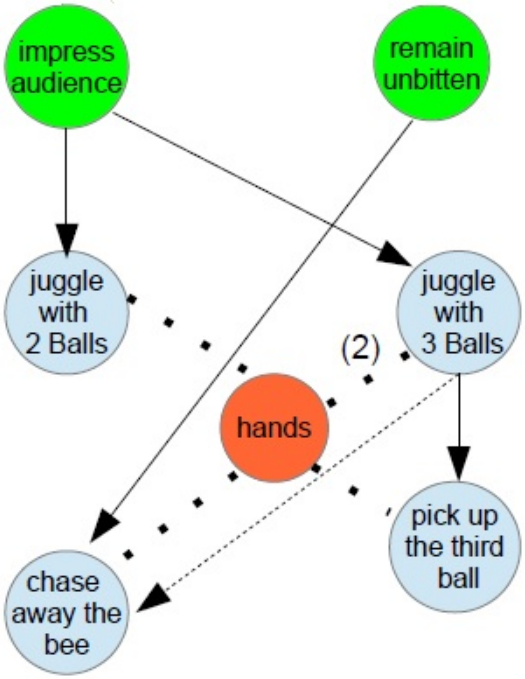


Figure 7.3: The juggler and the bee EBN

engaged and disengaged instead of being lost or won. One needs to declare the resources merely at the beginning.

Extracting a fuzzy logical model out of a mathematical model is rather hard. The process also depends strongly on the implementation of the routines. Improving the shooting routine so that a Nao can shoot further would influence a fuzzy logical variable that expresses whether a Nao is in range of a shot on goal. In the mathematical model nothing changes because of the new routine. But the way the distance is being transformed to the fuzzy logical variable has to modified. Moreover, the fuzzy logical model may contain some more complex ideas. It could take many parameters into account and express how critical a given situation is.

Furthermore, the code should call the routines which are proposed by a behavior network. Since the models are easily exchangeable by the redefinition of the XML data, the modeler can sometimes forget about safety concerns. In order to protect a Nao from overheating or damaging itself it makes sense to stop the model under some critical circumstances. Nevertheless, the safety aspects should be taken into account by a behavior network itself.

In order to identify some issues a logging mechanism should be also a part of the controller.

Finally, the controller must deal with situations when a robot executes a routine and cannot be interrupted (i.e. standing up after a fall). In such cases one can provide the Nao with the current world model and call activation spreading instead of action selection. That way the mechanism reacts according to the changes but does not propose new behaviors.

