technische universität
dortmund

Nao Devils Dortmund

Team Report 2010

Stefan Czarnetzki, Sören Kerner, Oliver Urbann,
Matthias Hofmann, Sven Stumm, Ingmar Schwarz

Robotics Research Institute
Section Information Technology

Contents

Chapter 1

Introduction

Competitions such as the RoboCup provide a benchmark for the state of the art in the field of autonomous mobile robots and provide researchers with a standardized setup to compare their research. Additionally the RoboCup *Standard Platform League* does not only provide researchers with a common setup, but also with the same hardware platform to use. This renders increased importance to publications of those teams, since extensive documentation and especially releasing source code allows other researchers to compare results and methods, reuse and improve them, and to further common research goals.

In the course of this report some of the points of the robot software and current the research approach of the RoboCup team *Nao Devils* are described. An overview about the Nao Devils software is given in section 1.2. This software has been used in the competitions in 2010 and is included in the code release that is published along with this report[1]. This code release contains the complete code used by team *Nao Devils* during the RoboCup 2010. Only the behavior and the tuned walking parameters are replaced by a more basic version. Also included are the developed tools, out of which the behavior debug tool is described in detail in section 4.4.

The following chapter of the document will describe the motion control process, emphasizing the *Dortmund Walking Engine*, which has been the first closed loop walking engine applied to the Nao in RoboCup 2008 and 2009. The version of RoboCup 2010 has been further developed and tuned to reach walking speeds of up to $44cm/s$, which is, to our knowledge, the highest speed yet achieved with the robot Nao. Incidentally this speed exceeds the "theoretical maximum walking speed" given by Aldebaran in an earlier specification by almost 50%. Chapter 3 focuses on the perception processes of the Nao, while section 4 presents the behavior specification including extensions to the XABSL specification of previous years. In each of those chapters 2, 3 and 4 the solutions currently in use are described in detail and further references supplied when appropriate, but also the current research is presented. This includes concepts that were already successfully applied and evaluated, but haven't found their way into the competition code for RoboCup 2010 for various reasons, as well as concepts currently under development. Chapter 5 summarizes those current research topics and the current development process for 2011.

The appendix provides further information and tutorials about how to set up the

[1] `http://www.irf.tu-dortmund.de/nao-devils/download/2010/NDD-CodeRelease2010.zip`

development environment and the robot (appendix A), use the given software framework (appendix B), and about the parametrization of the presented walking engine (appendix C).

1.1 Team Description

The *Nao Devils Dortmund* are a RoboCup team by the Robotics Research Institute of TU Dortmund University participating in the *Standard Platform League* since 2009 [1] as the successor of team BreDoBrothers, which was a cooperation of the University of Bremen and the TU Dortmund University [2]. The team consists of numerous undergraduate students as well as researchers. The senior team members of *Nao Devils Dortmund* have already been part of the teams *Microsoft Hellhounds* [3] (and therefore part of the *German Team* [4]), *DoH! Bots* [5] and *BreDoBrothers*.

Participation in the GermanOpen 2009 and RoboCup 2009 resulted in 3rd place each, and the second round robin pool has been reached in RoboCup 2010.The Team also participated in all *technical challenges* in these years reaching the 3rd and 10th place. Besides official RoboCup competitions the *Nao Devils* regularly participate at other international events such as the *Festival della Creatività* in Florence, Italy, the *RoboCup Exhibition and Engagement Event* in Eisteddfod of Wales, UK, and the *Athens Digital Week* in Greece. It is also planned to take part in the *Standard Platform League* of RoboCup 2011.

1.2 Software Overview

The software package used by team *Nao Devils* consists of a robotic framework, a simulator and different additional tools.

The framework, running on the Nao itself, is based on the German Team Framework [6]. The latest version, released by team *B-Human*[2], was used as a basic structure in 2010, replacing the motion, vision and behavior modules by team *Nao Devils'* own versions (see appendix B.3 for details about the origin of each module). The framework communicates with *NaoQi* using the *libBHuman*, completely separating it from Aldebaran's software modules. Support for Microsoft Visual Studio 2008 is included, using a cross compiler to generate native code for the Linux running on the Nao. For a short introduction and overview on the framework see appendix B. A more into depth description can be found in [7].

To test developments in simulation, the software *SimRobot* was used instead of commercial alternatives, such as *Webots* from Cyberbotics[3]. Being open source offers great advantage, allowing to adapt the code to own developments. In addition having the feature to directly connect to the robot and debug online (see appendix A.2.4) is very convenient during development. SimRobot [8] is a kinematic robotics simulator developed in Bremen which (like Webots) utilizes the Open Dynamics Engine[4] (ODE) to approximate solid state physics. Using update steps of up to 1 kHz for the physics engine enabled the possibility of realistic simulated walking experiments closely matching the gait of the

[2] `http://www.b-human.de/file_download/27/bhuman09_coderelease.tar.bz2`
[3] `http://www.cyberbotics.com/`
[4] `http://www.ode.org/`

real robot. It also features realistic camera image generation including effects like motion blurring, rolling shutter, etc.

Since *B-Human's* team report 2009 [7] covers the basics and usage of the simulator in great depth, a detailed description in exclude from this report. For further details please refer to [7] chapter 8.

To visualize behavior the adapted XABSL editor of the German Team as well as the Java reimplementation [5], done by team *Nao Team Humbolt* are used. Since debugging behavior running on the real robot can be really difficult to comprehend, team *Nao Devils* developed a XABSL debug tool. A logging mechanism records all XABSL decisions online during gameplay. With help of the XABSL debug tool, developed by team *Nao Devils*, these logs can be combined with a video file, to analyze and replay robots decisions. A detailed description of this tool can be found in section 4.4.

[5] `http://www.naoteamhumboldt.de/projects/xabsleditor/`

Chapter 2

Motion

The main challenge of humanoid robotics certainly are the various aspects of motion generation and biped walking. Dortmund has participated in the Humanoid Kid-Size League during Robocup 2007 as *DoH! Bots* [5] and before in RoboCup 2006 as the joint team *BreDoBrothers* together with Bremen University. Hence there has already been some experience in the research area of two-legged walking even before participating in the Nao Standard Platform League of 2008 as the rejoined *BreDoBrothers*.

The kinematic structure of the Nao has some special characteristics that make it stand out from other humanoid robot platforms. Aldebaran Robotics implemented the HipYawPitch joints using only one servo motor. This fact links both joints and thereby makes it impossible to move one of the two without moving the other. Hence the kinematic chains of both legs are coupled. In addition both joints are tilted by 45 degrees. These structural differences to the humanoid robots used in previous years in the Humanoid League result in an unusual workspace of the feet. Therefore existing movement concepts had to be adjusted or redeveloped from scratch. The leg motion is realized by an inverse kinematic calculated with the help of analytical methods for the stance leg. The swinging leg end position is then calculated with the constraint of the HipYawPitch joint needed for the stance foot. This closed form solution to the inverse kinematic problem for the Nao has been developed in Dortmund and used since RoboCup 2008 when other teams as well as Aldebaran themselves still used iterative approximations.

2.1 Walking

In the past different walking engines have been developed following the concept of static trajectories. The parameters of these precalculated trajectories are optimized with algorithms of the research field of *Computational Intelligence*. This allows a special adaption to the used robot hardware and environmental conditions. Approaches to move two legged robots with the help of predefined foot trajectories are common in the Humanoid Kid-Size League and offer good results. Nonetheless with such algorithms directly incorporating sensor feedback is much less intuitive. Sensing and reacting to external disturbances however is essential in robot soccer. During a game these disturbances come inevitably in the form of different ground-friction areas or bulges of the carpet. Additionally contacts with other players or the ball are partly unpreventable and result in external forces acting on

the body of the robot.

To avoid regular recalibration and repeated parameter optimization the walking algorithm should also be robust against systematic deviations from its internal model. Trajectory based walking approaches often need to be tweaked to perform optimally on each real robot. But some parameters of this robot are subject to change during the lifetime of a robot or even during a game of soccer. The reasons could manifold for instance as joint decalibration, wearout of the mechanical structure or thermic drift of the servo due to heating. Recalibrating for each such occurrence costs much time at best and is simply not possible in many situations.

The robot Nao comes equipped with the wide range of sensors capable of measuring forces acting on the body, namely an accelerometer, gyroscope and force sensors in the feet. To overcome the drawback of a static trajectory playback, team Nao Devils developed a walking engine capable of generating online dynamically stable walking patterns with the use of sensor feedback.

2.1.1 Dortmund Walking Engine

A common way to determine and ensure the stability of the robot utilizes the zero moment point (ZMP) [9]. The ZMP is the point on the ground where the tipping moment acting on the robot, due to gravity and inertia forces, equals zero. Therefore the ZMP has to be inside the support polygon for a stable walk, since an uncompensated tipping moment results in instability and fall. This requirement can be addressed in two ways.

On the one hand, it is possible to measure an approximated ZMP with the acceleration sensors of the Nao by using equations 2.1 and 2.2 [10]. Then the position of the approximated ZMP on the floor is (p_x, p_y). Note that this ZMP can be outside the support polygon and therefore follows the concept of the fictitious ZMP.

$$p_x = x - \frac{z_h}{g}\ddot{x} \tag{2.1}$$

$$p_y = y - \frac{z_h}{g}\ddot{y} \tag{2.2}$$

On the other hand it is clear that the ZMP has to stay inside the support polygon and it is also predictable where the robot will set its feet. Thus it is possible to define the trajectory of the ZMP in the near future. The necessity of this will be discussed later. A known approach to make use of it is to build a controller which transforms this reference ZMP to a trajectory of the center of mass of the robot [11]. Figure 2.1.1 shows the pipeline to perform the transformation. The input of the pipeline is the desired translational and rotational speed of the robot which might change over time. This speed vector is the desired speed of the robot, which does not translate to its CoM speed directly for obvious stability reasons, but merely to its desired average. The first station in the pipeline is the Pattern Generator which transforms the speed into desired foot positions $\mathbf{P}_{global}$ on the floor in a global coordinate system used by the walking engine only. Initially this coordinate system is the robot coordinate system projected on the floor and reset by the Pattern Generator each time the robot starts walking. The resulting reference ZMP trajectory p^{ref} calculated by "ZMP Generation" (see section 2.1.3 for details) is also defined in this global coordinate system.

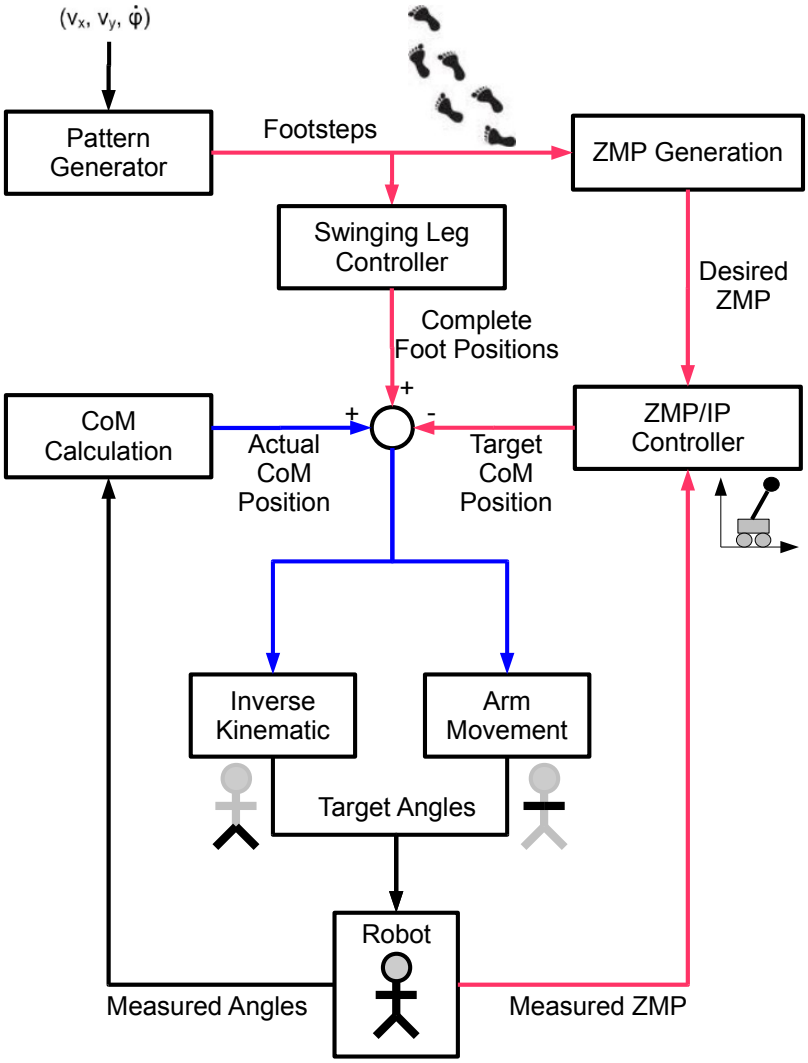


Figure 2.1: Control structure visualization of the walking pattern generation process. Data expressed in the robot coordinate system are represented by a blue line and data expressed in the global coordinate system is represented by a red line.

The core of the system is the ZMP/IP-Controller, which transforms the reference ZMP to a corresponding CoM trajectory ($\mathbf{R}_{ref}$) in the global coordinate system as mentioned above. The robot's CoM relative to its coordinate frame ($\mathbf{R}_{local}$) is given by the framework based on measured angles. Equation 2.3 provides the foot positions in a robot centered coordinate frame.

$$\mathbf{P}_{robot}(t) = \mathbf{P}_{global}(t) - \mathbf{R}_{ref}(t) + \mathbf{R}_{local}(t) \tag{2.3}$$

Those can subsequently be transformed into leg joint angles using inverse kinematics. Finally the leg angles are complemented with arm angles which are calculated using the x coordinates of the feet.

2.1.2 The ZMP/IP-Controller

The main problem in the process described in the previous section is computing the movement of the robot's body to achieve a given ZMP trajectory. To calculate this, a simplified model of the robot's dynamics is used, representing the body by its center of mass only. The ZMP/IP-Controller uses the state vector $\mathbf{x}=(x, \dot{x}, p)$ to represent the robot where x is the position of the CoM, $\dot{x}$ the speed of the CoM and p the resulting ZMP [10].

The system's continuous time dynamics can be represented by

$$\frac{d}{dt}\begin{bmatrix} x \\ \dot{x} \\ p \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 \\ \frac{g}{z_h} & 0 & -\frac{g}{z_h} \\ 0 & 0 & 0 \end{bmatrix}\begin{bmatrix} x \\ \dot{x} \\ p \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} v \tag{2.4}$$

where $v = \dot{p}$ is the system input to change the ZMP p according to the planned target ZMP trajectory p^{ref}. As can be seen in figure 2.2, it is not sufficient to start shifting the CoM simultaneously with the ZMP. Instead the CoM has to start moving before the ZMP does. Therefore a preview of p^{ref} is needed to be able to calculate a CoM movement leading to a stable posture.

