B-Human

Team Report and Code Release 2009

Thomas Röfer[1], Tim Laue[1], Judith Müller[1], Oliver Bösche[2], Armin Burchardt[2],
Erik Damrose[2], Katharina Gillmann[2], Colin Graf[2], Thijs Jeffry de Haas[2], Alexander Härtl[2],
Andrik Rieskamp[2], André Schreck[2], Ingo Sieverdingbeck[2], Jan-Hendrik Worch[2]

[1] Deutsches Forschungszentrum für Künstliche Intelligenz,
Enrique-Schmidt-Str. 5, 28359 Bremen, Germany
[2] Universität Bremen, Fachbereich 3, Postfach 330440, 28334 Bremen, Germany

Revision: October 23, 2009

Contents

Chapter 1

Introduction

1.1 About us

B-Human is a joint RoboCup team of the Universität Bremen and the German Research Center for Artificial Intelligence (DFKI). The team was founded in 2006 and it consists of numerous undergraduate students as well as of researchers of these two institutions. The latter have already been active in a number of RoboCup teams, such as the GermanTeam and the Bremen Byters (both Four-Legged League), B-Human in the Humanoid Kid-Size League, the BreDoBrothers (both in the Humanoid League and the Standard Platform League), and B-Smart (Small-Size League).

The senior team members have also been part of a number of other successes, such as winning the RoboCup World Championship three times with the GermanTeam (2004, 2005, and 2008), winning the RoboCup German Open also three times (2007 and 2008 with the GermanTeam, 2008 with B-Smart), and winning the Four-Legged League Technical Challenge twice (2003 and 2007 with the GermanTeam). In 2007, the team was strengthened by further computer science students, who support the team on the basis of their advanced study project.

In parallel to these activities, B-Human started as a part of the joint team BreDoBrothers, which has been a cooperation of the Technische Universität Dortmund and the Universität Bremen. The team participated in the Humanoid League in RoboCup 2006. The software was based on previous works of the GermanTeam [12]. This team was split into two single Humanoid teams, because of difficulties in developing and maintaining a robust robot platform across two locations. The DoH!Bots from Dortmund as well as B-Human from Bremen participated in RoboCup 2007; B-Human reached the quarter finals and was undefeated during round robin. In addition to the participation in the Humanoid League at the RoboCup 2008, B-Human also attended a new cooperation with the Technische Universität Dortmund. Hence, B-Human took part in the Two-Legged Competition of the Standard Platform League as part of the team BreDoBrothers, who reached the quarter finals. After the RoboCup 2008, we concentrated our work exclusively on the Two-Legged SPL. By integrating all the students of the Humanoid League team B-Human, the BreDoBrothers would have had more than thirty members. Therefore we decided to end the cooperation by mutual agreement to facilitate a better workflow and work-sharing.

In 2009, we participated in the RoboCup German Open Standard Platform League and won the competition. We scored 27 goals and received none in five matches against different teams. Furthermore, B-Human took part in the RoboCup World Championship and won the competition, achieving a goal ratio of 64:1. In addition, we could also win first place in the technical challenge, shared with Nao Team HTWK from Leipzig.

Figure 1.1: The almost complete B-Human team at the RoboCup 2009 awards ceremony.

The current team consists of the following persons:

Diploma Students. Oliver Bösche, Armin Burchardt, Erik Damrose, Katharina Gillmann, Colin Graf, Alexander Härtl, Thijs Jeffry de Haas, Judith Müller, Andrik Rieskamp, André Schreck, Andreas Seekircher, Ingo Sieverdingbeck, Thiemo Wiedemeyer, Jan-Hendrik Worch.

Researcher. Tim Laue.

Senior Researcher. Thomas Röfer (team leader).

1.2 About the Document

As we wanted to revive the tradition of the annual code release last year, it is obligatory for us to continue with it this year.

This document, which is partially based on last year's code release [19], gives a survey about the evolved system we used at RoboCup 2009. Last year we introduced a working port of our system, which was basically designed for the Humanoid League, to the Nao[1] platform. Now we completed porting, adjusted the system according to the rules of the Standard Platform League [2] and optimized it to fit with the properties of the Nao.

Chapter 2 starts with a short introduction to the software required, as well as an explanation of how to run the *Nao* with our software. Chapter 3 gives an introduction to the software framework. Chapter 4 deals with the cognition system and will give an overview of our perception and modeling components. In Chapter 5, we describe our walking approach and how to create special motion patterns. Chapter 6 gives an overview about the behavior which was used at the

[1]The robot used in the Standard Platform League, manufactured by Aldebaran Robotics.

RoboCup 2009[2] and how to create a new behavior. Chapter 7 explains our approaches that we used to compete for the Challenges. Finally, Chapter 8 describes the usage of SimRobot, the program that is both used as simulator and as debugging frontend when controlling real robots.

[2]In the CodeRelease, there is only a striker with a simple “go-to-ball-and-kick” behavior implemented which is not described in this document

Chapter 2

Getting Started

The aim of this chapter is to give an overview of the code release package, the software components used, and instructions about how to enliven a Nao with our code. For the latter, several steps are necessary: Unpacking the source code, compiling the code using Visual Studio 2008 or Linux, setting up the Nao, copying the files to the robot, and starting the software.

2.1 Unpacking

The code release package should be unpacked to a location, the path of which must *not* contain whitespaces. After the unpacking process, the chosen location should contain several subdirectories, which are described below.

Build is the target directory for temporary files created during the compilation of source code.

Config contains configuration files used to configure the Nao and the Simulator. A more thorough description of the individual files can be found below in the next section (cf. Sect. 2.2).

Doc contains some further documentation resources and is the target directory for the compiled documentation of the simulator and the behavior, which can be created by using the *BehaviorDoc* and the *SimulatorDoc* components (cf. Sect. 2.2).

Install contains all files needed to set up the flash drive of a Nao, two scripts to manage the wireless configuration of the robots, and a script to update color tables.

Make contains the Visual Studio project files, makefiles, other files needed to compile the code and the *Copyfiles* tool.

Src contains all the source code of the code release.

Util contains additional libraries and tools such as Doxygen [24].

2.2 Components and Configurations

The B-Human software is usable on Windows as well as on Linux and consists of a shared library for *NaoQi* running on the real robot, an additional executable for the robot, the same software running in our simulator SimRobot (without *NaoQi*), as well as some libraries and tools. Therefore, the software is separated into the following components:

Copyfiles is a tool for copying compiled code to the robot. For a more detailed explanation see Sect. 2.7.

VcProjGeneration is a tool (for Windows only) for updating project files based on available source files found in the *Src* directory. On Linux, all makefiles will be updated automatically on each call to *make*.

BehaviorDoc is a tool for creating the documentation of the behavior. The results will be located in *Doc/Reference/BH2009BehaviorControl*.

SimulatorDoc is a tool for creating the documentation of the complete simulator source code. The results will be located in *Doc/Reference/Simulator*.

SimRobotGUI is a library that contains the graphical user interface of SimRobot. This GUI is written in Qt4 and it is available in the configurations *Release* and *Debug*.

Controller is a library that contains Nao-specific extensions of the Simulator, the interface to the robot code framework, and it is also required for controlling and high level debugging of code that runs on a Nao. The library is also available in the configurations *Release* and *Debug*.

SimRobotCore is a library that contains the simulation engine of SimRobot. It is compilable with or without debug symbols (configurations *Release* and *Debug*).

libbhuman compiles the shared library used by the B-Human executable to interact with *NaoQi*.

URC stands for *Universal Resource Compiler* and is a small tool for automatic generation of some *.xabsl* files (cf. Sect. 6.1) and for compiling *special actions* (cf. Sect. 5.2.4).