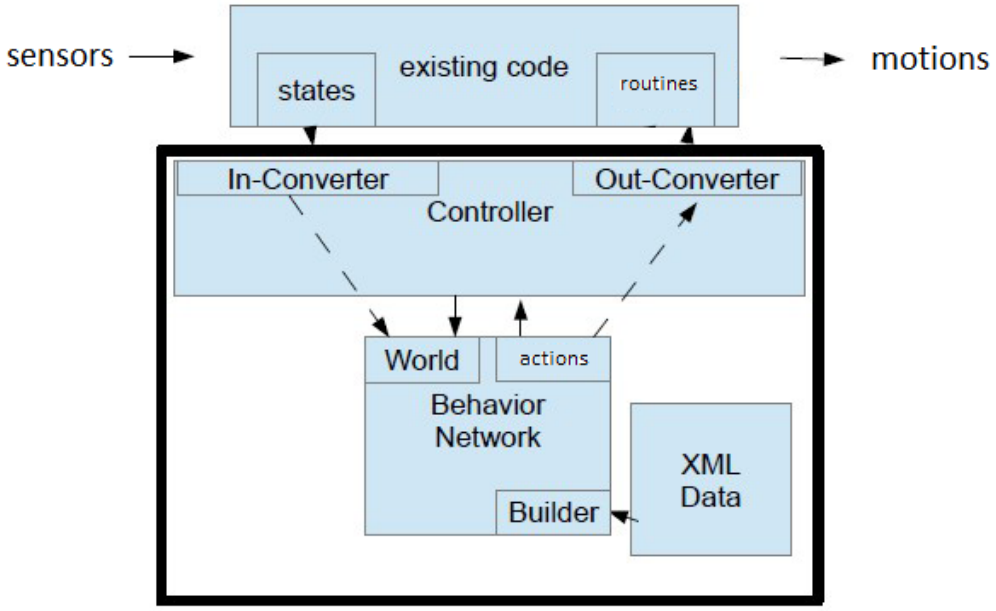


Figure 7.4: An abstract view on integration

7.10 Issues

A game of football has some states like ordinary play, penalty kick or free kick. Certain actions even if they are technically possible should be executed only in certain states. Handling it by the means of encouragement and discouragement does not seem to be a convenient idea as the instructions of the referee should be unconditionally followed. If one wishes to model the entire behavior by behavior networks one could use the nEBeN model to do so. Alternatively one can use an existent artificial intelligence and use a behavior network like REASM or EBN to merely for the ordinary play.

Some decisions in the game of football have to be random by nature in order to be optimal [32]. If there some game theoretical decisions that have to made throughout the game, like deciding in which corner to shoot the ball, behavior networks do not seem to be a right choice to deal with it. At a low level these decisions can be made by routines itself. However, even randomness on a higher level can be achieved by the means of specialized networks in the nEBeN model.

The REASM behavior network and the Extended Behavior Networks successfully deal with some negative characteristics that other similar models have [27]. However, there are some problems with the model that a modeler has to take into account while modeling a network. A white box approach to the modeling based on simple declaration what state is desired and what, under what circumstances can be done may fail. These issues are listed below and also apply to REASM, Extended Behavior Networks and nEBeN.

First of all, long paths of activation flow in a given behavior network indeed influence the network while estimating the best behaviors. However, they are strongly disadvantaged

alone due to their length even when the probabilities along the path are high. One should choose the sigma transfer function and activation threshold wisely and dimension the network right, so that such long paths have a proper impact on a behavior network.

Second of all, once there are two possible ways to achieve a goal, there are some situations where choosing one of the path is objectively better than choosing the other. Sometimes the network can favor the wrong path. Since the activation flow is distributed on every node and the computations are difficult to comprehend, it can be hard to encourage a network to prefer the right path over the wrong path at given circumstances without influencing the rest of the network's activation flow. Any change to a good functioning network, like adding new competence modules, may interfere the rest of the mechanism.

Finally, it is possible that a network will try to fulfill a goal or enable a behavior by executing a wrong behavior. That is the case, because it is not accurately analyzed why a condition (a precondition of a competence modules or goal conditions) is not fulfilled. Every condition is a conjunction of signed fuzzy logic variables. Once a goal is important or a behavior is likely to be beneficial and is currently unavailable the mechanism will try to fulfill all of those condition variables in a conjunction in an equal measure – independently of how true or false a single signed fuzzy variable already is.

Chapter 8

Path-Planning

written by: Sebastian Engels

Another theme in the behavior-part is the path-planning (also called motion planning). It is important that the robot not only can run safely and quickly, or pursue a good tactic as a team, but also that the positions specified by the behavior are underway sense. Should the robot get from its current position to a certain point on the field, so it is not enough if the *Nao* just walks linearly to that point. During the run, the *Nao* must recognize other robots as obstacles and handle them, never entered the penalty area and should be guided in a meaningful direction. It's task of the path planning to consider all these points and to calculate the best possible path.

This is especially important if the robot moves to the ball to start a dribbling, to pass or to make a shot on goal, since the robot must have a matching distance and the correct orientation to the ball. Till now, the robot moved curved to the ball and adjusted its orientation during the walk, so that he has the correct angle to the ball afterwards. This behavior is shown in figure 8.2.

A problem with this approach is, that sometimes the robot has not the correct orientation to the ball and therefore stands slightly displaced. This then means that the robot does not hit the ball correctly and the shot fails. The reason for this is that the robot does not rotate far enough during the run. To avoid this, we realize method, in which the robot moves in a straight line to the ball and then moves with side steps around the ball until it is in the direction of the opponent's goal. This approach could be already observed at other teams. This approach is illustrated in figure **??**.