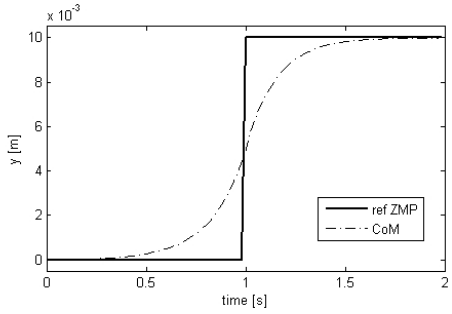


Figure 2.2: CoM motion required to achieve a given ZMP trajectory.

In the case of this biped walking problem the measurable part of the state vector is the ZMP position $p^{sensor}(k)$. Therefore it is necessary to integrate an observer in the system to estimate x and $\dot{x}$ [12]. Figure 2.3 shows the overall system configuration. The observer is put in parallel to the real robot and receives the same output of the controller

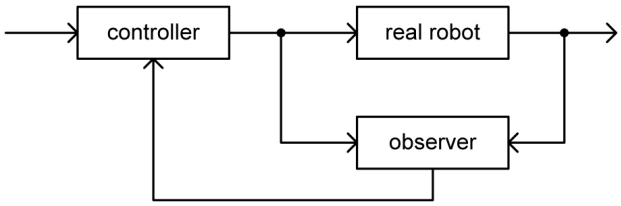


Figure 2.3: Control system with sensor feedback using an observer.

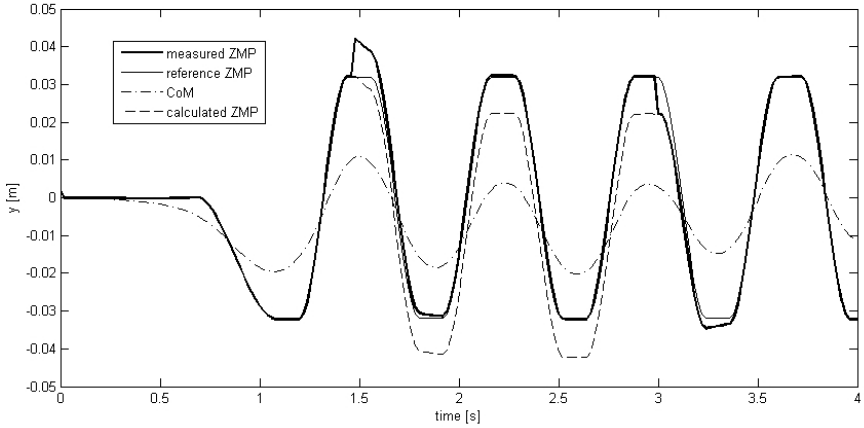


Figure 2.4: Performance of the controller under the influence of a constant external disturbance resulting in an error in the measured ZMP.

to estimate the behavior of the real system and is supported by the measurements of the ZMP. The output of the observer is the estimated state vector $\widehat{\mathbf{x}}$ needed by the controller to calculate the next desired CoM position.

An intuitive illustration of this observer-based controller's performance is given in figure 2.4, where a constant error is added to the ZMP measurement for a period of 1.5 s. This error could be interpreted as an unexpected inclination of the ground or a constant force pushing the robot to one side. The control system incorporates this difference and compensates by smoothly adjusting the CoM trajectory, which consequently swings more to the opposite direction.

A more detailed presentation of our approach and algorithms is presented in [13] and their application to the Nao in [14].

2.1.3 ZMP Generation

The ZMP Generation calculates a reference ZMP using the given foot steps. Within the support polygon the position can be freely chosen since every position results in a stable walk. On the x axis the ZMP proceeds with the desired speed. This results in a movement of the center of mass with constant velocity. On the y axis the ZMP is a Bézier curve with a control polygon of 4 points and dimension 1. The coordinate of each

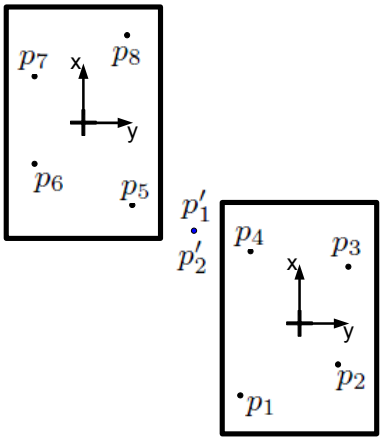


Figure 2.5: The control polygon consists of 4 points per foot with constants coordinates within the respective coordinate frame. In this example a positive x speed is assumed for a better visualization.

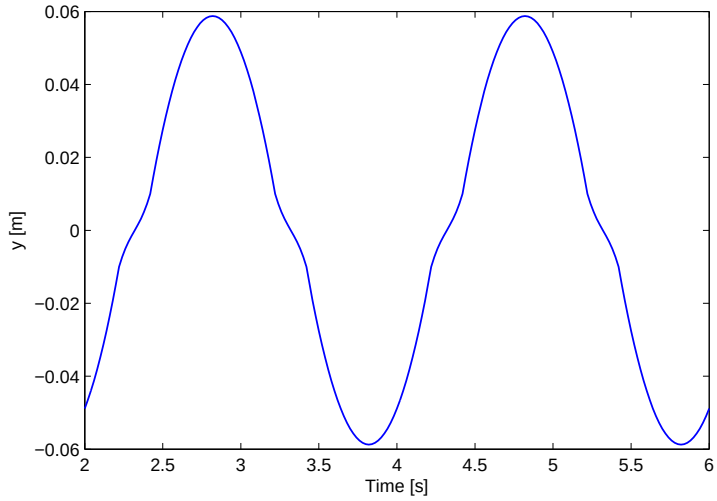
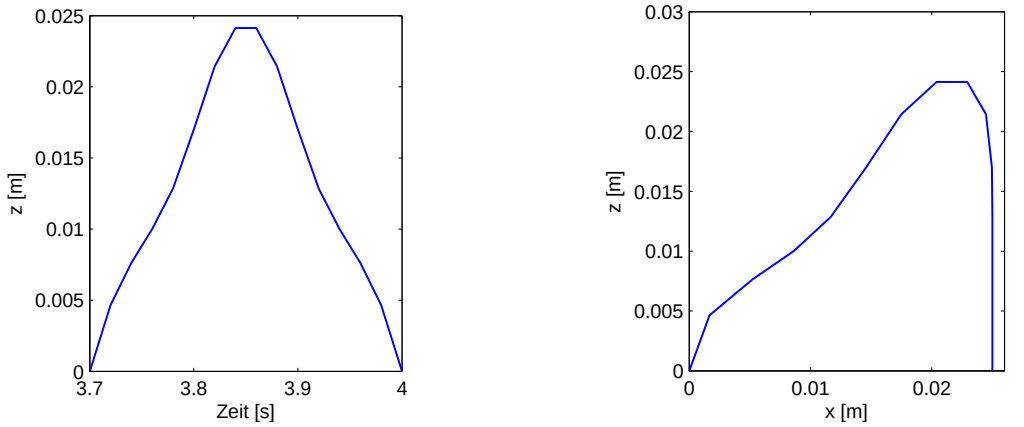


Figure 2.6: Resulting reference ZMP along the y axis.

point is constant within the coordinate system of the respective foot. Figure 2.5 gives an example. In the right single support phase the control polygon is $P_r = \{p_1, p_2, p_3, p_4\}$. The same applies to the other single support phase. In the double support phase the control polygon consists of the points p_4, p'_1, p'_2, p_5, where p'_1 and p'_2 are the same point in the middle of p_4 and p_5. This leads to a smooth transition between the single and double support phases. Figure 2.6 shows the resulting reference ZMP along the y axis.

2.1.4 Swinging Leg Controller



(a) Step height over time.

(b) Stepheight over x for a walk along the x axis.

Figure 2.7: Movement of the swinging leg in the global coordinate system.

The PatternGenerator sets the foot positions on the floor. They could be imagined like footsteps in the snow. Therefore the footpositions of the swinging leg during a single support phase are missing. They are added by the Swinging Leg Controller which calculates a trajectory from the last point of contact to the next utilizing a B-spline. The control polygon consists of 9 points with 3 dimensions each. The x and y coordinates are set along the line segment between the start and end point. The z coordinates are increased and decreased respectively by $\frac{1}{6}$ of the maximal step height. The z coordinate over time can be seen in figure 2.7(a). To reduce the influence of the leg inertia the feet are lifted and lowered at the same speed. Figure 2.7(b) shows the z coordinate over x. The foot reaches its end position along the x axis some time before the single support phase ends. This reduces the error if the foot hits the ground too early.

2.2 Special Actions

All none-walking movements are executed by playback of predefined motions called *Special Actions*. These movements consist of certain robot postures called key frames and transition times between these. Using these transition times the movement between the

key frames is executed as a synchronous point-to-point movement in the joint space. Such movements can be designed easily by concatenating recorded key frames.

2.3 Current Research

Solving tasks given by playing robotic soccer games requires precision in motion execution. Thinking about tasks such as kicking a ball this is rather obvious since the precision of the kick highly influences the outcome of the motion. When thinking about reaching a good shooting position first of all a precise localization is needed. One essential part of this is updating the localization incrementally with the help of motion updates. Thus having precise knowledge about the executed motion also greatly helps the quality of localization. The concept of measuring such motion updates with the help of sensors is called odometry. In difference to rolling robots, humanoid robots cannot measure their motion directly with the help of sensors, but can calculate those by tracking feet positions with the help of actuator positions and kinematics.

While this is theoretically trivial experiments have shown that an application to a real robot, such as the Nao, leads to a rather high difference between calculated and executed motion. Observation of the walk reveals that the main reason for this effect is sliding and slipping of the robot. Given the nature of a dynamically stable walk it becomes obvious, that keeping the ZMP inside the support polygon requires high accelerations, especially of the swinging foot. Whenever the resulting forces cannot be maintained be the friction of the robot foot and the ground the sliding constraint is violated resulting in a sliding of the support foot relative to the ground. This motion cannot be measured by the actuators and leads to the observed odometry error.

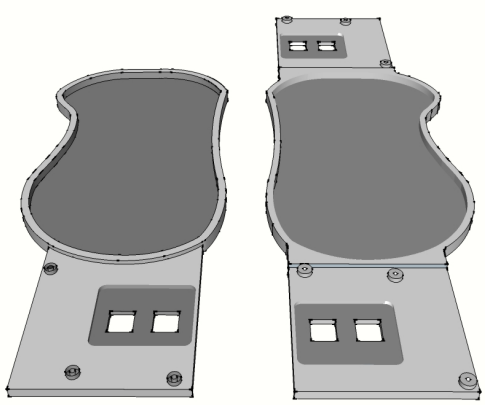
To improve the odometry information for humanoid robots team Nao Devils experiments with sensor capable to measure those error. While it's forbidden to add additional sensor during RoboCup games, using the average measured odometry error as a corrective factor would still be helpful to calibrate the robots motion during tournaments. Such sensors could even be integrated in future robotic platforms.

Measuring sliding motions relative to the ground is the task of computer mice. Modern optical mouse sensors are very capable, small in design and have low energy consumption, making them ideal for such a task. In addition they measure the traveled distance without having direct ground contact thus don't interfere with the measurement. Lifting them off the ground also ensures the measurement to stop. The motion of the swinging foot will thus be ignored. Measuring rotational errors is highly important since those error propagate to larger future errors over time. Unfortunately mouse sensors are only designed to measure motions in x- y-direction and cannot measure rotations. But combining two sensors, as can be seen in figure 2.8(b), overcomes this flaw (discussion can be found in [15]).

Since hardware modification is forbidden in the Standard Platform league team Nao Devils developed special shoes for the Nao, see figure 2.8(a), equipped with optical sensors. The gathered information is transmitted via ZigBee wireless transfer to an USB receiver inside the head. This allows for a temporary connection of up to four sensors.

First experiments have proven that the sliding error can be measured with these sensors, since both the translation and rotation can be tracked with efficiency. Figure 2.9 demonstrates a measurement from one walking experiment, walking in straight forward

(a) Sensor board



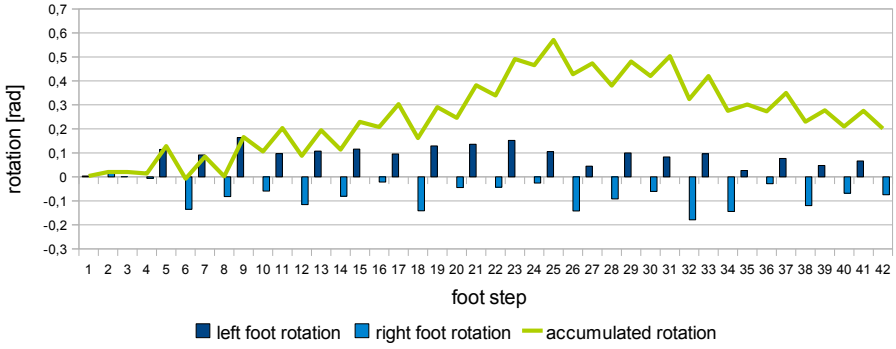
(b) Schematic view. Configuration 1 on the left, Configuration 2 on the right.

Figure 2.8: Sensor placement.

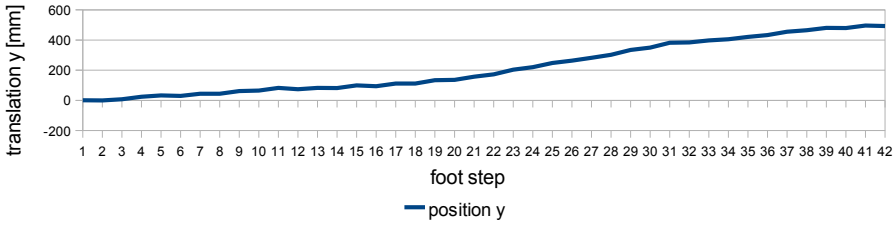
direction. The robot is turning slightly on its support foot during each single support phase. In general this is balanced by turning in the other direction during the next step. But the measurement reveals that the rotation is not compensated in every step, resulting in an deviation in y-direction over time.

While proving the concept the evaluation revealed that in praxis the measurement is not reliably enough. The reason is the used hardware, which is developed to be used in mobile mouse devices and is not powerful enough to measure the motion correctly under non-ideal circumstances. Observation of the walking experiment reveals that the rocking walking motion of the Nao sometimes violated the assumption that both sensors of one foot have ground contact, falsifying the measurement. Thus in the future further experiments will be conducted utilizing more advanced sensor hardware, such as Microsoft's *BlueTrack*[1] and Logitech's *DarkField Laser Tracking*[2]. A detailed overview on concept, implementation and evaluation can be found in [15].