Nao compiles the B-Human executable for the Nao. It is available in *Release*, *Optimized*, and *Debug* configurations, where *Release* produces "game code" without any support for debugging. The configuration *Optimized* produces optimized code, but still supports all debugging techniques described in Sect. 3.6.

Simulator is the executable simulator (cf. Chapter 8) for running and controlling the B-Human robot code. The robot code links against the components *SimRobotCore*, *SimRobotGUI*, *Controller* and some third-party libraries. It is compilable in *Optimized*, *Debug With Release Libs*, and *Debug* configurations. All these configurations contain debug code but *Optimized* performs some optimizations and strips debug symbols (Linux). *Debug With Release Libs* produces debuggable robot code while linking against non-debuggable *Release* libraries.

Behavior compiles the behavior specified in *.xabsl* files into an internal format (cf. Sect. 6.1).

SpecialActions compiles motion patterns (*.mof* files) into an internal format (cf. Sect. 5.2.4).

2.3 Compiling using Visual Studio 2008

2.3.1 Required Software

- Visual Studio 2008 SP1

- cygwin – 1.5 with the following additional packages: make, ruby, rsync, openssh, libxml2, libxslt. Add the ...\cygwin\bin directory to the PATH environment variable.
- gcc, glibc – Linux cross compiler for cygwin, download from `http://www.b-human.de/download.php?file=coderelease09_crosscompiler`, in order to keep symbolic links use a cygwin shell to extract. This cross compiler package is based on the cross compiler downloadable from `http://sourceforge.net/project/showfiles.php?group_id=135860`. To use this cross compiler together with our software, we placed the needed boost and python include files into the include directory.
- alcommon – copy the contents of *extern/c/aldebaran/alcommon* from the *Nao SDK release v1.3.13 linux* (*NaoQiRoboCup-1.3.13-Linux.tar.gz*) to the directory *Util/alcommon* of the unpacked B-Human software. The package is available at the internal RoboCup download area of Aldebaran Robotics. Please note that this package is only required to compile the code for the actual Nao robot.

2.3.2 Compiling

Open the Visual Studio 2008 solution file *Make/BHuman.sln*. It contains all projects needed to compile the source code. Select the desired configuration (cf. Sect. 2.2) out of the drop-down menu in Visual Studio 2008 and select *Build/Build Solution* to build everything including the documentation. Otherwise click on the project to be built (usually *_Simulator* or *_Nao*) and choose *Build/Build Project* in the menu bar. Select *_Simulator* as start project.

2.4 Compiling on Linux

2.4.1 Required Software

Additional requirements (listed by common package names) for an x86-based Linux distribution (e. g. Ubuntu Hardy):

- g++, make
- libqt4-dev – 4.3 or above (qt.nokia.com)
- ruby
- doxygen – For compiling the documentation.
- graphviz – For compiling the behavior documentation and for using the module view of the simulator. (www.graphviz.org)
- xsltproc – For compiling the behavior documentation.
- openssh-client – For deploying compiled code to the Nao.
- rsync – For deploying compiled code to the Nao.
- alcommon – copy the contents of *extern/c/aldebaran/alcommon* from the *Nao SDK release v1.3.13 linux* (*NaoQiRoboCup-1.3.13-Linux.tar.gz*) to the directory *Util/alcommon* of the unpacked B-Human software. The package is available at the internal RoboCup download area of Aldebaran Robotics. Please note that this package is only required to compile the code for the actual Nao robot.

2.4.2 Compiling

To compile one of the components described in section 2.2 (except *Copyfiles* and *VcProjGeneration*), simply select *Make* as the current working directory and type:

```
make <component> CONFIG=<configuration>
```

The Makefile in the *Make* directory controls all calls to generated sub-Makefiles for each component. They have the name *<component>.make* and are also located in the *Make* directory. Dependencies between the components are handled by the major Makefile. It is possible to compile or cleanup a single component without dependencies by using:

```
make -f <component>.make [CONFIG=<configuration>] [clean]
```

To clean up the whole solution use:

```
make clean
```

2.5 Configuration Files

In this section the files and subdirectories in the directory *Config* are explained in greater detail.

ballPerceptor.cfg contains parameters for the BallPerceptor (cf. Sect. 4.1.3.5).

bodyContour.cfg contains parameters for the BodyContourProvider (cf. Sect. 4.1.2).

goalPerceptor.cfg contains parameters for the GoalPerceptor (cf. Sect. 4.1.3.4).

jointHardness.cfg contains the default hardness for every joint of the *Nao*.

linePerceptor.cfg contains parameters for the LinePerceptor (cf. Sect. 4.1.3.3).

odometry.cfg provides information for the self-locator while executing special actions. See the file or section 5.2.4 for more explanations.

regionAnalyzer.cfg contains parameters for the RegionAnalyzer (cf. Sect. 4.1.3.2).

regionizer.cfg contains parameters for the Regionizer (cf. Sect. 4.1.3.1).

settings.cfg contains parameters to control the Nao. The entry *model* is obsolete and should be *nao*. The *teamNumber* is required to determine which information sent by the GameController is addressed to the own team. The *teamPort* is the UDP port used for team communication (cf. Sect. 3.5.4). The *teamColor* determines the color of the own team (blue or red). The *playerNumber* must be different for each robot of the team. It is used to identify a robot by the GameController and in the team communication. In addition, it can be used in behavior control. *location* determines which directory in the *Location* subdirectory is used to retrieve the location-dependent settings.

walking.cfg contains parameters for the WalkingEngine (cf. Sect. 5.2.3.2).

Images is used to save exported images from the simulator.

Keys contains the SSH keys needed by the tool *Copyfiles* to connect remotely to the Nao.

Locations contains one directory for each location. These directories control settings that depend on the environment, i. e. the lighting or the field layout. It can be switched quickly between different locations by setting the according value in the *settings.cfg*. Thus different field definitions, color tables, and behaviors can be prepared and loaded. The location *Default* can be used as normal location but has a special status. Whenever a file needed is not found in the current location, the corresponding file from the location *Default* is used.

Locations/<location>/behavior.cfg determines which agent behavior will be loaded when running the code.

Locations/<location>/camera.cfg contains parameters to control the camera.

Locations/<location>/coltable.c64 is the color table that is used for this location. There can also be a unique color table for each robot, in which case this color table is ignored.

Locations/<location>/field.cfg contains the field sizes and coordinates of features on the field.

Locations/<location>/fieldModel.tab is a binary file which can be generated by the SelfLocator. It contains look-up tables for mapping perceptions to lines and line crossings. It is used by the self-locator (cf. Sect. 4.2.1).

Locations/<location>/goalNet.tab is a binary file which can be generated by the SelfLocator. It contains a look-up table for distinguishing valid line perceptions from the goal net.

Locations/<location>/modules.cfg contains information about which representations are available and which module provides them while the code is running. Representations that are exchanged between the two main processes are given in the section *Shared*.

Locations/<location>/selfloc.cfg contains parameters for the module SelfLocator (cf. Sect. 4.2.1).

Logs contains logfiles that can be recorded and replayed using the simulator.

Robots contains one directory for each robot and the settings of the robot. The configuration files found here are used for individual calibration settings for each robot. The directory *Nao* is used by the simulator. For each robot, a subdirectory with the name of the robot must exist.

Robots/<robotName>/cameraCalibration.cfg contains correction values for camera- and body-roll and body-tilt and body translation (cf. Sect. 4.1.1.1).

Robots/<robotName>/jointCalibration.cfg contains calibration values for each joint. In this file offset, sign, minimal and maximal joint angles can be set individually. The calibration is also used to map between B-Human's joint angles and NaoQi's joint angles.

Robots/<robotName>/masses.cfg contains the masses of all robot limbs used to compute the center of mass (cf. Sect. 5.1.3).

Robots/<robotName>/robotDimensions.cfg contains values that are used by forward and inverse kinematics (cf. Sect. 5.2.3.4).