The target angle α_{target} for the orientation of the robot can be calculated using the current ball position pos_{ball} and the goal position $pos_{opp-goal}$ of the opposing team:

$$\alpha_{target} = \arctan(pos_{opp-goal} - pos_{ball}). \tag{8.1}$$

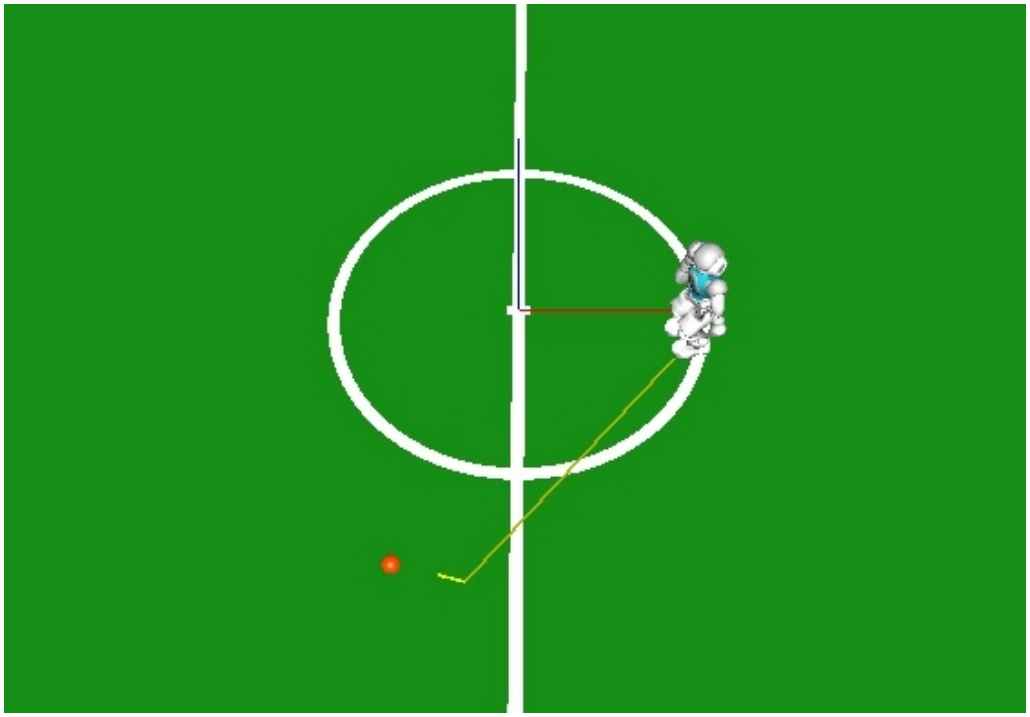


Figure 8.1: The robot adjusts its orientation during the walk to the ball

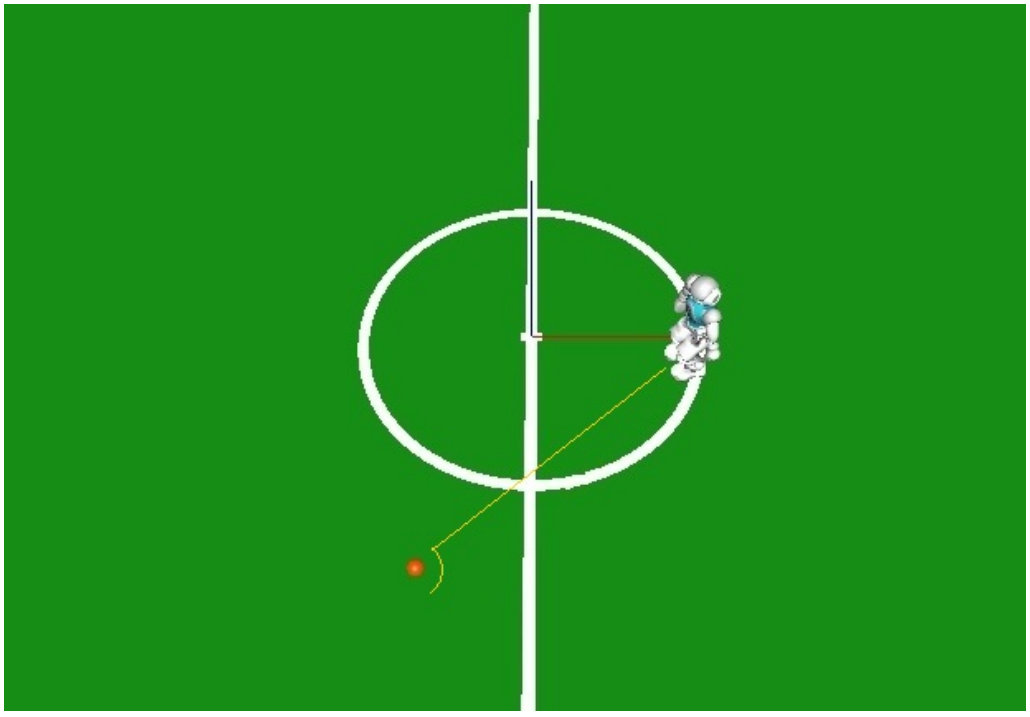


Figure 8.2: Adjustment of the orientation on the ball

From the resulting target angle α_{target} the angular deviation α_{dev} can be calculated with the current orientation α_{nao} of the *Nao* :

$$\alpha_{dev} = \alpha_{target} - \alpha_{nao}. \tag{8.2}$$

Based on the sign of the angular deviation the direction in which the robot must make the side steps can be distinguished (positive deviation: steps to the right, negative deviation: steps to the left).

To investigate the method more accurately, games against a team with the previous implementation were simulated. Above all the linear motion to the ball is a problem with this approach. The robot turns on the spot until he looks up to the ball and starts running. As a result, considerable time is wasted and the robot reaches the ball later. Furthermore, the direction must often be corrected, causing the robots slow down. However, it was gratifying that the robot was always aligned correct (in the direction of the opposing goal) when the robot was on the ball.

To avoid the mentioned problem with the linear movement a combination with the previous implementation is conceivable. The robot moves to the ball as shown in figure 8.2 and then uses the lateral steps to fix any orientation errors.

Part III

Conclusion

written by: Till Hartmann

Regarding motion, we mainly struggle with noisy and inaccurate data (not to mention hardware-wear of the robots) which makes finding accurate and stable criteria (e.g. for stability of walking) rather difficult; however, we were both able to rule out some approaches which do not work in this context (see PCA) and find promising approaches.