[1] `http://www.microsoft.com/hardware/mouseandkeyboard/tracklanding.mspx`
[2] `http://www.logitech.com/images/pdf/briefs/Logitech_Darkfield_Innovation_Brief_2009.pdf`



(a) Step rotation



(b) Accumulated y-translation error

Figure 2.9: Example walk in x direction

Chapter 3

Cognition

From the variety of Nao's sensors only microphones, the camera and sonar sensors can potentially be used to gain knowledge about the robot's surroundings. The microphones haven't been used so far and are not expected to give any advantage. The sonar sensors provide distance information of the free space in front of the robot and can be used for obstacle detection. However, the usage of these sensors depends on maintaining a strictly vertical torso or at least tracking its tilt precisely since otherwise the ground might give false positives due to the wide conic spreading of the sound waves. Additionally, for certain fast walk types the swinging arms were observed to generate such false positives, too, and the sonar sensor hardware in general is currently unreliable and often does not recover from failure once the robot has fallen down. Thus for exteroceptive perception the robot greatly relies on its camera mounted in the head.

The following chapter describes the information flow in the cognition process starting from image processing and its sub-tasks in section 3.1. Special focus is given to new developments since 2009, meaning an improved line and center circle detection (see section 3.2) and a new localization module based on a multiple-hypotheses unscented Kalman filter (see section 3.3). Finally section 3.4 gives an outlook about current research that did not find its way into the code for RoboCup 2010, yet, but is expected to be applied in 2011.

3.1 Image Processing

The vision system used for the Nao in RoboCup 2008, 2009 and 2010 is based on the Microsoft Hellhounds' development of 2007 [16]. The key idea is to use as much a-priori information as possible to reduce both the scanning effort and the subsequent calculations needed. To process only a small fraction of all pixels the image is scanned along scan lines of varying length depending on the horizon's projection in the image (see fig. 3.1). These scan lines are projections of radial lines originating from the point on the field below the camera center. Keeping the angular difference constant allows the implicit usage of this information during the scanning process, optimizes transformations between image and field coordinate systems and significantly simplifies the inter-scan-line clustering and reasoning about detected objects. In case of ambiguity additional verification scans are carried out. A sparse second set of horizontal scan lines is introduced to detect far goals.

All relevant objects and features are extracted with high accuracy and detection rates.

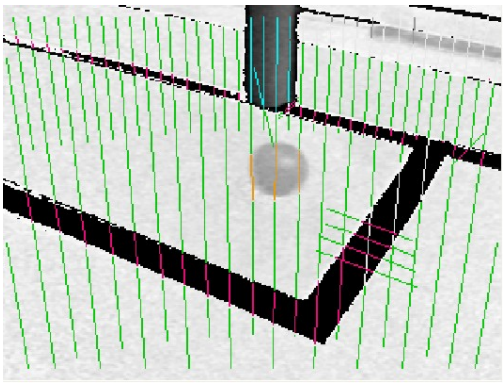
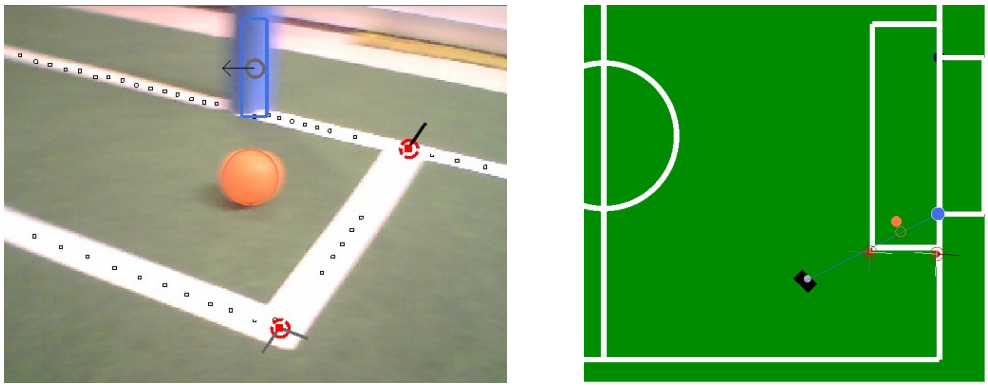


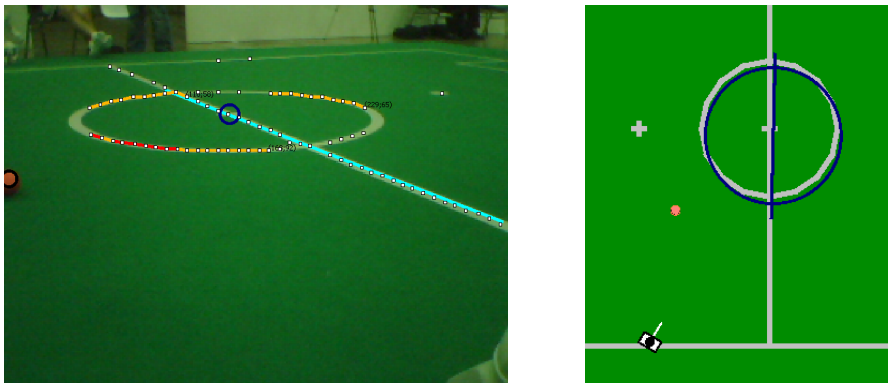
Figure 3.1: Image scanned along projections of radial lines on the field.



(a) Camera Image.

(b) Field view.

Figure 3.2: Objects and features recognized in the camera image.



(a) Camera Image. (b) Field view.

Figure 3.3: Recognition of the center circle.

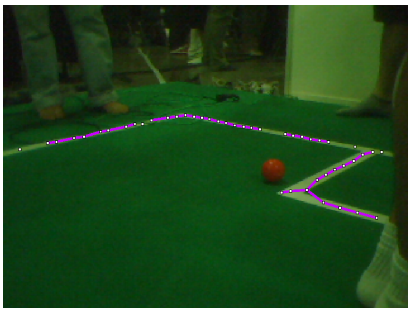
Since the Nao SPL is the first RoboCup league (neglecting the simulation leagues and the Small Size League with global vision) without extra landmarks beside the goals, detecting features on the field itself becomes more important. The detection of those is described in more detail in the following section.

3.2 Line and Circle Detection

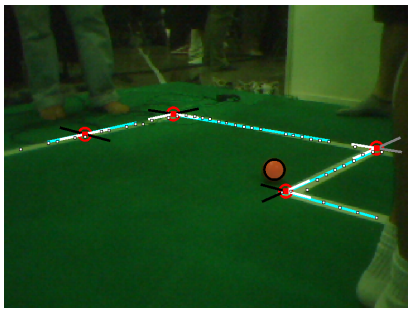
Besides lines also their crossings (fig. 3.2(a)) and the center circle (fig. 3.3(a)) are determined. The latter in combination with the middle line allows for a very accurate pose estimation in the center area of the field (see fig. 3.3(b)), leaving only two possible symmetrical positions which can be easily resolved with any goal observation. In certain situations line crossings from the penalty area can even be used to disambiguate a left or right goal post (see fig. 3.2). Thus instead of having to look up for the crossbar or around for the second post the robot can focus on tracking the ball.

The input for the line and circle detection algorithm is a set of line points generated in the main scanning routines described above. Each point annotates a position where a scan line intersected a field line. Besides the positions both in the image and on the field additional information are given about the gradient and the index of the scan line which generated this line point.

The previous approach used till 2009 consisted of a clustering heuristic based on positions and gradients to generate line fragments. This clustering has squared complexity in the number of line points but can be computed efficiently on the Aibo/Nao in less than 1 ms. Those line fragments are fused into lines to calculate line crossings on the field. In an intermediate step the line fragments are tested for circle tangents. The intersection of the perpendicular bisector of each pair of fragments in field coordinates can be considered as a center of the expected center circle. A visible center circle in the image tends to generate several of those intersections close to each other with approximately the right radius. The resulting center point's precision can be improved further by projecting it to the most likely middle line hypothesis. The existence of a suitable middle line hypothesis



(a) Detected line points linked together. Chains might be broken due to missing line detection in-between fragments.



(b) Field lines and resulting crossings, including a virtual crossing (left) and a partly unclassified one (upper right).

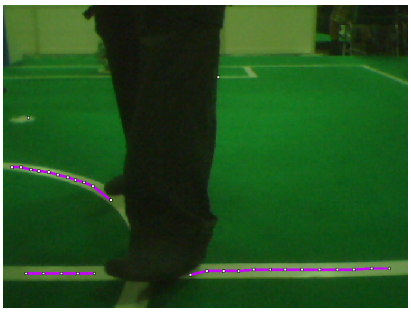
Figure 3.4: Forming chains out of detected line points to identify field lines.

is also useful to reject false positives generated from other green-white structures around the field. This is a fast method for line and circle detection that worked well on the Aibo since 2005 (as part of the code which won several Opens with the Microsoft Hellhounds and the RoboCup with the German Team) and also on the Nao (with minor adaptations in parametrization). A drawback on the current SPL field however is that the white goal net tends to generate a number of line fragment false positives. Additionally, a significant part of the center circle needs to be visible for reliable detection.

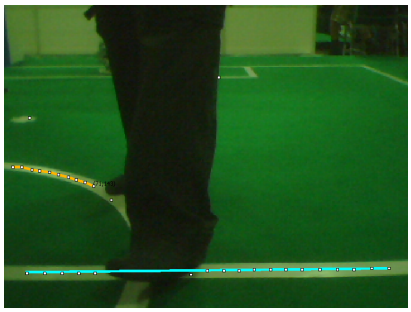
For 2010 the line and circle recognition has been rewritten with the goal to overcome those problems. The implementation of this new method can be found in the module *NDLineFinder2*. While the color information and the general quality of the Aibo's camera was inferior to the Nao's, the latter one's in-built edge enhancement often generates artifacts around the field line edges which in turn decrease the gradient quality when applying for example a simple Sobel operator as done here. This is why the new approach neglects gradient information. The input line points are linked into chains based on their originating scan lines, their relative distances, and white color classification on at least one of the pixels between them perpendicular to their connection. This concatenation is of linear complexity in the number of line points in case those line points are sorted by their originating scan lines which is given due to the scanning process itself.

Those resulting chains (as illustrated in figure 3.4) offer easy, reliable and efficient ways to extract line fragments or circle fragments. Line fragments can be found by identifying sub-chains exceeding a specified minimal length which comply to a given line heuristic. In this case the average and the biggest deviation from a linear regression line fit is chosen. In order not to evaluate new linear regressions for the addition of every new point those sub-chains are extended as long as additional points do not violate the thresholds for the initial line fit. In case the average or biggest error exceeds the initial line fit, a new fit is calculated and examined, and eventually extended further.

The extraction of circle parts out of the given line point chains is done in a similar fashion. One possibility to do this would be to try and find a series of points with similar non-zero curvature. This however is very prone to noise since the discretization introduced by the pixels is of the same magnitude as the curvature itself as can be seen in figure 3.3(a). The alternative approach taken here is the identification of trends in



(a) Detected line points linked together.



(b) Classification of field line and circle fragment.

Figure 3.5: Reliable center circle detection even when only small fragments are visible.

the direction change of line point connections. As a first step the local tangential angle at the line point is calculated as a smoothed value from the point and its predecessors and successors (if available). Similar to the linear regression for line fragments sub-chains are identified which angular changes show a trend significantly different from zero (since zero corresponds to straight lines) with average and maximum errors below a certain threshold. As for the line detection, recomputing the trend is only necessary for a small subset of points belonging to a single circle fragments (indicated with red connections in figure 3.3(a)). The points of all identified sub-chains are then used to estimate the best-fit circle in field coordinates using the Levenberg Marquardt method. This circle percept can be used as is or augmented with the line information to provide an orientation for the center circle in case a suitable middle line can be identified. The results of the described line and circle detection algorithms are shown in figure 3.3 and 3.5.

3.3 Localization

One main focus of research is on Bayesian filters, where several enhancements for real time vision-based Monte Carlo localization systems [17] have been presented, and the approach based on the detection of field features without using artificial landmarks has won the "almostSLAM" Technical Challenge at RoboCup 2005 [6]. The methods used up until RoboCup 2009 have all been based on those particle filters developed between 2005 and 2007. For RoboCup 2010 however a different approach has been developed and used in the competitions.

Inspired by [18] the basic idea was to combine the smoothness and performance of Kalman filtering with a multi-hypothesis system. The latter is necessary to allow recovery from huge errors due to extended periods of integrated odometry errors without correcting through observations, or rapid unexpected position changes due to contact with other robots or "teleportation" by human intervention. All of those issues occur in robot soccer games and are amplified by the huge odometry errors inherent in fast biped walking.

A classic Kalman filter applied to localization in a SPL scenario can only be used for position tracking, and only as long as the estimate does not deviate too much from the true positions. Most perceptions on a SPL field are ambiguous, so the sensor update will

only be done with the most likely data association. At the same time, the perception of field features like field line crossings is often uncertain so that L- and T-crossings can not be distinguished (e.g. as in figure 3.4(b)). In such cases wrong associations tend to drive the estimation further away from the true position. Recovery from such situations is only possible when assigning huge weights (e.g. small observation covariances) to unique perceptions like observing both goal posts in the same frame which then decreases the robustness against false perceptions originating from the audience around the field.

This problem is addressed in [18] with a sum-of-Gaussians Kalman filter, where each Gaussian is split into several new ones representing the results of different association choices. Applying all possible data associations to every hypothesis generates exponential growth which needs cutting back shortly after by applying pruning heuristics and fusing similar Gaussians to prevent an explosion of computational complexity. Thus lots of processing time is wasted on creating and destroying new hypothesis which are either unlikely or very similar to the ones that already exist.

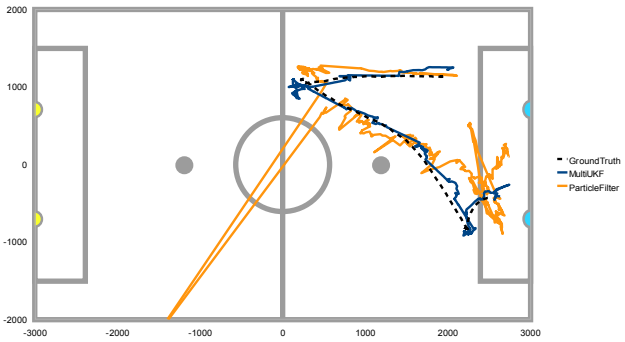
A different approach is implemented in the module *MultiUKFSelfLocator*. Only few new hypotheses are generated periodically at positions with high probability based only on recent sensor information. Those hypotheses are only updated using data that lies inside a certain expectation threshold. Non-linearity in the sensor model is addressed using the unscented transformation technique. Several other approximations and simplifications result in a localization method that is an order of magnitude faster than the previously used particle filter while providing superior localization quality and increased robustness to false positive perceptions (see figure 3.6). A detailed presentation of the approach and its stochastic soundness will be given elsewhere.