Robots/<robotName>/sensorCalibration.cfg contains calibration settings for the sensors of the robot.

Robots/<robotName>/walking.cfg . This file is optional. It contains the walking parameters for the robot. If this file exists, it is used instead of the general file in the *Config* directory.

Scenes contains different scenes for the simulator.

Sounds contains the sound files that are played by the robot and the simulator.

2.6 Setting up the Nao

2.6.1 Requirements

Setting up the Nao is only possible from a Linux OS. First of all, download the *OS image v1.3.13* that is available at the internal RoboCup download area. Unrar this file and move the extracted image to *Install/images*. To save space it is possible to compress the image with bzip2. After that, there should be a file *opennao-robocup-1.3.13-nao-geode.ext3* or *opennao-robocup-1.3.13-nao-geode.ext.bz2*. If your image file has a different name, you have the choice to change the *imageName* variable in line 7 of *flashAndInstall.sh* or to use the *-i* option with the name of your image file as argument when calling the install script. Note that the only supported compression is bzip2, which is only detected if the image file has the bz2 extension. All other file extensions are ignored and the image file is considered as uncompressed image file.

The only supported NaoQi and os image version is 1.3.13.

To use the scripts in the directory *Install* the following tools are needed:

sed, tune2fs, sfdisk, mount, umount, grep, awk, patch, bunzip2, tar, mktemp, whoami, mkfs.vfat, dd, and tr or bash in version 4.x

Each script will check its own requirements and will terminate with an error message if a tool needed is not found.

2.6.2 Creating Robot Configuration

Before you start setting up the Nao, you need to create configuration files for each robot you want to set up. To create a robot configuration run *createNewRobot.sh*. The script expects a team id, a robot id, and a robot name. The team id is usually equal to your team number configured in *Config/settings.cfg* but you can use any number between 1 and 254. The given team id is used as third part of the IP version 4 address of the robot on both interfaces. All robots playing in the same team need the same team id to be able to communicate with each other. The robot id is the last part of the IP address and must be unique for each team id. The robot name is used as hostname in the Nao operating system and is saved in the chestboard of the Nao as *BodyNickname*.

Here is an example for creating a new set of configuration files:

createNewRobot.sh -t 3 -r 25 Penny

Before creating your first robot configuration, check that the network configuration template file *_interfaces_template_* matches the requirements of your local network configuration.

Help for *createNewRobot.sh* is available using the option *-h*.

Running *createNewRobots.sh* creates all needed files to flash the robot. This script also creates a robot directory in *Config/Robots* as a copy of the template directory.

2.6.3 Wireless configuration

To use the wireless interface of the Nao, you need to create a *wpa_supplicant.conf* file. The easiest way is to copy the file *Install/files/wpa_supplicant.conf_template* and change it in the way needed for your wireless network. The name of the new configuration must be *wpa_supplicant_<suffix>* in order to use this file with our install script. To use this new configuration as default for all installs, change the variable *wlanConfigSuffix* in line 15 of *Install/flashAndInstall.sh* to the chosen suffix or use the option *-W* of the install script with the suffix as argument. To manage the wireless configurations of your robots, for example to copy new or updated configurations or switching the active configuration, you can use the scripts *updateWirelessConfig.sh* and *switchActiveWirelessConfiguration.sh*. *updateWirelessConfig.sh* copies all found *wpa_supplicant.conf* files to all known and reachable robots. For *switchActiveWirelessConfiguration.sh*, help is available using the option *-h*. This script activates the configuration specified by the argument on all known and reachable robots. While switching the active wireless configuration you should be connected via a cable connection. Switching the wireless configuration via a wireless connection is untested.

A robot is known if an interfaces file exists for that robot. Both scripts use all ip addresses found in all interfaces files except the *_interfaces_template_* file to connect to the robots. If you are connected to a robot via a cable and a wireless connection, all changes are done twice.

2.6.4 Setup

Open the head of the Nao and remove the USB flash memory. Plug the USB flash drive into the computer. Run *flashAndInstall.sh* as root with at least the name of the robot as argument. With *-h* as option *flashAndInstall.sh* prints a short help. The install script uses different mechanisms to detect the flash drive. If it fails or if the script detects more than one appropriate flash drive, you have to call the script using the option *-d* with the device name as argument.

With *-d* or with device auto detection some safety checks are enabled to prevent possible data loss on your harddrive. To disable these safety checks you can use *-D*. With *-D flashAndInstall.sh* will accept any given device name and will write to this device. Use *-D* only if it is necessary and you are sure what you are doing.

The install script writes the os image to the flash drive and formats the *userdata* partition of the drive. After that, the needed configuration files for the robot and all files needed to run *copyfiles.sh* for this robot and to use it with the B-Human software are copied to the drive. If the script terminates without any error message you can remove the flash drive from your computer and plug it into the Nao.

Start your Nao and wait until the boot finished. Connect via SSH to the Nao and log in as root with the password *cr2009*. Start *./phase2.sh* and reboot the Nao. After the reboot the Nao is ready to be used with the B-Human software.

2.7 Copying the Compiled Code

The tool *Copyfiles* is used to copy compiled code to the *Nao*.

Running Windows you have two possibilities to use *Copyfiles*. On the one hand, in Visual Studio you can do that by "building" the tool *Copyfiles*. Copyfiles can be built in the known configurations (*Debug*, *DebugWithReleaseLibs*, *Optimized*, or *Release*). It will check if the code built is up-to-date. If the code is not up-to-date in the desired configuration, it will be built. Otherwise you will be prompted for entering parameters. On the other hand you can just execute the file *copyfiles.cmd* at the command prompt.

Running Linux you have to execute the file *copyfiles.sh*, which is located in the *Make* directory.

Copyfiles requires two mandatory parameters. First, the configuration the code was compiled with (*Debug*, *Optimized*, or *Release*), and second, the IP address of the robot. To adjust the desired settings, it is possible to set the following optional parameters:

Option	Description
-l <location>	sets the location
-t <color>	sets the team color to blue or red
-p <number>	sets the player number
-d	deletes the local cache and the target directory
-m n <ip>	copy to <ip> and set player number to n

Possible calls could be:

```
copyfiles Optimized 10.0.1.103 -t red -p 2
copyfiles Release -d -m 1 10.0.1.101 -m 3 10.0.1.102
```

The destination directory on the robot is */media/userdata/Config*.

2.8 Working with the Nao

After pressing the chest button, it takes about 55 seconds until NaoQi is started. Currently the B-Human software consists of a shared library (*libbhuman.so*) that is loaded by *NaoQi* at startup, and an executable (*bhuman*) also loaded at startup.

/home/root contains scripts to start and stop *NaoQi* via SSH:

./stop stops running instances of *NaoQi* and *bhuman*.

./naoqi executes *NaoQi* in the foreground. Press *Ctrl+C* to terminate the process. Please note that the process will automatically be terminated if the SSH connection is closed.

./naoqid start|stop|restart starts, stops or restarts *NaoQi*. After updating *libbhuman* with *Copyfiles NaoQi* needs a restart.

./bhuman executes the *bhuman* executable in foreground. Press *Ctrl+C* to terminate the process. Please note that the process will automatically be terminated if the SSH connection is closed.

./bhumand start|stop|restart starts, stops or restarts the *bhuman* executable. After uploading files with *Copyfiles bhuman* must be restarted.

./status shows the status of *NaoQi* and *bhuman*.

The Nao can be shut down in two different ways:

shutdown -h now will shut down the Nao, but it can be booted again by pressing the chest button because the chestboard is still energized. If the B-Human software is running, this can also be done by pressing the chest button longer than three seconds.

./naoqid stop && harakiri --deep && shutdown -h now will shut down the Nao. If the Nao runs on battery it will be completely switched off after a couple of seconds. In this case an external power supply may be needed to start the Nao again.