One of these approaches is the stability criterion using the x-acceleration sensor. We were able to implement a speed clipping using this criterion. However, the parameters have to be tweaked in the future to improve the functionality.

We also provide a module for walk calibration resp. optimization for integration in the NAO-framework so that different (online) optimization routines can be easily supplied and tested.

As the framework which is used in Dortmund lacks of a proper strong kick, the so called *LongKick* was implemented. We ran out of time due to many problems and side-effects that were time-consuming to fix. The progress is well advanced, but mainly optimizations and fine-tuning are missing.

Regarding behaviour, we provide

- FuzzyLab
- Multi-Agent Behavior in a Simulated Soccer Environment
- Extended Behavior Network / REASM

FuzzyLab is a system for classification of game situations via fuzzy logics[1]. This helps making decisions based on crisp and noisy input data more stable and less error-prone and will probably be used in the behaviour for rcss2d.

We have written a software framework for developing artificial intelligences that participate in the rcss2d soccer simulation league. Using this framework, we first tried to develop an Ai based on *Angerona* using the design principles of BDI, unfortunately it turned out that using a system like that is not suitable for realtime decision-making, however it might be a good choice for making high-level strategic decisions. We then focussed on developing a faster system that models the planning process as a search problem in a world of states described by simple logical formulas. We developed basic special actions that describe elementary actions an agent can perform on the soccer field like looking for the ball or shooting. We then modeled the effects of performing those actions in logical formulas and were thus able to reason about their effects, ultimately allowing us to plan and 'think' ahead. We believe that this is a powerful approach that can also be applied on the real *Nao*-robots.