3.4 Current Research

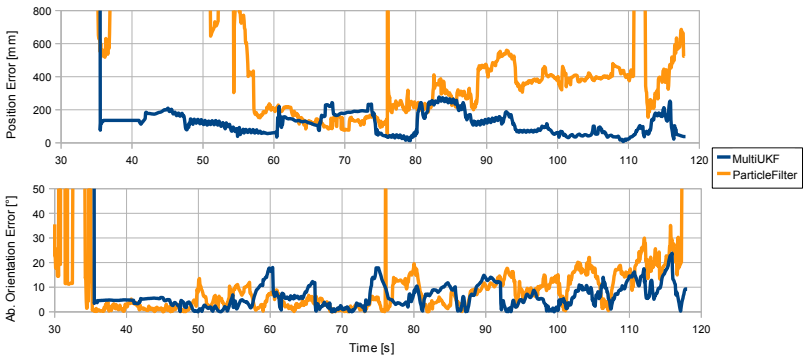
The modules and algorithms described in the previous sections 3.1 to 3.3 can all be found in the code released together with this document. This section summarizes both current and previous research and experiments for which the transition into the actual soccer code has not been done yet but is expected to influence the code for RoboCup 2011.

3.4.1 Active Vision

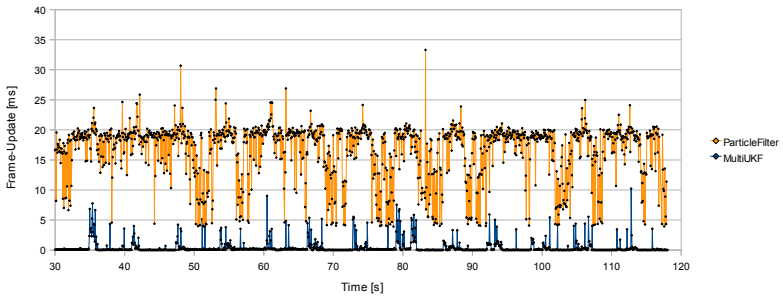
A recent development comes in the form of an active vision module replacing the common strategy to simply move the robots head continuously to cover as much as possible of the robot's environment. Details of the approach have been presented at the RoboCup symposium 2010 [19]. The basic idea of this new active vision module is to move the camera in a way that is optimal for localization instead of just scanning the environment using predefined trajectories (see figure 3.7). The basis for this is a particle filter localization providing the current belief state as a particle distribution. A commonly used quality criterion for such a belief state X is the entropy $H(X)$. Since the true localization of the robot is unknown, the evaluation of expected consequences of active vision decisions needs to be based on the uncertain belief state. The sensor modeling for the (mostly ambiguous) landmarks (visualized in figure 3.8 for two fixed positions and observations) provides an estimation of the probability distribution $P(y|u, X_t)$ for making observations y after executing a motion u based on the current belief state X_t. Thus a simulation of



(a) Estimated positions and ground truth on the field.

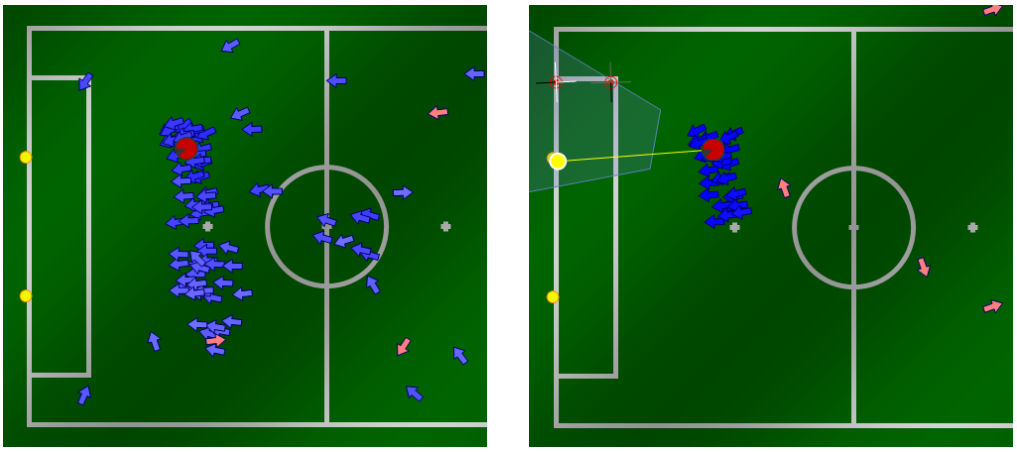


(b) Errors in pose estimation.



(c) Localization runtime.

Figure 3.6: Localization of Multiple-Hypotheses UKF compared to previous particle filter solution (which was used in RoboCup 2009). Both are running in parallel on the Nao using the same perception as input. Ground truth is provided by a camera mounted above the field.



(a) Ambiguity close to the opponent goal which can not be resolved by observing a single goal post alone.

(b) The optimal viewing direction estimated from the previous particle distribution.

Figure 3.7: Basic idea of active vision: Choosing the viewing direction which improves the current localization the most.

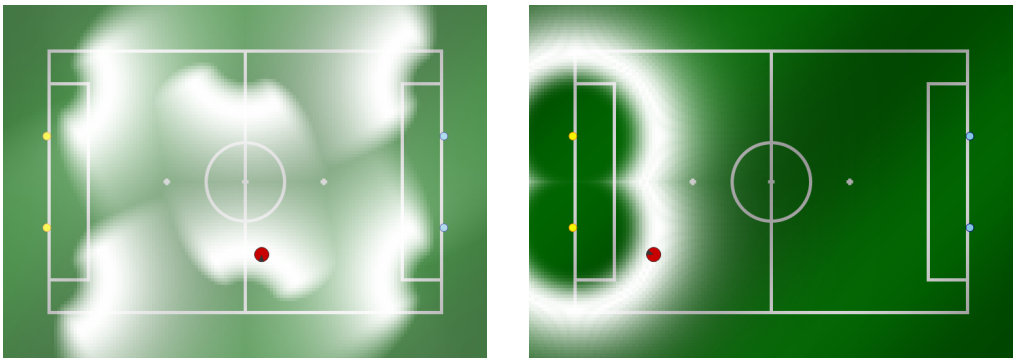
the localization updates for each possible action can be performed and the one expected to minimize the entropy can be chosen to be executed.

Since such simulations include considering all possible observations for all actions for a current belief state which is expressed with a number of particles, this calculation involves a significant computational complexity. Several approximations were considered, implemented (if practical), and evaluated for approximation quality and speed-up:

- Using clusters instead of separate particles for entropy calculation; neglects the particle variations inside the cluster.
- Precomputing observations in look-up table; introduces discretization error.
- Precomputing the measurement probability for each possible pose and action decision; not practical since the table size exceeds 100MB for sensible resolutions.
- Neglecting areas with low particle density.

The effects on estimation quality are visualized in figures 3.9 and 3.10. The resulting system has a average runtime of 4.2 ms on the Nao. Comparative results of the presented approach against the previously used passive scanning are visualized in figures 3.11 and 3.12 where the robot walked to a series of points.

This module has not been used in RoboCup 2010. The main reason has been the change of the localization strategy described in section 3.3. There are also some additional issues that need to be addressed. Since the underlying particle localization does not take into account negative information the active vision approach tends to fail in kidnapped robot scenarios where the belief state is absolutely wrong. In this case the active vision decision is based on incorrect data and the supposedly optimal decision may look at areas without useful features. Since "not seeing anything" does not change the particle



(a) Line crossing observations result in highly multi-modal but focused distributions.

(b) Observing a goal post indicates the field half, but distinguishing between left and right post is not always possible.

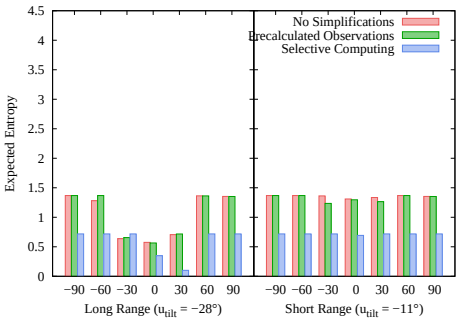
Figure 3.8: Measurement probability distribution $p(y_t|x_t)$ for different observations. Brighter areas correspond to higher probabilities. Illustration is related to x und y with probability integrated over φ.

distribution the belief state stays the same and so does the active vision decision. For this reason the described system needs a monitoring concept switching back to predefined scanning in such failures. Another point where different handling becomes necessary is ball tracking, especially when done conceptually different from the localization e.g. with a Kalman Filter, in which case the active vision decision could not be calculated in a single uniform model.

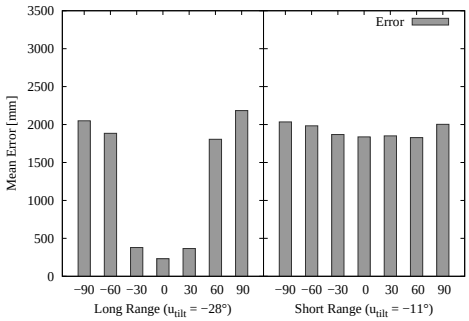
3.4.2 Distributed Simultaneous Localization and Robot Tracking

Current work includes robust cooperative world modeling and localization using concepts based on multi robot SLAM. Keeping track of the robot's dynamic environment (see figure 3.13) opens the possibility for tactical behavior decisions beyond simple reactive behaviors which are currently in use. In this joint modeling of the robot's state a particle filter estimates the robot's pose. Clusters of particles are combined into super-particles which map the dynamic environment using a number of Kalman filters. This represents an approximation of FastSLAM and both decreases the integration of odometry error compared to robot-centric local modeling (see figure 3.14) and allows resolving multi-modal localization belief states using shared information. The approach even preserves its SLAM functionality and is able to maintain a robot's localization based on mapped dynamic obstacles only. A detailed presentation of the approach is presented in [20].

By specifically addressing the heterogeneity of the perceived information and the need to synchronize the estimation between the team of robots the task's complexity can be reduced to be in the range of applicability on limited embedded platforms. This module's average runtime on the Nao in the configuration used on the RoboCup 2010 would have been slightly above 20 ms which would not have allowed to keep a frame rate of 15 Hz or even 30 Hz. Especially the switch to a motion frame rate of 100 Hz made its application impossible. Additionally the system is based on the outdated particle filter localization

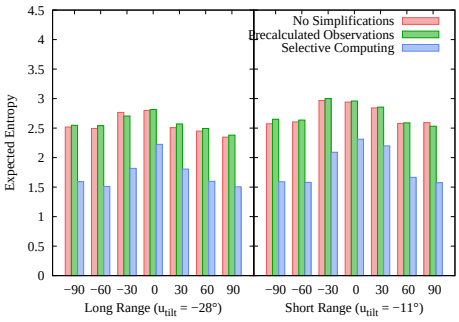


(a) Expected entropy.

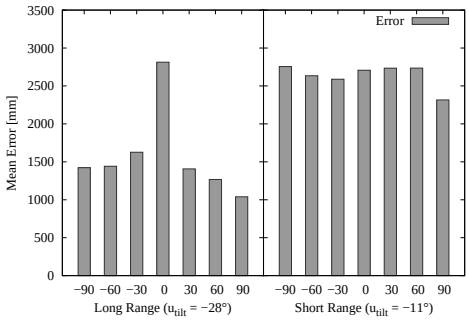


(b) Average error of particles.

Figure 3.9: The expected entropy and the real error of particles after executing an action on a penalty kick position facing the center circle.



(a) Expected entropy.

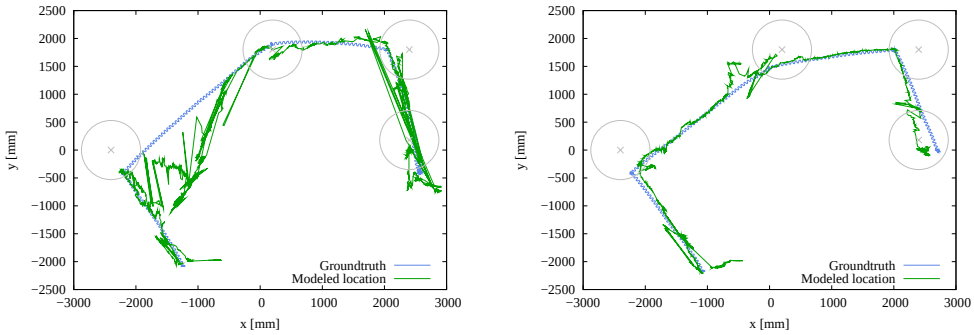


(b) Average error of particles.

Figure 3.10: The expected entropy and the real error of particles after executing an action on the removal-penalty's return position, i.e. on the side line facing the center circle.

of previous years.

For 2011 it is planned to integrate parts of this distributed world modeling approach with the new localization system described in section 3.3. Since the UKF localization concept does not allow the transformation of a combined estimator according to the Rao-Blackwell theorem the SLAM aspect can not be transferred in this case. It is still possible however to perform a simultaneous distributed estimation of several world map hypotheses using the concepts described in [20]. The advantages of coping with huge odometry errors can thus be transferred while still providing a suitable alternative world map when relocalizing after getting lost or being moved by external influence as happens frequently in RoboCup SPL games.



(a) Passive localization. (b) Active localization.

Figure 3.11: Localization (green) versus true position (blue) of the robot while walking to a series of target positions.

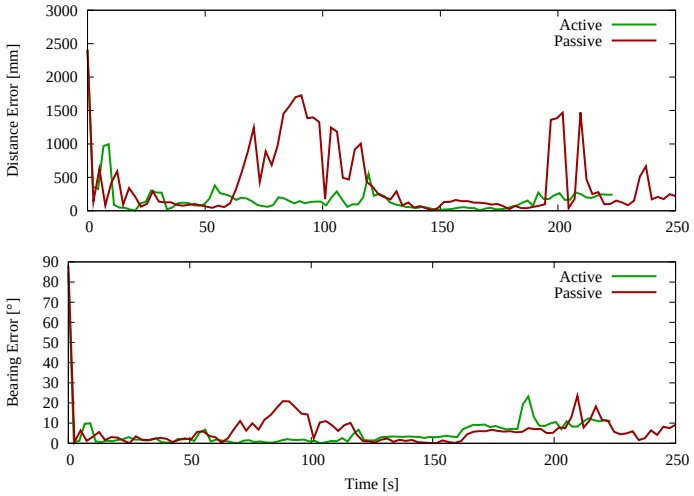


Figure 3.12: Position errors (in average 172 mm and 418 mm) and orientation errors (in average 4.97° and 6.14°) of localization while walking to target positions on the field for the active and passive approach, respectively.

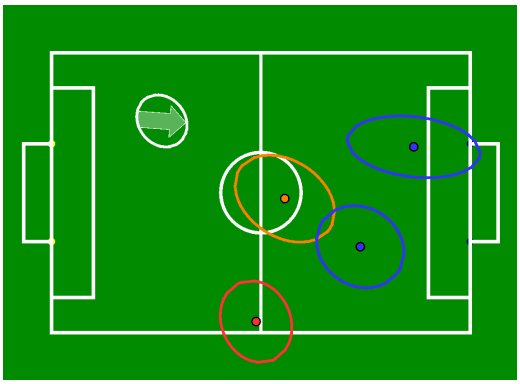
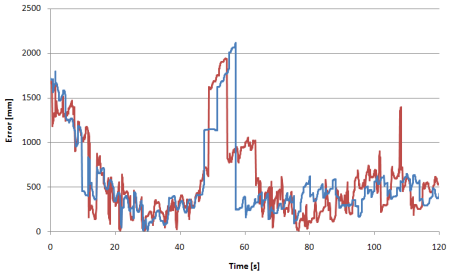
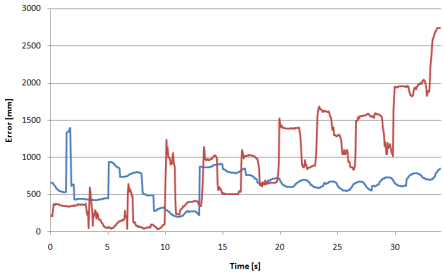


Figure 3.13: Modeling of the robot's dynamic environment.



