Team Research Report 2009

Prof. Dr. Karl-Udo Jahn,
Daniel Borkmann, Thomas Reinhardt, Rico Tilgner,
Nils Rexin, Stefan Seering

Leipzig University of Applied Sciences,
Faculty of Computer Science, Mathematics, and Natural Sciences

November 17, 2009

naohtwk@gmail.com
http://naoteam.imn.htwk-leipzig.de

Contents

1 System Architecture

1.1 Languages

Our software is written in C, Java and Perl. The core system which runs on the Nao itself is written in C. The build system for cross compiling contains useful scripts which are written in Perl and last but not least the control UI that runs on a client computer is written in Java. In general, the core system language decision was based on facts like performance and simplicity, we preferred C instead of C++. The core system has been implemented as a framework called NIO Framework and runs as an independent process on Naos Geode platform.

1.2 Source Code Management

As our revision control system we use git. git was initially designed for the Linux Kernel Development and is a free, performant and simple distributed control system. Important advantages like distributed development, a strong support for nonlinear development and the possibility of an efficient handling of large projects convinced us to make use of git.

1.3 NIO Framework

1.3.1 Motivation

Our NIO Framework is an independent peace of software that runs on Nao robots and extends the Aldebaran Robotics NaoQI framework. The term NIO stands for Nao Input Output.

The motivation for creating our own framework:

- Inconsistency of NaoQIs API
- Very limited debugging capabilities of NaoQI framework
- No need for a time intensive NaoQI restart after changes to parts of the software (e.g. motions, strategy)
- No thread safetyness of certain NaoQI calls
- Lots of NaoQI functionality we actually don't need
- Possibility of writing plain C instead of C++ code

1.3.2 Architecture and Core Components

The basic functionality of our NIO Framework is defined by a Unix Domain Socket client server pair. We have built a simple C++ module for the NaoQI framework that exports a subset of the NaoQI calls through the socket to the requesting process. This module is compiled as a shared library and will be linked against our actual kernel (core executable of our framework). Our exported API

calls are kept very simple and performant, the subset is small and threadsafe. On the other hand, NIO consists of a series of subsystems (cf. fig. 1). Each subsystem is almost independent of the others and serves a special purpose. Currently there are nine subsystems within our framework, a util API, a unit test framework and a series of build scripts.

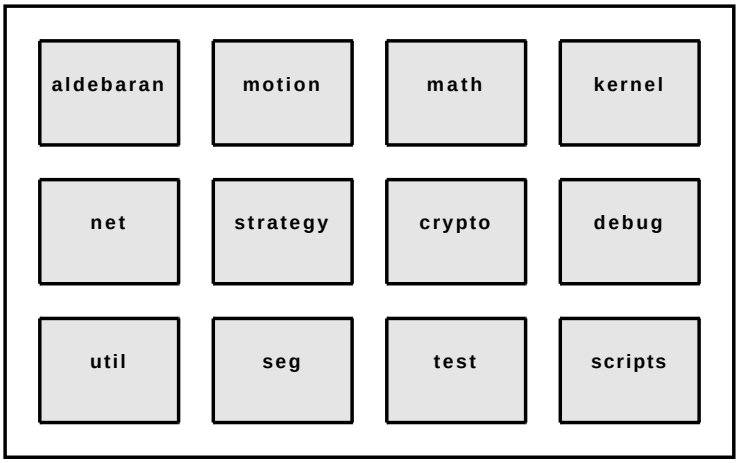

Figure 1: Basic NIO Framework components

A short description of our components:

aldebaran The aldebaran subsystem contains the Unix Domain Socket interface with the client-side and server-side implementations of our API. The API provides functionality spanning from simply reading the value of the ultrasound sensor to transmitting a video4linux frame.

motion Our motion subsystem mainly consists of a motion player, which reads motion frames from a motion file and sets the servo angles to the given value. There is also a temperature monitoring system within our motion player that detects critical thresholds and acts accordingly.

math We've implemented a highly performant math library with low-level as well as high-level math functions and future support of x86/Geode machine instruction extensions.

kernel The kernel itself represents the core executable of our framework. It bootstraps all of our functionality and makes use of all the other subsystems.

net The networking subsystem allows encrypted robot-to-robot communication with multicast and unicast protocols.

strategy Our strategy subsystem contains, as the name already tells, our game strategy. Furthermore, it is possible to swap several strategies during runtime.

crypto The crypto subsystem contains several cryptographic functions for encrypted robot-to-robot communication as well as pseudo random number generators and others.

debug The debug subsystem represents the bridge from our robot to the Java client with a userfriendly graphical interface. It is used for camera calibration, visual debugging and remote controlling the robot.

util Some util routines that didn't fit into a specific subsystem, for instance queues, linked lists, hash tables, trees and more.

seg The segmentation subsystem contains highly performant image segmentation algorithms for object recognition during gameplay.

test We have built a unit test framework for NIO that contains serveral test routines for our developed functionality.

scripts Some Perl routines for code checking and creating code metrics during the build process.