Yet another approach to modeling agent behavior are behavior networks which try to enable agents to select multiple (possibly concurrent) actions to achieve some kind of goal or at least to enable unavailable actions that are likely to achieve a currently important goal. The

[1] actually a complete fuzzy logic implementation with controllers etc

behavior networks work with a fuzzy logical world model and make decisions on the basis of expected utility. The REASM behavior network and its extension Extended Behavior Network have been implemented. Extended Behavior Networks have been extended to a model called nEBeN which has been also implemented.

Both motion optimization and behaviour modeling struggle with uncertain and/or noisy data. While the motion team tries to tries to cope with that by applying filters to reduce noise or finding stable criteria, the behaviour team considers using fuzzy logics in order to derive less jumpy input data.

Part IV

Appendix

List of Figures

Bibliography

[1] Aldebaran-Robotics: *NAO H25*, 2014. http://doc.aldebaran.com/1-14/family/nao_h25/index_h25.html.

[2] Aldebaran-Robotics: *Inertial unit*, 2014. http://doc.aldebaran.com/1-14/family/robots/inertial_robot.html.

[3] Aldebaran-Robotics: *FSRs*, 2014. http://doc.aldebaran.com/1-14/family/robots/fsr_robot.html.

[4] Smith, Steven W.: *The Scientist and Engineer's Guide to Digital Signal Processing*. Elsevier Ltd, Oxford, 3rd Revised edition edition, 2002.

[5] Meister, Andreas and C. Vömel: *Numerik linearer Gleichungssysteme*. Vieweg Verlag, Friedr, & Sohn Verlagsgesellschaft mbH, 2011.

[6] Kuckartz, U., S. Rädiker, T. Ebert and J. Schehl: *Statistik: Eine verständliche Einführung*. VS Verlag für Sozialwissenschaften, 2013.

[7] Czarnetzki, Stefan, Sören Kerner, Oliver Urbann, Matthias Hofmann, Sven Stumm and Ingmar Schwarz: *Nao Devils Dortmund Team Report 2010*. Technical Report, Robotics Research Institute, TU Dortmund University, 2010.

[8] Zadeh, L.A.: *Fuzzy sets*. Information and Control, 8(3):338 – 353, 1965.

[9] Kahlert, J.: *Fuzzy Control für Ingenieure*. Fuzzy Control für Ingenieure: Analyse, Synthese und Optimierung von Fuzzy-Regelungssystemen. Vieweg, Braunschweig/Wiesbaden, 1995.

[10] Bosko, B.: *Fuzzy Thinking: the new science of fuzzy logic*. Hyperion, New York, 1993.

[11] Lifschitz, Vladimir: *Action Languages, Answer Sets, and Planning*. In Apt, KrzysztofR., VictorW. Marek, Mirek Truszczynski and DavidS. Warren (editors): *The Logic Programming Paradigm*, Artificial Intelligence, pages 357–373. Springer Berlin Heidelberg, 1999.

[12] Bratman, Michael E.: *Intention, Plans and Practical Reason*. CSLI Publications, Stanford, 1999.

[13] Weiss, Gerhard: *Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence*. MIT Press, Cambridge, 1999.

[14] M. Georgeff, B. Pollack, M. Tambe & M. Wolldridge: *The belief-desire intention model of agency.* Proceedings of the 5th International Workshop on Intelligent Agents V: Agent Theories, Architectures and Languages, 1555:1–10, 1998.

[15] Braubach, Lars, Alexander Pokahr and Winfried Lamersdorf: *Jadex: A BDI Agent System Combining Middleware and Reasoning, Chapter of Software Agent-Based Applications, Platforms and Development Kits.* Birkhäuser Book, 2005.

[16] Bordini, Rafael H., Jomi F. Hübner and Renata Vieira: *Jason and the Golden Fleece of Agent-Oriented Programming.* In Bordini, Rafael H., Mehdi Dastani, Jürgen Dix and Amal El Fallah Seghrouchni (editors): *Multi-Agent Programming Languages, Platforms and Applications*, volume 15 of *Multiagent Systems, Artificial Societies, and Simulated Organizations*, chapter 1, pages 3–37. Kluwer Academic Publishers, 2005.