2.9 Starting SimRobot

On Windows, the simulator can either be started from Microsoft Visual Studio, or by starting a scene description file in *Config/Scenes*[1]. In the first case, a scene description file has to be opened manually, whereas it will already be loaded in the latter case. On Linux, just run *Build/Simulator/Linux/<configuration>/Simulator*, and load a scene description file afterwards. When a simulation is opened for the first time, a scene graph, console and editor window appear. All of them can be docked into the main window. Immediately after starting the simulation using the *Simulation/Start* entry in the menu, the scene graph will appear in the scene graph window. A scene view showing the soccer field can be opened by double-clicking *scene RoboCup*. The view can be adjusted by using the context menu of the window or the toolbar.

After starting a simulation, a script file may automatically be executed, setting up the robot(s) as desired. The name of the script file is the same as the name of the scene description file but with the extension *.con*. Together with the ability of SimRobot to store the window layout, the software can be configured to always start with a setup suitable for a certain task.

Although any object in the scene graph can be opened, only displaying certain entries in the object tree makes sense, namely the *scene*, the objects in the group *robots*, and all *information views*.

To connect to a real Nao, enter its IP address in the file *Config/Scenes/connect.con* on the PC. Afterwards, start the simulation scene *Config/Scenes/RemoteRobot.ros* (cf. Sect. 8.6.3). A remote connection to the Nao is only possible if the code running on the Nao was compiled in either the *Debug* or the *Optimized* configuration.

For more detailed information about SimRobot see Chapter 8.

[1]This will only work if the simulator was started at least once before.

Chapter 3

Architecture

The B-Human architecture is based on the framework of the GermanTeam 2007 [21], adapted to the Nao. This chapter summarizes the major features of the architecture: binding, processes, modules and representations, communication, and debugging support.

3.1 Binding

The only appropriate way to access the actuators and sensors (except the camera) of the Nao is to use the *NaoQi* SDK that is actually a stand-alone module framework that we do not use as such. Therefore, we deactivated all non essential pre-assembled modules and implemented the very basic module *libbhuman* for accessing the actuators and sensors from another native platform process called *bhuman* that encapsulates the B-Human module framework.

Whenever the Device Communication Manager (DCM) reads a new set of sensor values, it notifies the *libbhuman* about this event using an `atPostProcess` callback function. After this notification, *libbhuman* writes the newly read sensor values into a shared memory block and raises a semaphore to provide a synchronization mechanism to the other process. The *bhuman* process waits for the semaphore, reads the sensor values that were written to the shared memory block, calls all registered modules within B-Human's process *Motion* and writes the resulting actuator values back into the shared memory block right after all modules have been called. When the DCM is about to transmit desired actuator values (e. g. target joint angles) to the hardware, it calls the `atPreProcess` callback function. On this event *libbhuman* sends the desired actuator values from the shared memory block to the DCM.

It would also be possible to encapsulate the B-Human framework as a whole within a single *NaoQi* module, but this would lead to a solution with a lot of drawbacks. The advantages of the separated solution are:

- Both frameworks use their own address space without losing their real-time capabilities and without a noticeable reduction of performance. Thus, a malfunction of the process *bhuman* cannot affect *NaoQi* and vice versa.

- Whenever *bhuman* crashes, *libbhuman* is still able to display this malfunction using red blinking eye LEDs and to make the Nao sit down slowly. Therefore, the *bhuman* process uses its own watchdog that can be activated using the -w flag[1] when starting the *bhuman* process. When this flag is set, the process forks itself at the beginning where one instance

[1]The start up scripts *bhuman* and *bhumand* set this flag by default.

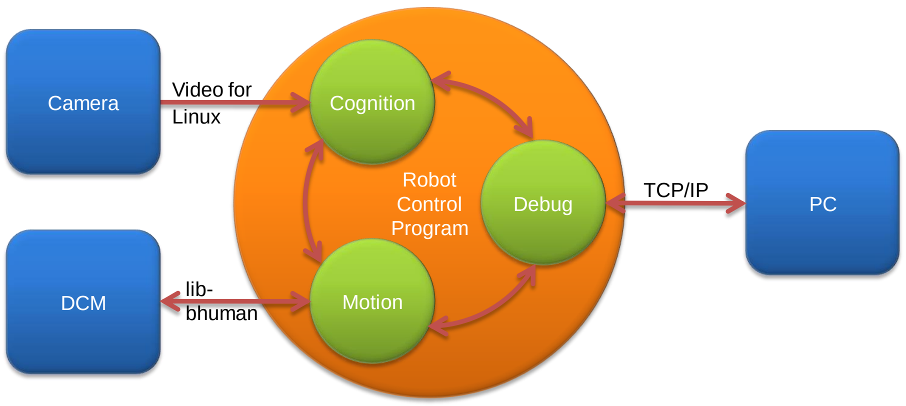


Figure 3.1: The processes used on the Nao

waits for a regular or irregular exit of the other. On an irregular exit the exit code can be written into the shared memory block. The *libbhuman* monitors whether sensor values were handled by the *bhuman* process using the counter of the semaphore. When this counter exceeds a predefined value the error handling code will be initiated. When using release code (cf. Sect. 2.2), the watchdog automatically restarts the *bhuman* process after an irregular exit.

- The process *bhuman* can be started or restarted within only a few seconds. The start up of *NaoQi* takes about 15 seconds, but because of the separated solution restarting of *NaoQi* is not necessary as long as the *libbhuman* was not changed.

- Debugging with a tool such as the GDB is much simpler since the *bhuman* executable can be started within debugger without taking care of NaoQi.

3.2 Processes

Most robot control programs use concurrent processes. The number of parallel processes is best dictated by external requirements coming from the robot itself or its operating system. The Nao provides images at a frequency of 15 Hz or 30 Hz and accepts new joint angles at 50 Hz. Therefore, it makes sense to have two processes running at these frequencies. In addition, the TCP communication with a host PC (for the purpose of debugging) may block while sending data, so it also has to reside in its own process. This results in the three processes *Cognition*, *Motion*, and *Debug* used in the B-Human system (cf. Fig. 3.1). *Cognition* receives camera images from *Video for Linux*, as well as sensor data from the process *Motion*. It processes this data and sends high-level motion commands back to the process *Motion*. This process actually executes these commands by generating the target angles for the 21 joints of the Nao. It sends these target angles to Nao's *Device Communication Manager*, and it receives sensor readings such as the actual joint angles, acceleration and gyro measurements, etc. In addition, *Motion* reports about the motion of the robot, e. g., by providing the results of dead reckoning. The process *Debug* communicates with the host PC. It distributes the data received from it to the other two processes, and it collects the data provided by them and forwards it back to the host machine. It is inactive during actual games.

Processes in the sense of the architecture described can be implemented as actual operating system processes, or as threads. On the Nao and in the simulator, threads are used. In contrast, in B-Human's team in the Humanoid League, framework processes were mapped to actual processes of the operating system (i. e. Windows CE).

3.3 Modules and Representations

A robot control program usually consists of several modules each of which performs a certain task, e. g. image processing, self-localization, or walking. Modules require a certain input and produce a certain output (i. e. so-called *representations*). Therefore, they have to be executed in a specific order to make the whole system work. The module framework introduced in [21] simplifies the definition of the interfaces of modules, and automatically determines the sequence in which the modules are executed. It consists of the *blackboard*, the *module definition*, and a visualization component (cf. Sect. 8.3.9).

3.3.1 Blackboard

The blackboard [9] is the central storage for information, i. e. for the representations. Each process has its own blackboard. Representations are transmitted through inter-process communication if a module in one process requires a representation that is provided by a module in another process. The blackboard itself only contains references to representations, not the representations themselves:

```
class BallPercept;
class FrameInfo;
// ...
class Blackboard
{
protected:
  const BallPercept& theBallPercept;
  const FrameInfo& theFrameInfo;
// ...
};
```

Thereby, it is possible that only those representations are constructed, that are actually used by the current selection of modules in a certain process. For instance, the process *Motion* does not process camera images. Therefore, it does not require to instantiate an image object (approximately 300 KB in size).