2 Motion

2.1 Evolutionary Algorithm

Most of the motions used in the tournament were closed loop motions trained using a specialized evolutionary algorithm. To achieve an acceptable speed of convergence while training on actual robots, the EA was first optimized using the Webots simulation environment. Optimizations included the utilization of symmetry and developing a mutation kernel based on B-Splines and multiple-joint mutations. Combined with a fitness function trimmed for fast as well as stable movement, we ran the EA on one of our Naos. To our surprise, the convergence characteristics actually improved when using the real robot instead of a simulation. After an evaluation of a total of just 4000 individuums of straight walking, the robot achieved a maximum speed of 32 cm/s (1.15 km/h). Other movement types like walking to the side and turning were also trained using the same approach.

To allow for bent walking, we modified the straight walking mode with optimized sinusoidal functions at the hip joints. This allowed for a turning circle adjustable down to a diameter of 1.6 meters on nearly full speed (22 °/s). The adjustments can be set at any time to any extent and therefore also allow for slalom-type walking.

Further, the straight walk was stabilized by controlling the shoulder pitch joint angles with a modified software PD controller. This was especially necessary when walking over field lines, as the robot would get stuck and fall over

because of the low vertical actuation of its feet resulting from the EA. The input into the PD controller was comprised of the current torso angles and a pre-recorded base-line of an optimal situation.

2.2 Stand-up

Both stand-up motions (front and back) were programmed by hand. Times of 5.2s for standing up from the back and 4.5s for standing up from the front were achieved, which is faster than the motions supplied with the robot. Falling down is detected using the torso angle and results in a reduced stiffness of all joints to lessen the severity of the hit.

2.3 Motion Player

A motion player is included into NIO. It can load and reload motions stored in a human-readable format on-the-fly and will play motions continuously until a function to stop or switch the motion is called. It will automatically utilize acceleration, deceleration and morphing between motions if they are supplied by the motion files. Further, it is possible to let the motion player handle stand-up routines when it detects a falling robot as well as a penalized mode that temporarily suspends all motions.

3 Vision

There are two separate cameras available in the robot. However, as switching between those cameras took too long with the version of NaoQI used for Robocup 2009, only the chin camera was used. This brought another advantage in that only one color calibration had to be done.

3.1 Camera Settings

In the ligthing conditions prevalent at Robocup2009, we mainly used the following camera parameters:

parameter	value
resolution	640x480
framerate	30fps
format	yuv 4:2:2
gain	100
red chroma	80
blue chroma	100
contrast	60
autogain	0
AutoWhiteBalance	0
AutoExposure	0
ExposureTime	80
Brightness	128
Saturation	160

3.2 Color Table

To classify objects like the ball, goals and floor, we use a 64x64x64 downsampled lookup table in YUV-space. It holds object classifications for every color, as well as a certanity of the correctness of the classifications.

We developed an interactive Java program to generate a color table within approx. 5 minutes by using its integrated color picker tools.

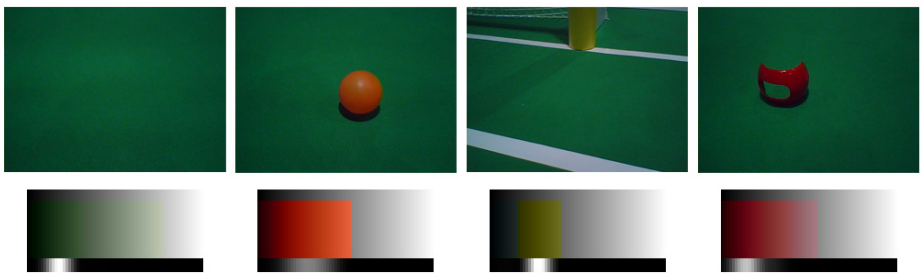

Figure 2: Extract of the color table and probability density function

3.3 Object Recognition

Two separate algorithms are used to recognize either the ball or goals and can be called from within the states of the strategy. The separation of both results in a higher framerate.