(a) No gain for frequently observed robots.



(b) Tracking of infrequently observed robots significantly improved.

Figure 3.14: Advantage by unified (blue) compared to separate robot-centric modeling (red).

Chapter 4

Behavior

For behavior design the *Nao Devils* and previous Dortmund teams relied mostly on XABSL (*Extensible Agent Behavior Specification Language*). XABSL was developed in its original form in 2004 using XML syntax [21] in Darmstadt and Berlin and adapted in 2005 to its current C-like syntax and a new ruby-based compiler by the *Microsoft Hellhounds*.

Behavior is specified by option graphs. Beginning from the root option, subsequent options are activated similar to a decision tree until reaching a leaf, i.e. an option representing a basic skill like "walk" or "execute_special_action" which are parameterized by the calling option. Each option contains a state machine to compute the activation decision based on a number of input symbols provided by other modules (see section 3.1). Part of the soccer playing option graph is shown in figure 4.1 as a demonstration example.

The behavior is divided in the three basic playing roles, *striker*, *supporter* and *goalie*. While the latter, due to the rules, has a fixed role and behavior, the striker and supporter switch roles dynamically, based on transmitted *soccer action* symbols and situational awareness. Details about the structure of the team communication can be found in [7] section 3.5.4.

While it is possible to design complex behavior using XABSL, several tasks may prove difficult or impossible to specify using XABSL alone, e.g. robust and efficient path planning including obstacle avoidance and also any strategic team behavior extending beyond simple role switching. Therefore several possibilities were investigated to augment XABSL.

4.1 Behavior Coordinate System

Past approaches to behavior planning of team *Nao Devils* were based on the robots current view and thereby on the *robot centric* coordinate system. As described in chapter 2.1 the walking engine applied by team *Nao Devils* is based on the concept of dynamic stability. Thus per definition it is impossible to stop the robot at any time during the execution of a walking motion. This problem is coped with by introducing a preview phase containing the next planned motion which has to be executed to ensure stability. Intuitively this corresponds to the inability to stop all motion while in the process of having one foot lifted to bring forward when waling fast. In every case one has at least to execute the

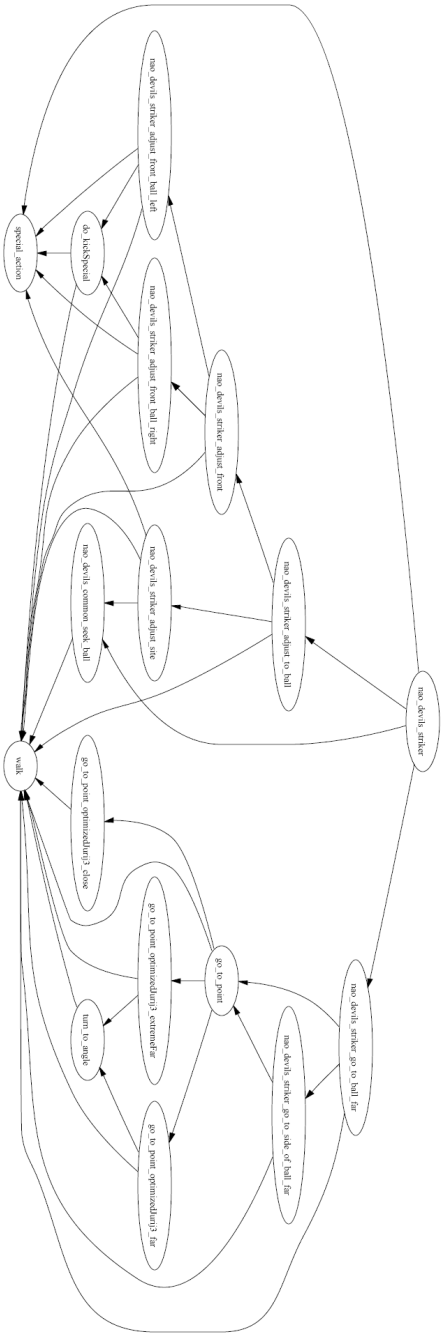


Figure 4.1: XABSL Option graph example.

current step to its end. But as a result a change of the walk request can only be executed within a given time, i.e. when the current swinging foot touches the ground and the next footstep is not yet planned into the current motion, thus resulting in a delay. This is especially true for a complete stop, which is applied to position the robot next to the ball. If a behavior is written to only set the target speed to zero the moment the target position is reached, the robot tends to overshoot and is likely to stumble against the ball.

In the past this problem was dealt with by stopping the robot some time before actually reaching the ball. But since the distance traveled after the stopping command is given differs depending on the speed and currently executed motion of the robot, finding the right distance was a matter of time consuming manual tuning of the behavior and still resulted in suboptimal results. Since XABSL itself does not address this problem, it is solved by introducing a new coordinate system, called *after preview*, centered on the expected future robot position. Actually this coordinate system is not related to a fixed time interval in the future, but rather to a dynamic offset depending on the executed motion or walking phase, which might also be zero in case of statically stable motions which can be interrupted at any time.

Consequently also the representations available to the XABSL behavior module are no longer the direct localization and tracking outputs, but those are transformed to the *after preview* coordinate system. This allows the behavior to make decisions based on the exact future state of the robot at which those behavior decisions actually have an impact. All intermediate actions till this point will be executed independent of the current decision, so now the decisions will be based on the correct environmental circumstances on which this decision will be acting upon, resulting in more precise and reactive behavior planning.

When visualizing the corresponding representations (marked by "*...AfterPreview*") in SimRobot, the origin for drawing first needs to be set using the following command:

```
vfd worldState origin:RobotPoseAfterPreview
```

4.2 GoTo Motion Command

The motion commands used in previous years till RoboCup 2009 have been based on desired speed vectors which were updated with every behavior execution. This mode of control dates back to the AIBOs which could be controlled like omnidirectional vehicles. Additionally for an AIBO it was sufficient to walk straight to the ball to grab and turn with it. For humanoid robots however the the omnidirectional characteristic of the walking generation is much less distinct, and at the same time a target position close to the ball has to be reached with a certain target orientation which increases the difficulty for trajectory control. While it is possible and also commonly done to generate omnidirectional walking patterns with the walking engine described in section 2.1, for a humanoid robot such as the Nao it is far more convenient to walk straight than it is to walk sideways. This is reflected in the possible walking speeds in each direction. Generating smooth path trajectories following the characteristics optimal for those described walking capabilities is obviously not possible in an intuitive way using a state-based behavior description language such as XABSL.

To overcome those limitations and ultimately to achieve more precision and speed in positioning close to the ball, a more advanced approach to path planning has been done for the *Nao Devils'* robots for RoboCup 2010. The utilized *Dortmund Walking*

Figure 4.2: SimRobot worldstate with path debug drawing.

Engine is based on foot step planning (see section 2.1). Previously the output of the XABSL behavior, i.e. the motion speed vector, has been used to generate those step patterns. Since XABSL has no knowledge about the planned footsteps much information about the motion is lost. Therefore a basic behavior has been introduced allowing the trajectory planning to be done by the walking engine which has much better control and feedback about the executed motion. The motion request has been adapted accordingly to accept a target position and orientation and different XABSL *go_to* commands have been implemented to cover common motion tasks.

In the very basic version a target position relative to the current behavior reference frame (compare section 4.1) can be specified. This generates a smooth trajectory guiding the robot to the desired position. In case the desired target orientation and position allows a direct approach where the robot walks mostly straight, this results in the robot approaching the desired position with maximum speed to the last step without any need to slow down significantly before reaching the position. Otherwise the last steps are executed using omnidirectional step patterns to avoid moving in too large loops and wasting time on the way. Even with the same walking parameters as in the previous year (and thus the same maximal speed) the average speed during game situations could raised significantly, and faster walking speeds resulting from recent changes in parametrization and ZMP trajectory generation could be exploited to a far bigger extend.

The paths resulting from those behavior calls can be visualized using SimRobot. After setting the origin for drawing to the behavior coordinate system as described in section 4.1, the following command will display the corresponding drawing as can be seen in figure 4.2:

```
vfd worldState module:RequestTranslator:Paths
```

4.3 Dynamic Positioning

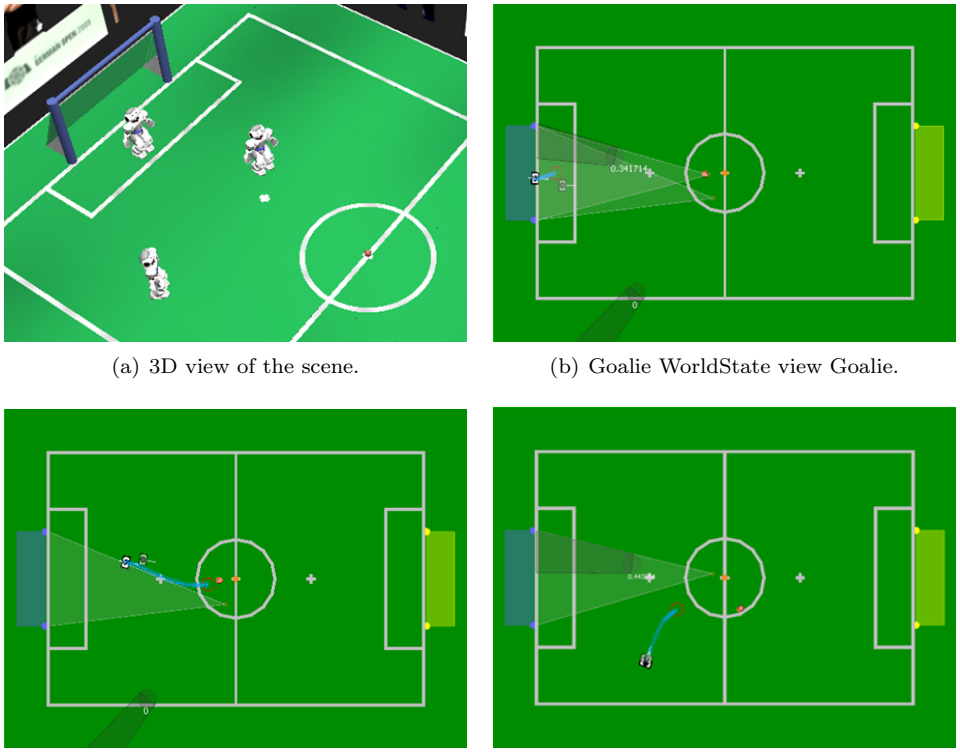
Two new key aspects are given by the improved localization described in section 3.3 and the *goto* command of the previous section. While the latter allows an easy and precise positioning in significantly less time, the localization quality allows far more than just to place the supporter between ball and goal. Instead it can be calculated how much of the goal is already blocked by other field players. For this the opening angle of the goal as perceived from the modeled ball position is calculated together with the ratio of this angle which is blocked by the corresponding field player. While the goal keeper will try to block its goal anyway, this information can be used for the striker to decide whether to approach the ball from the side or from the back, or for the supporter to dynamically position itself either defensively blocking the passage to the goal or in a more offensive position in case the striker already blocks enough of the opening angle to the goal just by positioning itself behind the ball for kicking.

This way it is relatively simple to design dynamic positioning which greatly helps team play and is useful for placing defending robots on ideal blocking positions between ball and own goal. Of course this would also be possible without the explicit *goto* command, but on the one hand the code needed even for such positioning is far less complex, and on the other hand the resulting motion execution allows repositioning to be done with the minimum possible number of steps necessary instead of constantly shifting the robot with small steps and low speeds. Thus the supporter itself can change its position to cover dynamically the part not yet covered by the striker. To demonstrates the benefit of this approach a situation with three robots is shown in figure 4.3. As an example figure 4.3(b) demonstrates the WorldState representation of the goalie, the grey shadows representing the communicated area which the other field players cover relative to their ball models. Of course this approach suffers from missing synchronization between the different models. This is expected to improve significantly with the distributed modeling approach described in section 3.4.2. However, this approach to dynamic positioning has already been applied successfully in RoboCup 2010 games.

4.4 Debug Tool

Debugging programmed code is essential to identify flaws and finding solutions to errors. This is true for the code of the robotic framework but also for the behavior code written to decide which action is most suitable regarding the current situation. As described in section 4, team Nao Devils utilizes the XABSL programming language to implement the soccer behavior. The written and compiled code of the behavior can be executed in a simulation environment or directly on the robotic hardware.

Different kinds of debugging tools are an essential part of every programming suite allowing to stop code during execution and get step by step information about the program state and variables. This concept is rather practical for debugging behavior using the simulator. Since the simulator runs both the robot code and the world simulation the whole simulation can be stopped to have a look at the current state and decisions. This allows to use the simulator to step frame by frame through the world state. To debug the behavior with the simulator SimRobot (figure 4.3(a)) a view the current XABSL tree containing XABSL symbols and variables (figure 4.4) can be displayed allowing a frame-by-frame forward simulation. Stepping back is not possible since the framework allows



(a) 3D view of the scene.

(b) Goalie WorldState view Goalie.

(c) Striker WorldState view.

(d) Supporter WorldState view.

Figure 4.3: Robot WorldState views demonstrating the communicated goal blocking

no rewind.

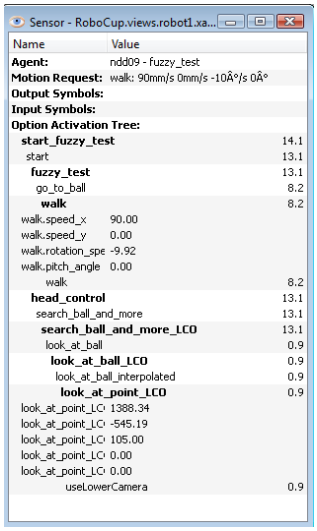


Figure 4.4: Example of a XABSL tree.

The used simulator is a physical simulation based on the Open Dynamics Engine (ODE). Therefore the simulation of even one robot is a rather time consuming process resulting in a simulation slower than real-time on most modern standard PC platforms. Simulating a complete match including six robots decreases the framerate to numbers that make a user observation rather difficult. Even if the simulation would run in real time the physical simulation is insufficient to model the real world exactly. Since small situational differences can result in big differences in behavior decision the simulation model is not satisfactory enough to test soccer behavior more complex than basic behavior. Thus the most important behavior debugging can not be done in the simulator but on the real robot.