3.3.2 Module Definition

The definition of a module consists of three parts: the module interface, its actual implementation, and a statement that allows to instantiate the module. Here an example:

```
MODULE(SimpleBallLocator)
  REQUIRES(BallPercept)
  REQUIRES(FrameInfo)
  PROVIDES(BallModel)
END_MODULE

class SimpleBallLocator : public SimpleBallLocatorBase
{
  void update(BallModel& ballModel)
  {
    if(theBallPercept.wasSeen)
    {
      ballModel.position = theBallPercept.position;
      ballModel.wasLastSeen = theFrameInfo.frameTime;
    }
  }
}

MAKE_MODULE(SimpleBallLocator, World Modeling)
```

The module interface defines the name of the module (e. g. `MODULE(SimpleBallLocator)`), the representations that are required to perform its task, and the representations provided by the module. The interface basically creates a base class for the actual module following the naming scheme `<ModuleName>Base`. The actual implementation of the module is a class that is derived from that base class. It has read-only access to all the required representations in the blackboard (and only to those), and it must define an `update` method for each representation that is provided. As will be described in Section 3.3.3, modules can expect that all their required representations have been updated before any of their provider methods is called. Finally, the `MAKE_MODULE` statement allows the module to be instantiated. It has a second parameter that defines a category that is used for a more structured visualization of the module configuration (cf. Sect. 8.3.9).

The module definition actually provides a lot of hidden functionality. Each `PROVIDES` statement makes sure that the representation provided can be constructed and deconstructed (remember, the blackboard only contains references), and will be available before it is first used. In addition, representations provided can be sent to other processes, and representations required can be received from other processes. The information that a module has certain requirements and provides certain representations is not only used to generate a base class for that module, but is also available for sorting the providers, and can be requested by a host PC. There it can be used to change the configuration, for visualization (cf. Sect. 8.3.9), and to determine which representations have to be transferred from one process to the other. Please note that the latter information cannot be derived by the processes themselves, because they only know about their own modules, not about the modules defined in other processes. Last but not least, the execution time of each module can be determined (cf. Sect. 3.6.6) and the representations provided can be sent to a host PC or even altered by it.

The latter functionality is achieved by variants of the macro `PROVIDES` that add support for `MODIFY` (cf. Sect. 3.6.5), support for streaming the representation to be recorded in a log file (`OUTPUT`, requires a *message id* with the same name as the representation, cf. Sect. 3.5), and drawing based on a parameterless method *draw* implemented by the representation itself. The maximum version of the macro is `PROVIDES_WITH_MODIFY_AND_OUTPUT_AND_DRAW`. For a reduced functionality, the sections of the name that are not required or not supported can be left out.

Besides the macro `REQUIRES`, there also is the macro `USES(<representation>)`. `USES` simply gives access to a certain representation, without defining any dependencies. Thereby, a module can access a representation that will be updated later, accessing its state from the previous frame. Hence, `USES` can be used to model cyclic relations. The module view (cf. Sect. 8.3.9) does not display `USES` connections.

3.3.3 Configuring Providers

Since modules can provide more than a single representation, the configuration has to be performed on the level of providers. For each representation it can be selected which module will provide it or that it will not be provided at all. In addition it has to be specified which representations have to be shared between the processes, i. e. which representations will be sent from one process to the other. The latter can be derived automatically from the providers selected in each process, but only on a host PC that has the information about all processes. Normally the configuration is read from the file *Config/Location/<location>/modules.cfg* during the boot-time of the robot, but it can also be changed interactively when the robot has a debugging connecting to a host PC.

The configuration does not specify the sequence in which the providers are executed. This sequence is automatically determined at runtime based on the rule that all representations required by a provider must already have been provided by other providers before, i. e. those providers have to be executed earlier.

In some situations it is required that a certain representation is provided by a module before any other representation is provided by the same module, e. g., when the main task of the module is performed in the `update` method of that representation, and the other `update` methods rely on results computed in the first one. Such a case can be implemented by both requiring and providing a representation in the same module.

3.3.4 Pseudo-Module *default*

During the development of the robot control software it is sometimes desirable to simply deactivate a certain provider or module. As mentioned above, it can always be decided not to provide a certain representation, i. e. all providers generating the representation are switched off. However, not providing a certain representation typically makes the set of providers inconsistent, because other providers rely on that representation, so they would have to be deactivated as well. This has a cascading effect. In many situations it would be better to be able to deactivate a provider without any effect on the dependencies between the modules. That is what the module *default* was designed for. It is an artificial construct – so not a real module – that can provide all representations that can be provided by any module in the same process. It will never change any of the representations – so they basically remain in their initial state – but it will make sure that they exist, and thereby, all dependencies can be resolved. However, in terms of functionality a configuration using *default* is never complete and should not be used during actual games.

3.4 Streams

In most applications, it is necessary that data can be serialized, i. e. transformed into a sequence of bytes. While this is straightforward for data structures that already consist of a single block of memory, it is a more complex task for dynamic structures, as e. g. lists, trees, or graphs. The implementation presented in this document follows the ideas introduced by the C++ iostreams library, i. e., the operators << and >> are used to implement the process of serialization. It is also possible to derive classes from class *Streamable* and implement the mandatory method *serialize(In*, Out*)*. In addition, the basic concept of streaming data was extended by a mechanism to gather information on the structure of the data while serializing it.

There are reasons not to use the C++ iostreams library. The C++ iostreams library does not guarantee that the data is streamed in a way that it can be read back without any special handling, especially when streaming into and from text files. Another reason not to use the C++ iostreams library is that the structure of the streamed data is only explicitly known in the streaming operators themselves. Hence, exactly those operators have to be used on both sides of a communication, which results in problems regarding different program versions or even the use of different programming languages.

Therefore, the *Streams* library was implemented. As a convention, all classes that write data into a stream have a name starting with “Out”, while classes that read data from a stream start with “In”. In fact, all writing classes are derived from class *Out*, and all reading classes are derivations of class *In*.

All streaming classes derived from *In* and *Out* are composed of two components: One for reading/writing the data from/to a physical medium and one for formatting the data from/to a specific format. Classes writing to physical media derive from *PhysicalOutStream*, classes for reading derive from *PhysicalInStream*. Classes for formatted writing of data derive from *StreamWriter*, classes for reading derive from *StreamReader*. The composition is done by the *OutStream* and *InStream* class templates.

3.4.1 Streams Available

Currently, the following classes are implemented:

PhysicalOutStream. Abstract class

- **OutFile.** Writing into files
- **OutMemory.** Writing into memory
- **OutSize.** Determine memory size for storage
- **OutMessageQueue.** Writing into a MessageQueue

StreamWriter. Abstract class

- **OutBinary.** Formats data binary
- **OutText.** Formats data as text
- **OutTextRaw.** Formats data as raw text (same output as “cout”)

Out. Abstract class

- **OutStream<PhysicalOutStream,StreamWriter>.** Abstract template class
  - **OutBinaryFile.** Writing into binary files
  - **OutTextFile.** Writing into text files
  - **OutTextRawFile.** Writing into raw text files
  - **OutBinaryMemory.** Writing binary into memory
  - **OutTextMemory.** Writing into memory as text
  - **OutTextRawMemory.** Writing into memory as raw text
  - **OutBinarySize.** Determine memory size for binary storage
  - **OutTextSize.** Determine memory size for text storage
  - **OutTextRawSize.** Determine memory size for raw text storage
  - **OutBinaryMessage.** Writing binary into a MessageQueue
  - **OutTextMessage.** Writing into a MessageQueue as text