[17] Krümpelmann, Patrick, Tim Janus and Gabriele Kern-Isberner: *Angerona - A Multiagent Framework for Logic Based Agents.* Technical Report, Technische Universität Dortmund, Department of Computer Science, 2014.

[18] Intelligent Physical Agents, Foundation for: *FIPA ACL Message Structure Specification*, 12 2002.

[19] Fikes, Richard E. and Nils J. Nilsson: *STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving.* In *Proceedings of the 2Nd International Joint Conference on Artificial Intelligence*, IJCAI'71, pages 608–620, San Francisco, CA, USA, 1971. Morgan Kaufmann Publishers Inc.

[20] Thimm, Matthias: *Tweety - A Comprehensive Collection of Java Libraries for Logical Aspects of Artificial Intelligence and Knowledge Representation.* In *Proceedings of the 14th International Conference on Principles of Knowledge Representation and Reasoning (KR'14)*, July 2014.

[21] Leone, Nicola, Gerald Pfeifer, Wolfgang Faber, Thomas Eiter, Georg Gottlob, Simona Perri and Francesco Scarcello: *The DLV system for knowledge representation and reasoning.* ACM Transactions on Computational Logic (TOCL), 7(3):499–562, 2006.

[22] Thimm, Matthias and Patrick Krümpelmann: *Know-how for Motivated BDI Agents.* In *Proceedings of The 8th International Conference on Autonomous Agents and Multiagent Systems - Volume 2*, AAMAS '09, pages 1143–1144, Richland, SC, 2009. International Foundation for Autonomous Agents and Multiagent Systems.

[23] Rao, Anand S.: *AgentSpeak(L): BDI Agents Speak Out in a Logical Computable Language.* In *Proceedings of the 7th European Workshop on Modelling Autonomous Agents in a Multi-Agent World (MAAMAW'06)*, 1996.

[24] Baral, Chitta and Michael Gelfond: *Reasoning agents in dynamic domains*, pages 257–279. Kluwer Academic Publishers, Norwell, MA, USA, 2000.

[25] Móra, Michael da Costa, José Gabriel Pereira Lopes, Rosa Maria Vicari and Helder Coelho: *BDI Models and Systems: Bridging the Gap.* In *Proceedings of the 5th International Workshop on Intelligent Agents V, Agent Theories, Architectures,*

and Languages (ATAL'98), ATAL '98, pages 11–27, London, UK, UK, 1999. Springer-Verlag.

[26] Watson, Richard: *An Application of Action Theory to the Space Shuttle.* In *Proceedings of the First International Workshop on Practical Aspects of Declarative Languages*, PADL '99, pages 290–304, London, UK, UK, 1998. Springer-Verlag.

[27] Dorer, Klaus: *Behavior Networks for Continuous Domains using Situation-Dependent Motivations.* In *In Proc. 16th Int. Joint Conf. on Artificial Intelligence (IJCAI*, pages 1233–1238. Morgan Kaufmann, 1999.

[28] Dorer, Klaus: *Extended Behavior Networks for Behavior Selection in Dynamic and Continuous Domains*, 2004.

[29] Maes, Pattie: *The Dynamics of Action Selection.* In *Proceedings of the 11th International Joint Conference on Artificial Intelligence - Volume 2*, IJCAI'89, pages 991–997, San Francisco, CA, USA, 1989. Morgan Kaufmann Publishers Inc.

[30] Maes, Pattie: *Situated Agents Can Have Goals.* Robot. Auton. Syst., 6(1-2):49–70, June 1990.

[31] Varela, F.J. and P. Bourgine: *Toward a Practice of Autonomous Systems: Proceedings of the First European Conference on Artificial Life.* Bradford Bks. MIT Press, 1992.

[32] Palacios-Huerta, I.: *Beautiful Game Theory: How Soccer Can Help Economics.* Princeton University Press, 2014.