Debugging code on the real robot is quite different. The use of a breakpoint concept to stop the program during execution would stop the motion of the robot resulting most likely in a fall of the robot. Therefore most debugging in autonomous robotics involves monitoring rather than breakpoint approaches. The program execution is supervised with the help of a remote connection that broadcasts information about the program state, sensor input and decisions. In contrast to the breakpoint approach this method allows for a debugging in real game situations involving other autonomous robots but is prohibited during tournament game play. Unfortunately the nature of the supervision process results in a delay of information. Therefore the supervision cannot be matched exactly with the current field situation. In addition the broadcast connection results in an disturbance of the the normal code execution. In general the disturbance is minor in scope but can result in timing problems. The result would be a stuttering of the robot which changes the state that should only be monitored. Especially for debugging of robot behavior supervision of the written code is a rather inadequate solution.

Since execution directly onto the robot is of special interest for behavior development team Nao Devils developed a tool allowing a different debugging concept combining the

convenience of simulator debugging with code testing on the real robot. The concept is based on logging each behavior decision during the execution. To allow a later analysis of the logged data the corresponding sensor information is also stored in addition to each decision. Since the complete sensor information, including the camera pictures, would result in an unmanageable amount of data, only the resulting information about the world model are saved.

With this stored data a playback of all behavior decisions and a debugging of decisions on given data can be done. But an analysis of the world model is not possible, since only the modeled gamestate is stored and cannot be compared with reality. But to allow even that comparison a video camera can be used to record the real gamestate in a movie file.

Based in these concepts team Nao Devils developed a tool with following features:

- The robot logs its internal state machine, the required symbols and hardware commands (motion, LED-Requests etc.) once per frame.
- A graphical user interface allows for an easy reading of the logfile and shows the input/output symbols and current XABSL state tree.
- A 2D field view shows the modeled robot position, direction and velocity, the ball position and the team mate positions.
- A video can be loaded and displayed. The time is synchronized manually by matching known state information to the video file.
- The the behavior log can be played back, stopped, stepped through frame by frame and rewound to interesting positions.

The tool is implemented using QT C++ as a platform independent standalone tool. The written log (.xlg) is loaded line by line and a parser extracts the information of the symbols and the state machine. The parser is specialized to interpret XABSL logs but due to clearly defined interfaces between the interpreter and the different widgets an adaptation to another behavior language would be possibly by changing the parser. Additional debug views specialized for other description languages can also be added easily.

The progress bar allows an intuitive navigation within the log file. The current frame information is shown using the following widgets (see figure 4.5):

- Input/output Symbols-Treeview Widget: Displays an overview with all logged symbols and their values represented in a tree. It's possible to select specific symbols to display more detailed information.
- Detailed Symbols Widget: All symbols selected in the Symbols-Tree-View are shown as a table here, so that the user has a quick overview about the needed symbols. The columns represent the symbol names, the rows represent different frames. By highlighting current frame the user can see the chronological history of the symbols. This is useful, if the user wants to judge the stability of a given symbol.
- Woldstate Widget: A 2D-Field is painted, displaying the current modeled robot position, direction, velocity and the latest ball position. This overview allows the user to judge the internal robot world model and compare it to the reality represented by the video file.

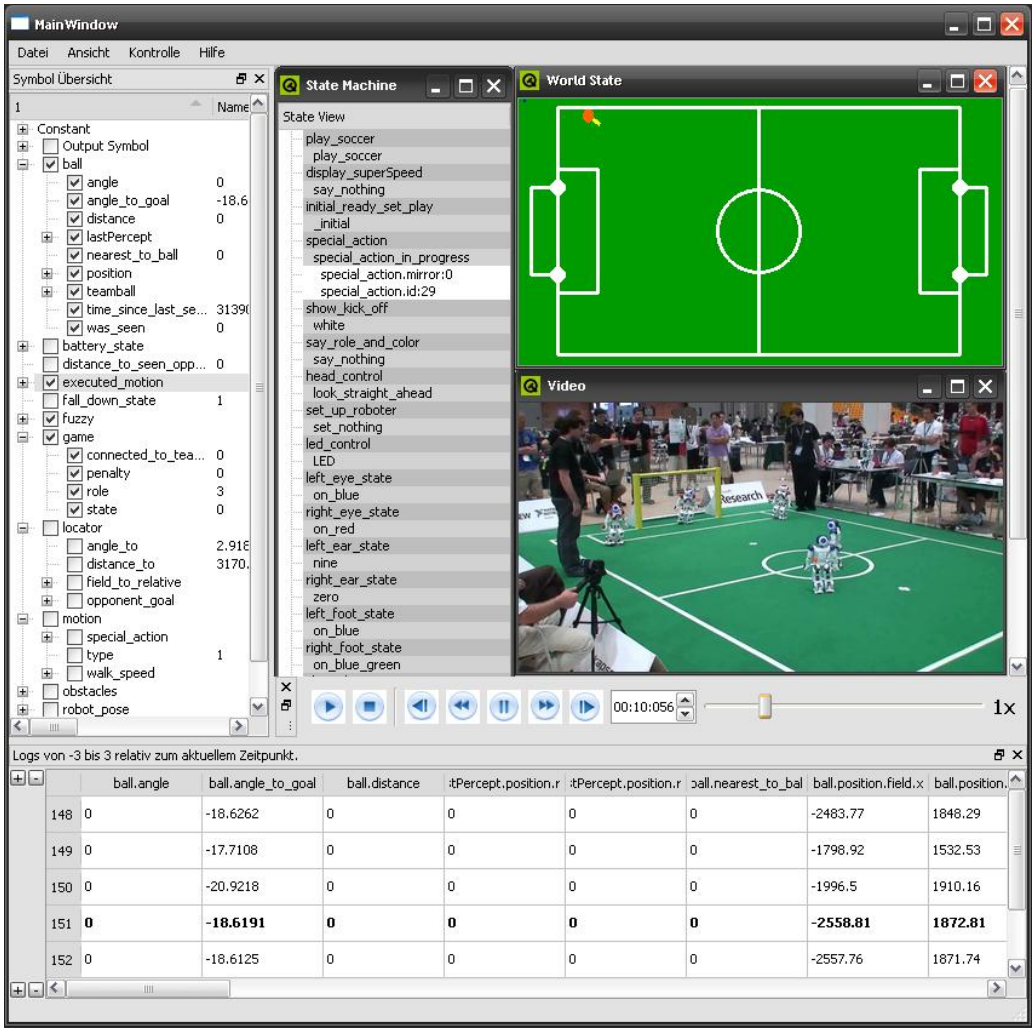


Figure 4.5: XABSL Debug Tool.

- Video Widget: Display of the video file synchronized to the internal robot state.
- Control Widget: The user can navigate within the logfile including jump forward, back step by step, play slow-motion etc.
- Statemachine Widget: This Widget shows the current option-state path.

All this is designed to enable a behavior designer to replay all the states and analyze the decisions a robot made. It is therefore easy to find out,

- whether a decision was based on a false perception: If the robot would have correctly seen a certain feature, it would have behaved correctly. This kind of error results from either a false perception or wrong interpretation and modeling.
- whether the decision was based on false decisions made on correct knowledge: The user observes a gamestate which resulted in an undesired decision. If the internal world model matches the real world the wrong decision is a result of a decision error or a coding bug.

The tool is still in an early state of development missing features such as synchronizing log files of different robots to debug team decisions. But still the debugging tool has proven helpful in replaying gamestate decisions and allowing expert programmers to judge and debug errors thereby improving the written XABSL code. The most useful information was thereby gained from actual tournament games which were otherwise impossible to supervise due to the given rules.

4.5 Current Research

4.5.1 Artificial immune network and XABSL

Using XABSL for behavior development has several disadvantages: The more information about the world is available, the more difficult is it to model transitions between XABSL states in an appropriate way. A lot of symbols has to be used and combined for decision making. Hence, behavior development turns often out as a time-consuming, error-prone process. Furthermore, XABSL on its own is not a learning system and the success of the behavior is therefore completely dependent on expert specifications. Due to this limitations one can say that XABSL is suitable for the specification of less complex behaviors, e.g. going to the ball or to a certain position, searching, kicking and passing the ball.
The main goal of our current research is it to develop a method in order to select the next action of the soccer agent. The action selection mechanism uses local and distributed information and is based on an artificial immune network. Figure 4.6 shows the interface between XABSL and the action selection mechanism: The option *decide* executes the artificial immune network algorithm, interprets the result and calls an underlying basic behavior option. The immune network algorithm uses also the defined XABSL symbols, e.g. the position of the ball or shared information between the robots (which is a consolidated, common model of the robot soccer world). The action selection mechanism can be seen as an extension of XABSL.

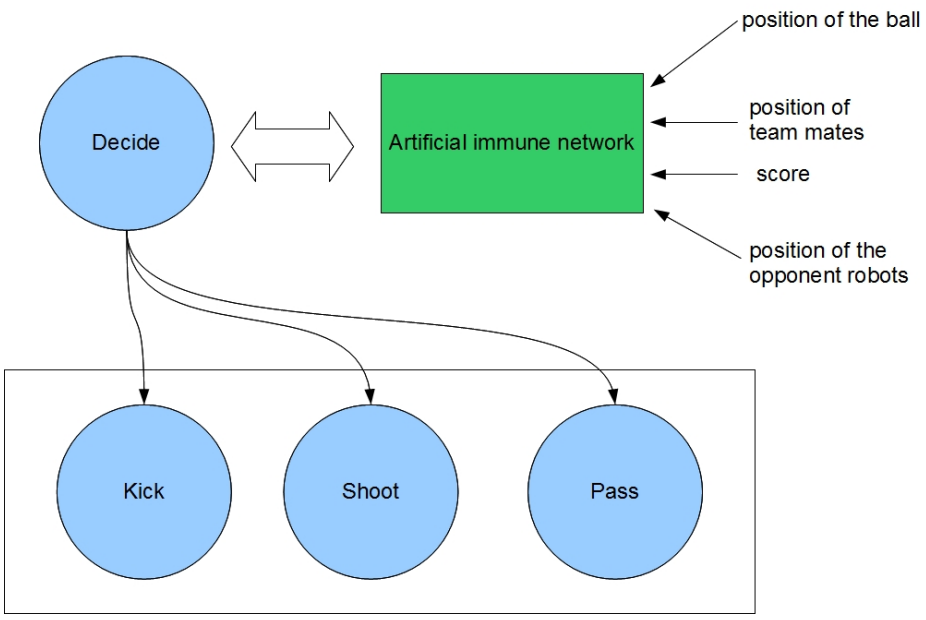


Figure 4.6: Interface between XABSL and artificial immune network algorithm.

4.5.2 Artificial immune network algorithm

This subsection provides a short outline of the artificial immune network algorithm and its application in the robot soccer domain. A general introduction concerning artificial immune systems, its biological background and its main algorithms can be found at [22]. Jerne defines an artificial immune network as a mathematical model in which the immune system can be viewed as a network of interacting cells [23]. In order to understand the immune network algorithm and the structure of its implementation, it is necessary to clarify a number of metaphors:

1. *Antigens* (i.e. pathogens) are cells which invaded a (vertebrate) body, e.g. virus. In the context of robot soccer, an antigen is the representation of the current state of the robot which is simply a vector consisting of information about the world on fixed point in time t.

2. *Antibodies* are produced as an reaction of the immune systems against pathogens. They represent the (pre-defined and fixed) possible actions which can be selected from the robot at any point of time.

3. *Affinity function* is a function which calculates the degree of stimulation (match) between the current antigen and each antibody. They can be implemented with a fuzzy inference system as shown in [24].

4. *T-helper cells* control the process of proliferation[1] with stimulation and suppression. They can be used in order to implement reinforcement learning algorithms as done

[1] Proliferation is the process of adapting and cloning antibodies

in [25].

Referring to the immune network theory, the number of antigens are determined trough stimulation of the antigens and inter-depend stimulation and suppression of the other types of antibodies. The following differential equation 4.1 computes the rate of change of the antibody concentration at a fixed point of time t [24].

$$\frac{dA_i}{dt} = (\sum_{l=1}^{N_{Ab}} m_{il}^{st} a_l(t) - \sum_{k=1}^{N_{Ab}} m_{ki}^{su} a_k(t) + m_i - k_i) a_i(t) \tag{4.1}$$

$$a_i(t) = \frac{1}{1 + exp(0.5 - A_i(t))} \tag{4.2}$$

N_{Ab} means the number of types of antibodies. A_i and a_i are concentration and stimulus of the i-th type of antibody, i,l and k subscripts to distinguish between several types of antibodies. k_i models a natural death rate of antibodies in the absence of any interaction, m_i refers to the affinity between antigen and each type of antibody, m_{il}^{st} and m_{ki}^{su} are showing the stimulative and the suppressive interaction respectively between antibodies. The antibody with the highest concentration represents the selected action at time t [24]. Equation 4.2 is part of the Jernes immune network model and stabilizes the dynamics of the system[22].

4.5.3 Optimizing the parameters

Another main issue of our research work is it to find out which actions are successful against certain types of play. For example, which actions should be preferable selected against dribbling teams. Therefore, a fixed reference behavior has to be determined. The AIN[2]-controller based behavior plays against the specified reference behavior in simulation. The adaption of the AIN-controller is performed through parameter optimization[3] over time. The objective function for the optimization can be composed of score, spatial and temporal distribution of the ball on the field. One execution of the objective function is a fixed time in the game.
The result of the optimization depends on the reference behavior, it is likely that each type of play needs another set of parameters. Furthermore, the repertoire of basic behaviors (which means antibodies or actions that can be selected) has a significant influence on the result.

[2]artificial immune network

[3]the parameters of the AIN are the affinity functions, the degree of interdependent stimulation and inhibition of the types of antibody and the death rate parameter

Chapter 5

Conclusion and Outlook

The *Nao Devils Dortmund* is a team from the TU Dortmund University with roots in several other teams which have competed in RoboCup competitions over the last years. In RoboCup 2010 the second round robin pool has been reached, missing the quarter finals by a close shave. The 10th place was achieved in the ranking of Technical Challenges 2010.

Experience from this years competition has revealed that much more tackling is involved during games than in last year's tournament. Thus for next year the Nao Devils will focus on the development of more sophisticated team play behavior. By adding visual robot detection the opponent awareness and avoidance should be greatly complemented. Also it's planned to integrate the briefly proposed multi target tracking approach into the behavior to allow a more strategical positioning and advanced tactical play of the robots. A more advanced team ball localization should help the ball detection even during crowded situations on the field. In addition part of the focus will still lie on motion generation, refining and extending the existing walking algorithms and further develop the path planning and motion execution.

Appendix A

Getting Started

A.1 Setting up the development environment

The framework can be compiled on windows and linux operating systems.