OutTextRawMessage. Writing into a MessageQueue as raw text

PhysicalInStream. Abstract class

InFile. Reading from files

InMemory. Reading from memory

InMessageQueue. Reading from a MessageQueue

StreamReader. Abstract class

InBinary. Binary reading

InText. Reading data as text

InConfig. Reading configuration file data from streams

In. Abstract class

InStream<PhysicalInStream,StreamReader>. Abstract class template

InBinaryFile. Reading from binary files

InTextFile. Reading from text files

InConfigFile. Reading from configuration files

InBinaryMemory. Reading binary data from memory

InTextMemory. Reading text data from memory

InConfigMemory. Reading config-file-style text data from memory

InBinaryMessage. Reading binary data from a MessageQueue

InTextMessage. Reading text data from a MessageQueue

InConfigMessage. Reading config-file-style text data from a MessageQueue

3.4.2 Streaming Data

To write data into a stream, *Tools/Streams/OutStreams.h* must be included, a stream must be constructed, and the data must be written into the stream. For example, to write data into a text file, the following code would be appropriate:

```
#include "Tools/Streams/OutStreams.h"
// ...
OutTextFile stream("MyFile.txt");
stream << 1 << 3.14 << "Hello Dolly" << endl << 42;
```

The file will be written into the configuration directory, e. g. *Config/MyFile.txt* on the PC. It will look like this:

```
 1 3.14000 "Hello Dolly"
 42
```

As spaces are used to separate entries in text files, the string “Hello Dolly” is enclosed in double quotes. The data can be read back using the following code:

```
#include "Tools/Streams/InStreams.h"
// ...
InTextFile stream("MyFile.txt");
```

```
int a,d;
double b;
std::string c;
stream >> a >> b >> c >> d;
```

It is not necessary to read the symbol *endl* here, although it would also work, i. e. it would be ignored.

For writing to text streams without the separation of entries and the addition of double quotes, *OutTextRawFile* can be used instead of *OutTextFile*. It formats the data such as known from the ANSI C++ *cout* stream. The example above is formatted as following:

```
13.14000Hello Dolly
42
```

To make streaming independent of the kind of the stream used, it could be encapsulated in functions. In this case, only the abstract base classes *In* and *Out* should be used to pass streams as parameters, because this generates the independence from the type of the streams:

```
#include "Tools/Streams/InOut.h"

void write(Out& stream)
{
  stream << 1 << 3.14 << "Hello Dolly" << endl << 42;
}

void read(In& stream)
{
  int a,d;
  double b;
  std::string c;
  stream >> a >> b >> c >> d;
}
// ...
OutTextFile stream("MyFile.txt");
write(stream);
// ...
InTextFile stream("MyFile.txt");
read(stream);
```

3.4.3 Making Classes Streamable

A class is made streamable by deriving it from the class *Streamable* and implementing the abstract method *serialize(In*, Out*)*. For data types derived from *Streamable* streaming operators are provided, meaning they may be used as any other data type with standard streaming operators implemented. To realize the *modify* functionality (cf. Sect. 3.6.5), the streaming method uses macros to acquire structural information about the data streamed. This includes the data types of the data streamed as well as that names of attributes. The process of acquiring names and types of members of data types is automated. The following macros can be used to specify the data to stream in the method `serialize`:

STREAM_REGISTER_BEGIN() indicates the start of a streaming operation.

STREAM_BASE(<class>) streams the base class.

STREAM(<attribute>) streams an attribute, retrieving its name in the process.

STREAM_ENUM(<attribute>, <numberOfEnumElements>, <getNameFunctionPtr>) streams an attribute of an enumeration type, retrieving its name in the process, as well as the names of all possible values.

STREAM_ARRAY(<attribute>) streams an array of constant size.

STREAM_ENUM_ARRAY(<attribute>, <numberOfEnumElements>, <getNameFunctionPtr>) streams an array of constant size. The elements of the array have an enumeration type. The macro retrieves the name of the array, as well as the names of all possible values of its elements.

STREAM_DYN_ARRAY(<attribute>, <numberOfElements>) streams a dynamic array with a certain number of elements. Note that the number of elements will be overridden when the array is read.

STREAM_VECTOR(<attribute>) streams an instance of `std::vector`.

STREAM_REGISTER_FINISH() indicates the end of the streaming operation for this data type.

These macros are intended to be used in the `serialize` method. For instance, to stream an attribute `test` and a vector called `testVector`:

```
virtual void serialize(In* in, Out* out)
{
  STREAM_REGISTER_BEGIN();
  STREAM(test);
  STREAM_VECTOR(testVector);
  STREAM_REGISTER_FINISH();
}
```

3.5 Communication

Three kinds of communication are implemented in the B-Human framework, and they are all based on the same technology: *message queues*. The three kinds are: *inter-process communication*, *debug communication*, and *team communication*.

3.5.1 Message Queues

The class `MessageQueue` allows storing and transmitting a sequence of messages. Each message has a type (defined in *Src/Tools/MessageQueue/MessageIDs.h*) and a content. Each queue has a maximum size which is defined in advance. On the robot, the amount of memory required is pre-allocated to avoid allocations during runtime. On the PC, the memory is allocated on demand, because several sets of robot processes can be instantiated at the same time, and the maximum size of the queues is rarely needed.

Since almost all data types have streaming operators, it is easy to store them in message queues. The class `MessageQueue` provides different write streams for different formats: messages that are stored through `out.bin` are formatted binary. The stream `out.text` formats data as text and `out.textRaw` as raw text. After all data of a message was streamed into a queue, the message must be finished with `out.finishMessage(MessageID)`, giving it a *message id*, i. e. a type.

```
MessageQueue m;
m.setSize(1000); // can be omitted on PC
m.out.text << "Hello world!";
m.out.finishMessage(idText);
```

To declare a new message type, an id for the message must be added to the enumeration type `MessageID` in *Src/Tools/MessageQueue/MessageIDs.h*. The enumeration type has three sections: the first for representations that should be recorded in log files, the second for team communication, and the last for infrastructure. These sections should always be extended at the end to avoid compatibility issues with existing log files or team mates running an older version of the software. For each new id, a string for the type has to be added to the method `getMessageIDName(MessageID)` in the same file.

Messages are read from a queue through a message handler that is passed to the queue's method `handleAllMessages(MessageHandler&)`. Such a handler must implement the method `handleMessage(InMessage&)`. That method will be called for each message in the queue. It must be implemented in a way as the following example shows:

```
class MyClass : public MessageHandler
{
protected:
  bool handleMessage(InMessage& message)
  {
    switch(message.getMessageID())
    {
    default:
      return false;

    case idText:
      {
        std::string text;
        message.text >> text;
        return true;
              :
```

The handler has to return whether it handled the message or not. Messages are read from a `MessageQueue` via streams. Thereto, `message.bin` provides a binary stream, `message.text` a text stream, and `message.config` a text stream that skips comments.

3.5.2 Inter-process Communication

The representations sent back and forth between the processes *Cognition* and *Motion* are defined in the section *Shared* of the file *Config/Location/<location>/modules.cfg*. The ModuleManager automatically derives the direction in which they are sent from the information about which representation is provided in which process.

3.5.3 Debug Communication

For debugging purposes, there is a communication infrastructure between the processes *Cognition* and *Motion* and the PC. This is accomplished by *debug message queues*. Each process has two of them: `theDebugSender` and `theDebugReceiver`, often also accessed through the references `debugIn` and `debugOut`. The macro `OUTPUT(<id>, <format>, <sequence>)` defined in *Src/Tools/Debugging/Debugging.h* simplifies writing data to the outgoing debug message queue. *id* is a valid message id, *format* is `text`, `bin`, or `textRaw`, and *sequence* is a streamable expression, i. e. an expression that contains streamable objects, which – if more than one – are separated by the streaming operator <<.

```
OUTPUT(idText, text, "Could not load file " << filename << " from " << path);
OUTPUT(idImage, bin, Image());
```

As most of the debugging infrastructure, the macro `OUTPUT` is ignored in the configuration *Release*. Therefore, it should not produce any side effects required by the surrounding code.