3.3.1 Ball Recognition

For segmenting the ball, a classification according to the color table is done at points determined by a raster. This raster is determined by the torso angle of the robot. Only points below the horizon are used with a distance inbetween inversely proportional to the relative distance to the robot. Regions with high probabilities of beloning to class "ball" are examined further by utilizing a circle-shaped filter. This eliminates image noise and misclassifications (e.g., orange bags).

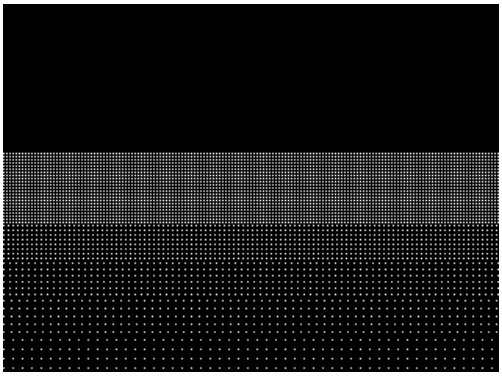

Abbildung 3: Rasterization of the image according to est. distance

3.3.2 Anyball Challenge

The anyball challenge was set to identify multiple balls of different color and size and kick them, preferably into a goal. The main problem was the identification of balls of different color. The idea used by our team was to classify everything which is not the floor or field lines as ball. A modified circle-shaped filter was used to exclude objects that were not surrounded by green floor or white lines.

3.3.3 Goal Recognition

Typically, a problem of goal recognition are people wearing blue jeans that are recognized as blue goal posts by some segmentation methods. To solve this problem, we used a simple model of the goal frame and calculated, using the RANSAC algorithm, the correct position and orientation. Similar to the ball recognition, the image needs to be scanned only partially. For this purpose, lines orthogonal to the horzion are evaluated.

4 Strategy

Our strategy is based on a finite state machine (FSM), cycling between 6 states: find ball, turn to ball, walk to ball, adjust on ball, turn around ball and shoot. Each of these states is comprised of a series of motions and segmentation settings. While, e.g., in "walk to ball", the segmentation will only recognize a ball whilst ignoring the goals or other objects, in "turn around ball", only the goals are segmented without caring about the actual position of the ball (it is assumed to be beneath it's feet and checked after a goal is found). This increases the execution speed of both states considerably which in turn results in faster reaction times and less head movement.

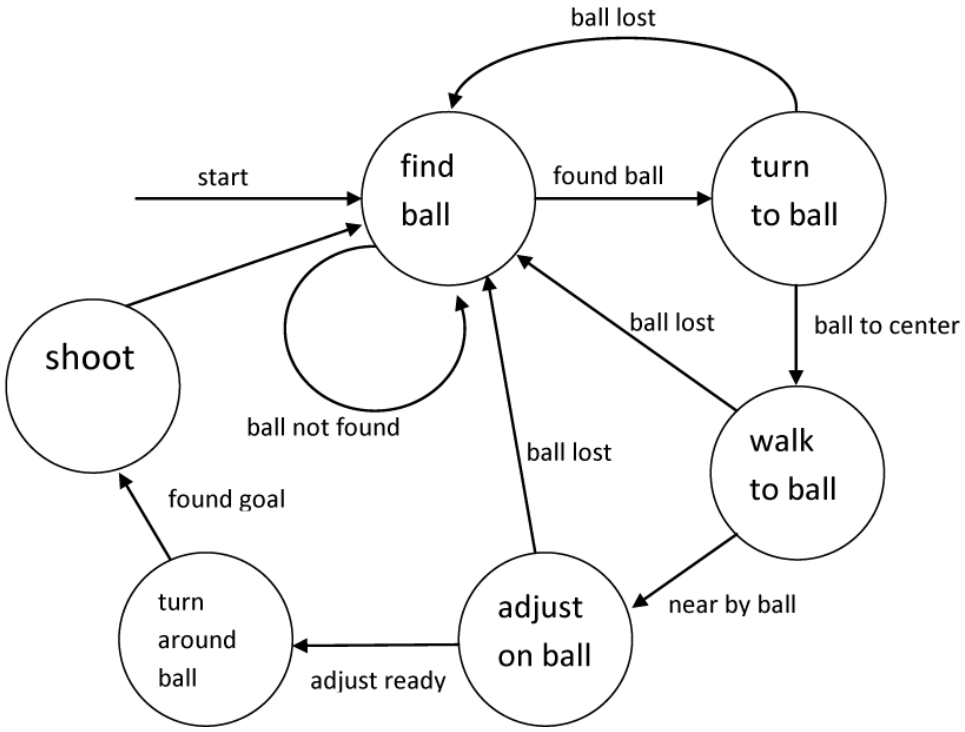

Figure 4: Finite State Machine