A.1.1 Windows

1. Install Visual Studio 2008 Professional or Visual Studio 2010.
2. Install Cygwin 1.7.5_1 or higher. Use the 'install from internet' option (see figure A.1). When coming to the screen "Cygwin Setup - Select Packages" switch the view to "Full" and add the following additional packages (see figure A.2): bash, libxml2, libxslt, make, openssh, python, ruby, rsync
3. Furthermore, the following directories have to be added to the PATH variable in your system environment: C:\cygwin\lib and C:\cygwin\bin (if you have not installed cygwin into that directory, change it accordingly).
4. Unzip the code release source package to a directory of your choice.
5. Set up the cross compiler: Copy the folder <coderelease directory>/Util/i586-opennao-linux-gnu into the Cygwin directory (e.g. /home/user/ or /usr/cross compiler). Open the file 'g++' which is located at <coderelease directory>/Make/crosstool and the change the path of CXX to the cross compiler directory like that: CXX=/usr/i586-opennao-linux-gnu/bin/i586-opennao-linux-gnu-g++.exe.
6. You should now be able to compile code for the robot. Open NDevils.sln, it includes several projects for Simulator, libbhuman and the ndevils main program.
7. Additional software is needed to connect with and load files onto the robot. We recommend using Putty (`http://www.chiark.greenend.org.uk/~sgtatham/putty/download.html`) as a remote console interface and WinSCP (`http://winscp.net/eng/download.php`) as SCP client software.
8. If you want to change the parameters of the Dortmund Walking Engine you may need Matlab 2007a or higher including the Control System Toolbox.

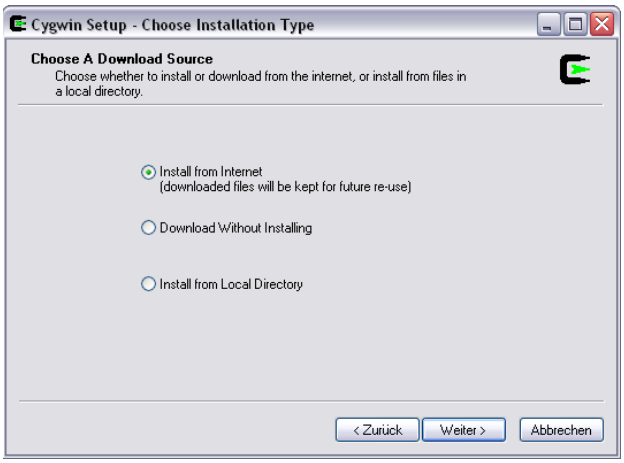


Figure A.1: Cygwin Setup.

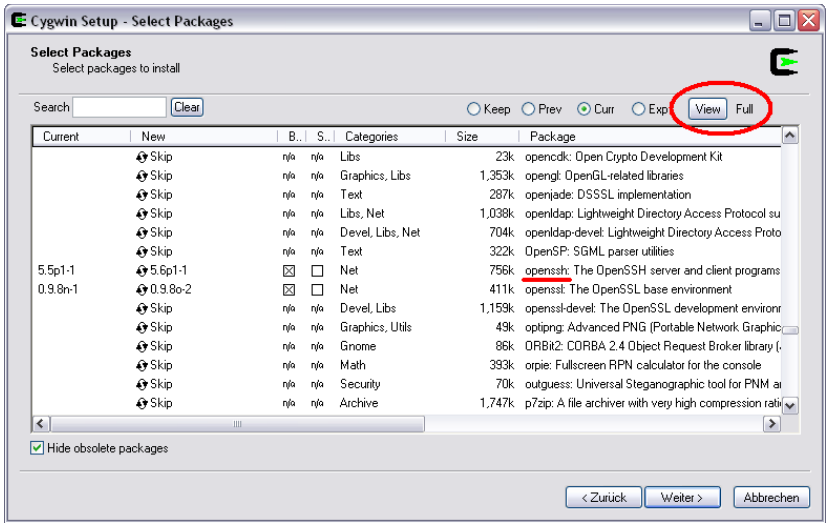


Figure A.2: Installing additional Cygwin packages.

A.1.2 Linux

You can use any Linux distribution as well to compile the robot code. Simply unzip the source package into a directory of your choice. Like in the windows setup, bash, ruby, gnu make, ssh, rsync, an xslt parser software and python have to be installed. Adapt the g++ file as described at the windows section. Switch to the Make directory of the coderelease directory and run make (e.g. with the parameters <Nao|libbhuman|Behavior|SpecialActions> Config=<Debug|Release>). The Linux makefiles for each project are generated automatically with zbuild. Note that the Simulator can not be compiled under Linux Platforms.

A.1.3 Build configurations

Our code release consists of two different build configurations, RELEASE and DEBUG. The DEBUG configuration makes it possible to connect with SimRobot. The actual state of the robot can be shown and modified. Running the robot with the DEBUG configuration will result in frame drops which means that the Cognition and Motion thread can not be executed stable at 33 Hz and 100 Hz respectively. Use RELEASE when high performance is required. The RELEASE configuration ignores every source which is needed for debugging except Profiling features.

A.2 Setting up the robot

Setting up the robot to run the newly compiled code consists of several steps. The memory sticks of the Nao V3+ robots needs to be flashed once for each robot as described in the following section. Each time a different user works with a robot he needs to set up his configuration (see section A.2.2). Section A.2.3 and A.2.4 finally describe how to run and debug code on the robot.

A.2.1 Preparing the memory stick

Linux is required in order to set up a new stick for the Nao robot. This only needs to be done once per robot.

1. Download opennao-robocup-1.6.13-nao-geode.ext3.gz from the Aldebaran website and copy it into /images of the Install folder (or use -i ¡image.ext3¿).
2. Plug the flashstick in your pc USB socket.
3. Run ./flash-and-install.sh as an administrator, e.g.: sudo ./flash-and-install.sh under Ubuntu.

The script uses Connman to connect the wireless LAN. Therefore, a few more steps are needed to configure the wireless connection:

1. After flash-and-install.sh is executed, the Nao should be booted with the new USB-stick and a connected network cable.
2. Press the chest button of the Nao in order to get the current IP of the robot.

3. Open the IP with a browser from a PC which is in the same network as the robot (user:nao pw:yourpass).

4. Use the webinterface to change the robot's name and connect to your local wireless LAN.

5. Connect to the robot via ssh (user:root pw:yourpass) and execute ./firstStart.sh which removes unneeded services (not including the webinterface) and changes the autoload.ini .(This process can be reverted, by executing ./reset2Aldebaran.sh)

6. Reboot the Nao. The setup process is finished.

A.2.2 Copying your current code and configuration to the robot

Multiple users working with the same robot often need different configurations or code, so it is important to have an automated process to make sure every necessary file on the robot is up to date. This copyfiles script makes it easy to set up the robot. All needed files are copied into the designated directories.

Parameter	**Bedeutung**
-r <robotname>	set robotName
-u <username>	set the username (name of the configuration files directory)
-l <location>	set location (e.g. useful for different colortables)
-t <color>	set team color to blue or red
-p <number>	set player number. Note that two parallel operated robots shpuld not have the same robot number.
-d	delete remote cache first

Table A.1: Parameters of the copyfiles script

Examples: copyfiles.sh [Debug|Release] [<IP address>|(-m n <iP address>)*] (i.e. ./copyfiles.sh Debug 134.102.204.229 -p 1).

A.2.3 Starting and stopping the robot

Starting and stopping the NAO can be proceeded as follows:

1. Switch on the robot (press the chest button).

2. Wait until the eye LEDs are switched off.

3. Open Putty and connect to the robot.

4. The framework and NaoQi are booted by default. The connection with SimRobot will work better if you reboot both. To shut down / boot up the framework type ./ndevils stop or ./ndevils start respectively. Starting/Stopping NaoQi is done by typing nao stop / nao start. All Boot Scripts are located at /root on the stick.

5. Type halt to shut down the robot.

A.2.4 How to connect with the robot and debug

SimRobot can be used to connect with the robot. Simply, you have to modify the file *connect.con* which is located under <Coderelease>/Config/Scenes. Replace the IP address in the command *sc Remote 192.168.1.2*. Boot up SimRobot and open the Scene *RemoteRobot.con*.
The following table shows a set of basic Simulator commands which are useful for debugging purposes.

Parameter	**Description**
log start	starts recording a logfile.
log stop	stops recording a logfile.
log save <filename>	save the actual log to a specified file. The saved file is located at <Coderelease>/Config/Logs.
vi image raw	Transmits uncompressed raw images from the robot to the Simulator.
vid raw representation: <class>: <element>	draws recognized elements on the image view. It is useful in order to determine if the robot detects balls, goal posts etc.
vf worldState	creates a worldState view on which additional 2D information can be drawn
vfd worldState representation: <class>: <element>	draws elements on the worldState view. This can be e.g. ball percepts, robot poses etc.

Table A.2: Parameters of the copyfiles script

A detailed description of SimRobot and its commands can be found in [7] chapter 8. Please consult the manual for creating a colortable as well.

A.2.5 Using the XABSL Debug Tool

The XABSL debug logging can be activated in the 'behavior.cfg' file which can be found on the robot under '[your Config]/Locations/[your location]'. Now create the directory for the log(s) : the debug tool expects a 'Logs' folder in your config folder on the robot. As the last step add logging for your XABSL symbols by editing the respective .cpp file as following :

- For static symbols (e.g. constant symbols or field symbols) add a method debugLogStatics() to your c++ symbol files (see Listing A.1).
- For every-frame-updated symbols add a debugLog() and debugLogPrintTitle() method to your c++ smybol files (see Listing A.2).

```
/** Static Debug Symbol Logging **/
void ConstantSymbols::debugLogStatics()
{
  DebugSymbolsLog::printStaticSymbol("constants.pi",pi);
  DebugSymbolsLog::printStaticSymbol("constants.max_pan",maxPan);
  DebugSymbolsLog::printStaticSymbol("constants.min_pan",minPan);
  // .. and so on
}
```

Listing A.1: "static symbol logging"

```
/** ColumnCaption - only once per log **/
void LocatorSymbols::debugLogPrintTitle()
{
  DebugSymbolsLog::print("locator.pose.x");
  DebugSymbolsLog::print("locator.pose.y");
  DebugSymbolsLog::print("locator.pose.angle");
  // .. and so on
}

/** Every Frame Logging **/
void LocatorSymbols::debugLog()
{
  DebugSymbolsLog::print(robotPose.translation.x);
  DebugSymbolsLog::print(robotPose.translation.y);
  DebugSymbolsLog::print(getPoseAngle());
  // .. and so on
}
```

Listing A.2: "every frame symbol logging"

Appendix B

Framework

The German Team Framework [6] was chosen to be the basis for all code being written. Since both B-Human and the Microsoft Hellhounds were using variations or predecessors of this framework large amounts of already available modules (e.g. ImageProcessing, BallModelling) could be ported rapidly and with ease. A more detailed description of the most recent version of this framework can be found in [7].

The framework itself is based on a modular structure. A module can be seen as a possible solution for a certain task. This could either be self-localization, image-processing or something else. Modules can be exchanged at runtime which simplifies the comparison and evaluation of different solutions for a specific task. The coupling between the modules is created by so called representations which can either be required or provided by a specific module. The modules itself are grouped in so called processes. For now there exist only two processes. The motion process encapsulates all modules concerning the generation of motion trajectories. This includes for example the WalkingEngine or the SpecialActionEngine. The cognition process however contains modules for imageprocessing, localization and bahavior control. The data exchange between the framework processes is done via means of Inter-Process Communication. For now the data is exchanged via a shared-memory system.

B.1 Modules

As mentioned before modules are the essential elements of the framework. A module consists of three parts:

- the module interface
- its actual implementation
- a statement that allows to instantiate the module.

The module interface defines the name of the module, the representations required by the module to operate and the representations it provides/modifies. This interface basically creates a basis for the actual implementation. An example module definition can be seen in listing B.1.

```
/** Module Definition **/
MODULE(DemoImageProcessor)
        /** Needs Image Representation **/
        REQUIRES(Image)
        /** Needs CameraMatrix Representation **/
        REQUIRES(CameraMatrix)
        /** Needs FrameInfo Representation **/
        REQUIRES(FrameInfo)
        /** Provides BallModel Representation **/
        PROVIDES(BallModel)
        /** Provides PointsPercept Representation
        with debugging capabilities**/
        PROVIDES_WITH_MODIFY_AND_OUTPUT(PointsPercept)
END_MODULE
```

Listing B.1: "Example Module Definition"

The actual implementation of the module is done in a class derived from the module definition. The implementation has class-wide read-only access to the required representations. For each provided representation an update method with a reference to the representation as parameter has to be implemented (e.g. **void update(BallModel&)**).

At last the module needs to be instantiated, this is done via a call to *MAKE_MODULE*, which takes a category as its second parameter allowing to specify a category for debugging purposes. Listing B.2 shows as an example how to implement an module from the previous module definition.

B.2 Representations

Representations are used to exchange data between modules. They are essentially light weight data structures, which only contain the necessary information needed and provided by a module. Additionally all representations must implement an serialization interface called *Streamable* which on the one hand enables the exchange of data between framework processes and on the other hand is used for debugging purposes. Listing B.3 shows an example Representation.

B.3 Origin of Modules

The code released together with this report is based on the most recent version of the *GermanTeam* framework originating from the code release by *BHuman* in 2009. As described earlier this is used due to the *Nao Devils*' history both in the *GermanTeam* and the *BreDoBrothers*. Most of the code related to infrastructure and framework-internals is left unchanged. The modules containing cognition and motion algorithms however are mostly developed in Dortmund. An overview about the origins of these modules is given in tables B.1 and B.2. Infrastructure modules, e.g. those containing the code to obtain images and sensor readings or to read or write joint angles, are omitted in those tables.

```
/** Module Implementation **/
/** Implement DemoImageProcessor derived from Module Definition**/
class DemoImageProcessor : public DemoImageProcessorBase
{
        void update(BallModel& pBallModel)
        {
                //update the BallModel representation
        }
        void update(PointsPercept& pPointsPercept)
        {
                //update the PointsPecept representation
        }
};

/** Module Instantiation **/
/** Instantiate Module DemoImageProcessor
        in category "ImageProcessing" **/
MAKE_MODULE(DemoImageProcessor, ImageProcessing)
```

Listing B.2: "Example module implementation and instantiation"

Modules with origin in "Bremen" are kept from the *BHuman* code release for various reasons, because they were either kept from the *BreDoBrothers* cooperation (e.g. the ParticleFilterBallLocator), equivalent in function (as the RobotModelProvider [7] compared to the EgoModelProvider [26]) or providing new functionality (as the ObstacleModelProvider for the sonar filtering). Origin "Dortmund" indicates *Nao Devils*' developments or late *Microsoft Hellhounds* code. "*GermanTeam*" marks those parts of the code which were not necessarily in this exact modul form, but nevertheless date back to 2005 or before and were developed together by the *GermanTeam* of which both Bremen and Dortmund have been members.

```
class BallPercept : public Streamable
{
        /** Streaming function
        * @param in  streaming in
        * @param out streaming out
        */
        void serialize(In* in, Out* out)
        {
                STREAM_REGISTER_BEGIN();
                STREAM(positionInImage);
                STREAM(radiusInImage);
                STREAM(ballWasSeen);
                STREAM(relativePositionOnField);
                STREAM_REGISTER_FINISH();
        }

public:
        /** Constructor */
        BallPercept() : ballWasSeen(false) {}
        /**< The position of the ball in the current image */
        Vector2<double> positionInImage;
        /**< The radius of the ball in the current image */
        double radiusInImage;
        /**< Indicates, if the ball was seen in the current image. */
        bool ballWasSeen;
        /**< Ball position relative to the robot. */
        Vector2<double> relativePositionOnField;
};
```