For receiving debugging information from the PC, each process also has a message handler, i. e. it implements the method `handleMessage` to distribute the data received.

The process *Debug* manages the communication of the robot control program with the tools on the PC. For each of the other processes (*Cognition* and *Motion*) it has a sender and a receiver for their debug message queues (cf. Fig. 3.1). Messages that arrive via WLAN from the PC are stored in `debugIn`. The method `Debug::handleMessage(InMessage&)` distributes all messages in `debugIn` to the other processes. The messages received from *Cognition* and *Motion* are stored in `debugOut`. When a WLAN connection is established, they are sent to the PC via TCP/IP. To avoid communication jams, it is possible to send a *QueueFillRequest* to the process *Debug*. The command *qfr* to do so is explained in Section 8.5.3.

3.5.4 Team Communication

The purpose of the team communication is to send messages to the other robots in the team. These messages are always broadcasted, so all teammates can receive them. The team communication uses a message queue embedded in a UDP package. The first message in the queue is always `idRobot` that contains the number of the robot sending the message. Thereby, the receiving robots can distinguish between the different packages they receive. The reception of team communication packages is implemented in the module TeamDataProvider. It also implements the network time protocol (*NTP*) and translates time stamps contained in packages it receives into the local time of the robot.

Similar to debug communication, data can be written to the team communication message queue using the macro `TEAM_OUTPUT(<id>, <format>, <sequence>)`. The macro can only be used in process *Cognition*. In contrast to the debug message queues, the one for team communication is rather small (1396 bytes). So the amount of data written should be kept to a minimum. In addition, team packages are only broadcasted approximately every 100 ms. Hence, and due to the use of UDP in general, data is not guaranteed to reach its intended receivers. The representation *TeamMateData* contains a flag that states whether a team communication package will be sent out in the current frame or not.

3.6 Debugging Support

Debugging mechanisms are an integral part of the *B-Human* framework. They are all based on the debug message queues already described in Section 3.5.3. All debugging mechanisms are available in all project configurations but *Release*. In *Release*, they are completely deactivated (i. e. not even part of the executable), and the process *Debug* is not started.

3.6.1 Debug Requests

Debug requests are used to enable and disable parts of the source code. They can be seen as a runtime switch available only in debugging mode. This can be used to trigger certain debug messages to be sent, as well as to switch on certain parts of algorithms. Two macros ease the use of the mechanism as well as hide the implementation details:

DEBUG_RESPONSE(<id>, <statements>) executes the statements if the debug request with the name `id` is enabled.

DEBUG_RESPONSE_NOT(<id>, <statements>) executes the statements if the debug request with the name `id` is *not* enabled. The statements are also executed in the release configuration of the software.

These macros can be used anywhere in the source code, allowing for easy debugging. For example:

```
DEBUG_RESPONSE("test", test());
```

This statement calls the method `test()` if the debug request with the identifier ”test” is enabled. Debug requests are commonly used to send messages on request, as the following example shows:

```
DEBUG_RESPONSE("sayHello", OUTPUT(idText, text, "Hello"); );
```

This statement sends the text “Hello” if the debug request with the name `"sayHello"` is activated. Please note that only those debug requests are usable that are in the current path of execution. This means that only debug request in those modules can be activated that are currently executed. To determine which debug requests are currently available, a method called *polling* is employed. It asks all debug responses to report the name of the debug request that would activate it. This information is collected and sent to the PC (cf. command *poll* in Sect. 8.5.3).

3.6.2 Debug Images

Debug images are used for low level visualization of image processing debug data. They can either be displayed as background image of an image view (cf. Sect. 8.3.1) or in a color space view (cf. Sect. 8.3.2). Each debug image has an associated textual identifier that allows referring to it during image manipulation, as well as for requesting its creation from the PC. The identifier can be used in a number of macros that are defined in file *Src/Tools/Debugging/DebugImages.h*, and that facilitate the manipulation of the debug image.

DECLARE_DEBUG_IMAGE(<id>) declares a debug image with the specified identifier. This statement has to be placed where declarations of variables are allowed, e. g. in a class declaration.

INIT_DEBUG_IMAGE(<id>, image) initializes the debug image with the given identifier with the contents of an image.

INIT_DEBUG_IMAGE_BLACK(<id>) initializes the debug image as black.

SEND_DEBUG_IMAGE(<id>) sends the debug image with the given identifier as bitmap to the PC.

SEND_DEBUG_IMAGE_AS_JPEG(<id>) sends the debug image with the given identifier as JPEG-encoded image to the PC.

DEBUG_IMAGE_GET_PIXEL_<channel>(<id>, <x>, <y>) returns the value of a color channel (Y, U, or V) of the pixel at (x, y) of the debug image with the given identifier.

DEBUG_IMAGE_SET_PIXEL_YUV(<id>, <xx>, <yy>, <y>, <u>, <v>) sets the Y, U, and V-channels of the pixel at (xx, yy) of the image with the given identifier.

DEBUG_IMAGE_SET_PIXEL_<color>(<id>, <x>, <y>) sets the pixel at (x, y) of the image with the given identifier to a certain color.

GENERATE_DEBUG_IMAGE(<id>, <statements>) only executes a sequence of statements if the creation of a certain debug image is requested. This can significantly improve the performance when a debug image is not requested, because for each image manipulation it has to be tested whether it is currently required or not. By encapsulating them in this macro (and maybe in addition in a separate method), only a single test is required.

DECLARE_DEBUG_GRAY_SCALE_IMAGE(<id>) declares a grayscale debug image. Grayscale debug images only represent the brightness channel of an image, even reducing it to only seven bits per pixel. The remaining 128 values of each byte representing a pixel are used for drawing colored pixels from a palette of predefined colors.

INIT_DEBUG_GRAY_SCALE_IMAGE(<id>, image) initializes the grayscale debug image with the given identifier with the contents of an image.

INIT_DEBUG_GRAY_SCALE_IMAGE_BLACK(<id>) initializes the grayscale debug image as black.

SEND_DEBUG_GRAY_SCALE_IMAGE(<id>) sends the grayscale debug image with the given identifier as bitmap to the PC.

SET_COLORED_PIXEL_IN_GRAY_SCALE_IMAGE(<id>, <x>, <y>, <color>) sets a colored pixel in a grayscale debug image. All available colors are defined in class `ColorIndex` (declared in file *Src/Tools/ColorIndex.h*).

These macros can be used anywhere in the source code, allowing for easy creation of debug images. For example:

```
DECLARE_DEBUG_IMAGE("test");
INIT_DEBUG_IMAGE("test", image);
DEBUG_IMAGE_SET_PIXEL_YUV("test", 0, 0, 0, 0, 0);
SEND_DEBUG_IMAGE_AS_JPEG("test");
```

The example initializes a debug image from another image, sets the pixel $(0, 0)$ to black and sends it as a JPEG-encoded image to the PC.

3.6.3 Debug Drawings

Debug drawings provide a virtual 2-D drawing paper and a number of drawing primitives, as well as mechanisms for requesting, sending, and drawing these primitives to the screen of the PC. In contrast to debug images, which are raster-based, debug drawings are vector-based, i. e., they store drawing instructions instead of a rasterized image. Each drawing has an identifier and an associated type that enables the application on the PC to render the drawing to the right kind of drawing paper. In addition, a description can be specified (currently, it is not used). In the B-Human system, two standard drawing papers are provided, called `drawingOnImage` and `drawingOnField`. This refers to the two standard applications of debug drawings, namely drawing in the system of coordinates of an image and drawing in the system of coordinates of the field. Hence, all debug drawings of the type `drawingOnImage` can be displayed in an image view (cf. Sect. 8.3.1) and all drawings of type `drawingOnField` can be rendered into a field view (cf. Sect. 8.3.3).