Listing B.3: "Example module implementation and instantiation"

Table B.1: Motion Module Overview

Module	Function	Origin
ArmAnimator	Controls arm movement in Dortmund Walking Engine	Dortmund
HeadMotionEngine	Controls head movements	GermanTeam
GroundContactDetctor	Checks ground contact (used in Inertia Matrix)	Bremen
InertiaMatrixProvider	Calculates the basis for the Camera Matrix (previous CameraMatrix calculation dates back to the GermanTeam)	Bremen
LimbCombinator	Puts leg- and armjointrequest from Dortmund Walking Engine together	Dortmund
MotionCombinator	Combines requests from all motion modules to create a final joint request	GermanTeam
MotionSelector	Switches between motion types	GermanTeam
NaoKinematic	Inverse kinematic chain calculations	Dortmund
PatternGeneratorModule	Creates the footsteps for Dortmund Walking Engine	Dortmund
RequestTranslatorModule	Creates a pattern request from desired movement speed/target	Dortmund
RobotModelProvider	Provides informations about the robot model (this module is equivalent to the EgoModelProvider [26] and kept instead of porting the latter)	Bremen
SpecialActions	provides special action (hardcoded motions)	GermanTeam
ZMPIPControllerModule	Controls ZMP while walking (older version)	Dortmund
ZMPIPControllerModule2009	Controls ZMP while walking (in use)	Dortmund
ZMPModelProvider	Calculates ZMP used in Dortmund Walking Engine	Dortmund

Table B.2: Cognition Module Overview

Module	Function	Origin
CameraMatrixProvider	Provides the transformation from camera to the robot coordinate frame (a slight change in the functionality requires the InertiaMatrix now)	GermanTeam
MSHImageProcessor	Provides the basic scanning routines described in section 3.1	Dortmund
BodyContourProvider	Provides a projection of the body contour to the camera image	Bremen
BallPerceptor	Verifies the detection of possible balls given certain ball candidates	GermanTeam
NDLineFinder2	Finds field lines, crossings and the center circle as described in section 3.2	Dortmund
NDContextProcessor	Extends the information of certain percepts using context of others	Dortmund
ParticleFilterBallLocator	Estimates the ball position (used since BreDoBrothers 2008)	Bremen
ObstacleModelProvider	Models obstacles in a local coordinate system based on sonar readings	Bremen
UKFSampleTemplateGenerator	Provides regions of high localization probability based on recent percepts only	Dortmund
MultiUKFSelfLocator	Self localization of the robot as described in section 3.3	Dortmund
Predictor	Transforms certain models into a coordinate system based on the point at which behavior decisions take effect (see section 4.1)	Dortmund
BehaviorControl	XABSL behavior control (see section 4)	GermanTeam

Appendix C

Walking Engine Parameters

In this section the most important parameters to control the Walking Engine are described. They can be found in the file <coderelease directory>/Config/walkingParams.cfg. Some parameters needed by the ZMP/IP-Controller are located in the file <coderelease directory>/Util/Matlab/NaoV3.m. The Matlab script writeParamsV3.m calculates the values used by the ZMP/IP-Controller using the parameters in this file and stores the results to <coderelease directory>/Config/Robots/<robot name>/ZMPIPController.dat. To execute the script open Matlab, change the current directory to <coderelease directory>/Util/Matlab and enter the following command:

```
writeParamV3(NaoV3('<robot name>'))
```

All values are expressed in SI units, unless otherwise stated.

C.1 File walkingParams.cfg

The file walkingParams.cfg consists of the following parameters:

footPitch

To avoid hitting the ground with the forward section of the foot an offset is applied to the foot rotation around the y axis. The added value reaches its maximum at the middle of the single support phase. The maximum can be set using footPitch.

xOffset

The center of the feet can be shifted along the x axis by setting this value unequal to 0. This offset is constant for the whole walk.

stepHeight

Maximum z position of the foot during the single support phase, see section 2.1.4.

sensorControlRatio

The sensorControlRatio is multiplied with the difference between the target ZMP and the measured ZMP. Legal values are $[0 \dots 1]$, where 0 means 'no sensor control' and 1 'full sensor control'.

doubleSupportRatio

Proportion of the double support phase during a step.

crouchingDownPhaseLength, startingPhaseLength, stoppingPhaseLength

These values define the duration of the transitions between a walk and special actions.

armFactor

The movement of an arm depends on the x coordinate of the opposite foot. The x coordinate is multiplied with armFactor and applied to the ShoulderPitch.

arms1

This value is the angle of ShoulderRoll for both arms. It is constant during the walk.

zmpSmoothPhase

To avoid hard sensor feedback during transitions from special actions to walk and vice versa the ZMP error is faded in and out. This parameters sets the length in frames of this phase.

maxSpeedXForward, maxSpeedXBack, maxSpeedYLeft, maxSpeedYRight, maxSpeedR

These values define the maximum speed. All requests sent to the Walking Engine are clipped to these values.

stepDuration

The PatternGenerator defines the footsteps based on two single support phases and two double support phases. The duration of all together is the stepDuration.

footYDistance

Distance between the feet.

stopPosThresholdX, stopPosThresholdY, stopSpeedThresholdX, stopSpeedThresholdY

Stopping a walk and executing a special action is only possible when the Walking Engine sets a flag which indicates that it does not compromise the stability. The Walking Engine uses this 4 indicators to check if the speed of the center of mass is low enough and the position within a stable range.

outFilterOrder

Before sending the angles to the robot they are low-pass filtered. This parameter defines the order of the filter.

sensorDelay

The sensor control relies on a good comparison between the measured ZMP and the target ZMP. In most cases the measurements are some frames old and must be compared with the target of the same frame.

maxFootSpeed, fallDownAngle

The Walking Engine has an integrated check for instabilities to avoid breaking joints. Due to the sensor control the robot can react in an unexpected way resulting in very fast movements. The maximum speed of the feet can be limited by using maxFootSpeed. Additionally all movements are stopped immediately when the robot exceeds the maximum tilt or roll angle defined by fallDownAngle.

polygonLeft, polygonRight

As mentioned in section 2.1.3 the reference ZMP is calculated using a Bézier curve. This parameters define the y coordinates of the control points in the corresponding foot coordinate system.

offsetLeft, offsetRight

Due to high torques acting on the joints during the single support phase large angle errors can be observed. To compensate the effects a constant offset is added to each leg joint which can be configured using these parameters.

walkRequestFilterLen

To avoid abrupt speed changes a low-pass filter can be activated to filter the incoming speed requests. A value of 1 means deactivated.

maxAccX, accDelayX, maxAccY, accDelayY, maxAccR, accDelayR

Besides the low-pass filter there is an other way to limit speed changes. If the acceleration resulting from the new speed request is higher than maxAcc only maxAcc is applied until accDelay frames are elapsed.

speedApplyDelay

This is the third way to limit speed changes. If a speed change has been detected further speed changes are ignored for speedApplyDelay frames.

legJointHardness

The stiffness parameter for the leg joints. The stiffness cannot be set for each leg separately.

heightPolygon

This five values are multiplied with stepHeight to get the z coordinates of the control polygon for the swinging leg as explained in section 2.1.4.

C.2 File NaoV3.m

The following parameters can be found in the file NaoV3.m:

g

The Gravity.

z_h

Target height of the center of mass over the ground.

dt

Length of one frame.

R

Controller-Parameter R [13].

N

Duration of the preview phase as explained in section 2.1.2.

Qx

Controller-Parameter Qx [13].

Qe

Controller-Parameter Qe [13].

Ql

Gain for calculating L [13].

RO

Gain for calculating L [13].

path

Path to the file ZMPIPController.dat.

Bibliography

[1] Czarnetzki, S., Kerner, S.: Nao Devils Dortmund Team Description for RoboCup 2009. In Baltes, J., Lagoudakis, M.G., Naruse, T., Shiry, S., eds.: RoboCup 2009: Robot Soccer World Cup XII Preproceedings, RoboCup Federation (2009)

[2] Czarnetzki, S., Hebbel, M., Kerner, S., Laue, T., Nisticò, W., Röfer, T.: BreDoBrothers Team Description for RoboCup 2008. In Iocchi, L., Matsubara, H., Weitzenfeld, A., Zhou, C., eds.: RoboCup 2008: Robot Soccer World Cup XII Preproceedings, RoboCup Federation (2008)

[3] Hebbel, M., Nisticò, W., Kerkhof, T., Meyer, M., Neng, C., Schallaböck, M., Wachter, M., Wege, J., Zarges, C.: Microsoft hellhounds 2006. In: RoboCup 2006: Robot Soccer World Cup X RoboCup Federation, RoboCup Federation (2006)

[4] Röfer, T., Brunn, R., Czarnetzki, S., Dassler, M., Hebbel, M., Jüngel, M., Kerkhof, T., Nisticò, W., Oberlies, T., Rohde, C., Spranger, M., Zarges, C.: Germanteam 2005. In Bredenfeld, A., Jacoff, A., Noda, I., Takahashi, Y., eds.: RoboCup 2005: Robot Soccer World Cup IX Preproceedings, RoboCup Federation (2005)

[5] Czarnetzki, S., Hebbel, M., Nisticò, W.: DoH!Bots Team Description for RoboCup 2007. In: RoboCup 2007: Robot Soccer World Cup XI Preproceedings. Lecture Notes in Artificial Intelligence, Springer (2007)

[6] Röfer, T., Laue, T., Weber, M., Burkhard, H.D., Jüngel, M., Göhring, D., Hoffmann, J., Altmeyer, B., Krause, T., Spranger, M., Schwiegelshohn, U., Hebbel, M., Nisticó, W., Czarnetzki, S., Kerkhof, T., Meyer, M., Rohde, C., Schmitz, B., Wachter, M., Wegner, T., Zarges, C., von Stryk, O., Brunn, R., Dassler, M., Kunz, M., Oberlies, T., Risler, M.: GermanTeam RoboCup 2005. Technical report (2005) Available online: `http://www.germanteam.org/GT2005.pdf`.

[7] Röfer, T., Laue, T., Müller, J., Bösche, O., Burchardt, A., Damrose, E., Gillmann, K., Graf, C., de Haas, T.J., Härtl, A., Rieskamp, A., Schreck, A., Sieverdingbeck, I., Worch, J.H.: B-human team report and code release 2009 (2009) Only available online: `http://www.b-human.de/file_download/26/bhuman09_coderelease.pdf`.

[8] Laue, T., Spiess, K., Röfer, T.: Simrobot - a general physical robot simulator and its application in robocup. In Bredenfeld, A., Jacoff, A., Noda, I., Takahashi, Y., eds.: RoboCup 2005: Robot Soccer World Cup IX. Number 4020 in Lecture Notes in Artificial Intelligence, Springer; http://www.springer.de/ (2006) 173–183

[9] Vukobratovic, M., Borovac, B.: Zero-moment point – Thirty five years of its life. International Journal of Humanoid Robotics **1** (2004) 157–173

[10] Kajita, S., Kanehiro, F., Kaneko, K., Fujiwara, K., Yokoi, K., Hirukawa, H.: Biped walking pattern generator allowing auxiliary zmp control. In: IROS, IEEE (2006) 2993–2999

[11] Kajita, S., Kanehiro, F., Kaneko, K., Fujiwara, K., Harada, K., Yokoi, K., Hirukawa, H.: Biped walking pattern generation by using preview control of zero-moment point. In: ICRA, IEEE (2003) 1620–1626

[12] Katayama, T., Ohki, T., Inoue, T., Kato, T.: Design of an optimal controller for a discrete-time system subject to previewable demand. International Journal of Control **41** (1985) 677 – 699

[13] Czarnetzki, S., Kerner, S., Urbann, O.: Observer-based dynamic walking control for biped robots. Robotics and Autonomous Systems **57** (2009) 839–845

[14] Czarnetzki, S., Kerner, S., Urbann, O.: Applying dynamic walking control for biped robots. In: RoboCup 2009: Robot Soccer World Cup XIII. Lecture Notes in Artificial Intelligence, Springer (2010) 69 – 80

[15] Czarnetzki, S., Kerner, S., Hegele, M.: Odometry correction for humanoid robots using optical sensors. In: RoboCup 2010: Robot Soccer World Cup XIV. Lecture Notes in Artificial Intelligence, Springer (2011) to appear

[16] Hebbel, M., Kruse, M., Nisticò, W., Wege, J.: Microsoft hellhounds 2007. In: RoboCup 2007: Robot Soccer World Cup XI Preproceedings, RoboCup Federation (2007)

[17] Nisticò, W., Hebbel, M.: Temporal smoothing particle filter for vision based autonomous mobile robot localization. In: Proceedings of the 5th International Conference on Informatics in Control, Automation and Robotics (ICINCO). Volume RA-1., INSTICC Press (2008) 93–100

[18] Quinlan, M.J., Middleton, R.H.: Multiple model kalman filters: A localization technique for robocup soccer. In: RoboCup 2009: Robot Soccer World Cup XIII. Lecture Notes in Artificial Intelligence, Springer (2010) 276–287

[19] Czarnetzki, S., Kerner, S., Kruse, M.: Real-time active vision by entropy minimization applied to localization. In: RoboCup 2010: Robot Soccer World Cup XIV. Lecture Notes in Artificial Intelligence, Springer (2011) to appear

[20] Czarnetzki, S., Rohde, C.: Handling heterogeneous information sources for multi-robot sensor fusion. In: Proceedings of the 2010 IEEE International Conference on Multisensor Fusion and Integration for Intelligent Systems (MFI 2010), Salt Lake City, Utah (2010) 133 – 138

[21] Lötzsch, M., Bach, J., Burkhard, H.D., Jüngel, M.: Designing agent behavior with the extensible agent behavior specification language XABSL. In Polani, D., Browning, B., Bonarini, A., eds.: RoboCup 2003: Robot Soccer World Cup VII. Volume 3020 of Lecture Notes in Artificial Intelligence., Padova, Italy, Springer (2004) 114–124

[22] Castro, L.N.d.: Artificial Immune Systems: A New Computational Intelligence Approach. Springer-Verlag, London (2002)

[23] Jerne, N.K.: The immune system. Scientific American **229** (1973) 52–60

[24] Luh, G.C., Wu, C.Y., Liu, W.W.: Artificial immune system based cooperative strategies for robot soccer competition. In: The 1st International Forum on Strategic Technology. (2006) 76–79

[25] Wang, Y.T., You, Z.J., Chen, C.H.: Ain-based action selection mechanism for soccer robot systems. In: Journal of Control Science and Engineering. (2009) 10p

[26] Czarnetzki, S., Kerner, S., Urbann, O., Abelev, J., Bökkerink, C., Hofmann, M., Ibisch, A., Jalali, K., Kuhnert, M., Ollendorf, M., Schorlemmer, P., Stumm, S., , Weber, M.A.: Nao devils dortmund team report for robocup 2009. Technical report, Robotics Research Institute, TU Dortmund University (2009)