The creation of debug drawings is encapsulated in a number of macros in *Src/Tools/Debugging/DebugDrawings.h*. Most of the drawing macros have parameters such as pen style, fill style, or color. Available pen styles (`ps_solid`, `ps_dash`, `ps_dot`, and `ps_null`) and fill styles (`bs_solid` and `bs_null`) are part of the class `Drawings`. Colors can be specified as `ColorRGBA` or using the enumeration type `ColorClasses::Color`. A few examples for drawing macros are:

DECLARE_DEBUG_DRAWING(<id>, <type>) declares a debug drawing with the specified *id* and *type*. In contrast to the declaration of debug images, this macro has to be placed in a part of the code that is regularly executed.

CIRCLE(<id>, <x>, <y>, <radius>, <penWidth>, <penStyle>, <penColor>, <fillStyle>, <fillColor>) draws a circle with the specified radius, pen width, pen style, pen color, fill style, and fill color at the coordinates (x, y) to the virtual drawing paper.

LINE(<id>, <x1>, <y1>, <x2>, <y2>, <penWidth>, <penStyle>, <penColor>) draws a line with the pen color, width, and style from the point $(x1, y1)$ to the point $(x2, y2)$ to the virtual drawing paper.

DOT(<id>, <x>, <y>, <penColor>, <fillColor>) draws a dot with the pen color and fill color at the coordinates (x, y) to the virtual drawing paper. There also exist two macros `MID_DOT` and `LARGE_DOT` with the same parameters that draw dots of larger size.

DRAWTEXT(<id>, <x>, <y>, <fontSize>, <color>, <text>) writes a text with a font size in a color to a virtual drawing paper. The upper left corner of the text will be at coordinates (x, y).

TIP(<id>, <x>, <y>, <radius>, <text>) adds a tool tip to the drawing that will pop up when the mouse cursor is closer to the coordinates (x, y) than the given radius.

ORIGIN(<id>, <x>, <y>, <angle>) changes the system of coordinates. The new origin will be at (x, y) and the system of coordinates will be rotated by *angle* (given in radians). All further drawing instructions, even in other debug drawings that are rendered afterwards in the same view, will be relative to the new system of coordinates, until the next origin is set. The origin itself is always absolute, i. e. a new origin is not relative to the previous one.

COMPLEX_DRAWING(<id>, <statements>) only executes a sequence of statements if the creation of a certain debug drawing is requested. This can significantly improve the performance when a debug drawing is not requested, because for each drawing instruction it has to be tested whether it is currently required or not. By encapsulating them in this macro (and maybe in addition in a separate method), only a single test is required. However, the macro `DECLARE_DEBUG_DRAWING` must be placed outside of `COMPLEX_DRAWING`.

These macros can be used wherever statements are allowed in the source code. For example:

```
DECLARE_DEBUG_DRAWING("test", "drawingOnField");
CIRCLE("test", 0, 0, 1000, 10, Drawings::ps_solid, ColorClasses::blue,
                               Drawings::bs_solid, ColorRGBA(0, 0, 255, 128));
```

This example initializes a drawing called `test` of type `drawingOnField` that draws a blue circle with a solid border and a semi-transparent inner area.

3.6.4 Plots

The macro `PLOT(<id>, <number>)` allows plotting data over time. The plot view (cf. Sect. 8.3.7) will keep a history of predefined size of the values sent by the macro `PLOT` and plot them in different colors. Hence, the previous development of certain values can be observed as a time series. Each plot has an identifier that is used to separate the different plots from each other. A plot view can be created with the console commands *vp* and *vpd* (cf. Sect. 8.5.3).

For example, the following statement plots the measurements of the gyro for the pitch axis. Please note that the measurements are converted to degrees.

```
PLOT("gyroY", toDegrees(theSensorData.data[SensorData::gyroY]));
```

The macro `DECLARE_PLOT(<id>)` allows using the `PLOT(<id>, <number>)` macro within a part of code that is not regularly executed as long as the `DECLARE_PLOT(<id>)` macro is executed regularly.

3.6.5 Modify

The macro `MODIFY(<id>, <object>)` allows reading and modifying of data on the actual robot during runtime. Every streamable data type (cf. Sect. 3.4.3) can be manipulated and read, because its inner structure is gathered while it is streamed. This allows generic manipulation of runtime data using the console commands *get* and *set* (cf. Sect. 8.5.3). The first parameter of `MODIFY` specifies the identifier that is used to refer to the object from the PC, the second parameter is the object to be manipulated itself. When an object is modified using the console command *set*, it will be overridden each time the `MODIFY` macro is executed.

```
int i = 3;
MODIFY("i", i);
WalkingEngineParameters p;
MODIFY("parameters:WalkingEngine", p);
```

The macro `PROVIDES` of the module framework (cf. Sect. 3.3) also is available in versions that include the `MODIFY` macro for the representation provided (e. g. `PROVIDES_WITH_MODIFY`). In these cases the representation, e. g., *Foo* is modifiable under the name `representation:Foo`.

3.6.6 Stopwatches

Stopwatches allow the measurement of the execution time of parts of the code. The statements the runtime of which should be measured have to be placed into the macro `STOP_TIME_ON_REQUEST(<id>, <statements>)` (declared in *Src/Tools/Debugging/Stopwatch.h*) as second parameter. The first parameter is a string used to identify the time measurement. To activate a certain time measurement, e. g., `Foo`, a debug request `stopwatch:Foo` has to be sent. The measured time can be seen in the timing view (cf. Sect. 8.3.8). By default, a stopwatch is already defined for each representation that is currently provided. In the release configuration of the code, all stopwatches in process *Cognition* can be activated by sending the release option *stopwatches* (cf. command *ro* in Sect. 8.5.3).

An example to measure the runtime of a method called `myCode`:

```
STOP_TIME_ON_REQUEST("myCode", myCode(); );
```

Chapter 4

Cognition

In the B-Human system, the process *Cognition* (cf. Sect. 3.2) can be structured into the three functional units *perception*, *modeling*, and *behavior control*. The major task of the perception modules is to detect landmarks such as goals and field lines, as well as the ball in the image provided by the camera. The modeling modules work on these percepts and determine the robot's position on the field, the relative position of the goals, the position and speed of the ball, and the free space around the robot. Only these modules are able to provide useful information for the behavior control that is described separately (cf. Chapter 6).

4.1 Perception

B-Human uses a scan line-based perception system. The *YUV422* images provided by the Nao camera have a resolution of 640×480 pixels. They are interpreted as *YUV444* images with a resolution of 320×240 pixels by ignoring every second row and the second Y channel of each *YUV422* pixel pair. The images are scanned on vertical scan lines (cf. Fig. 4.8). Thereby, the actual amount of scanned pixels is much smaller than the image size. The perception modules provide representations for the different features. The *BallPercept* contains information about the ball if it was seen in the current image. The *LinePercept* contains all field lines, intersections, and the center-circle seen in the current image, and the *GoalPercept* contains information about the goals seen in the image. All information provided by the perception modules is relative to the robot's position.

An overview of the perception modules and representations is visualized in Fig. 4.1.

4.1.1 Definition of Coordinate Systems

The global coordinate system (cf. Fig. 4.2) is described by its origin lying at the center of the field, the x-axis pointing toward the opponent goal, the y-axis pointing to the left, and the z-axis pointing upward. Rotations are specified counter-clockwise with the x-axis pointing toward $0°$, and the y-axis pointing toward $90°$.

In the egocentric system of coordinates (cf. Fig. 4.3) the axes are defined as followed: the x-axis points forward, the y-axis points to the left, and the z-axis points upward.

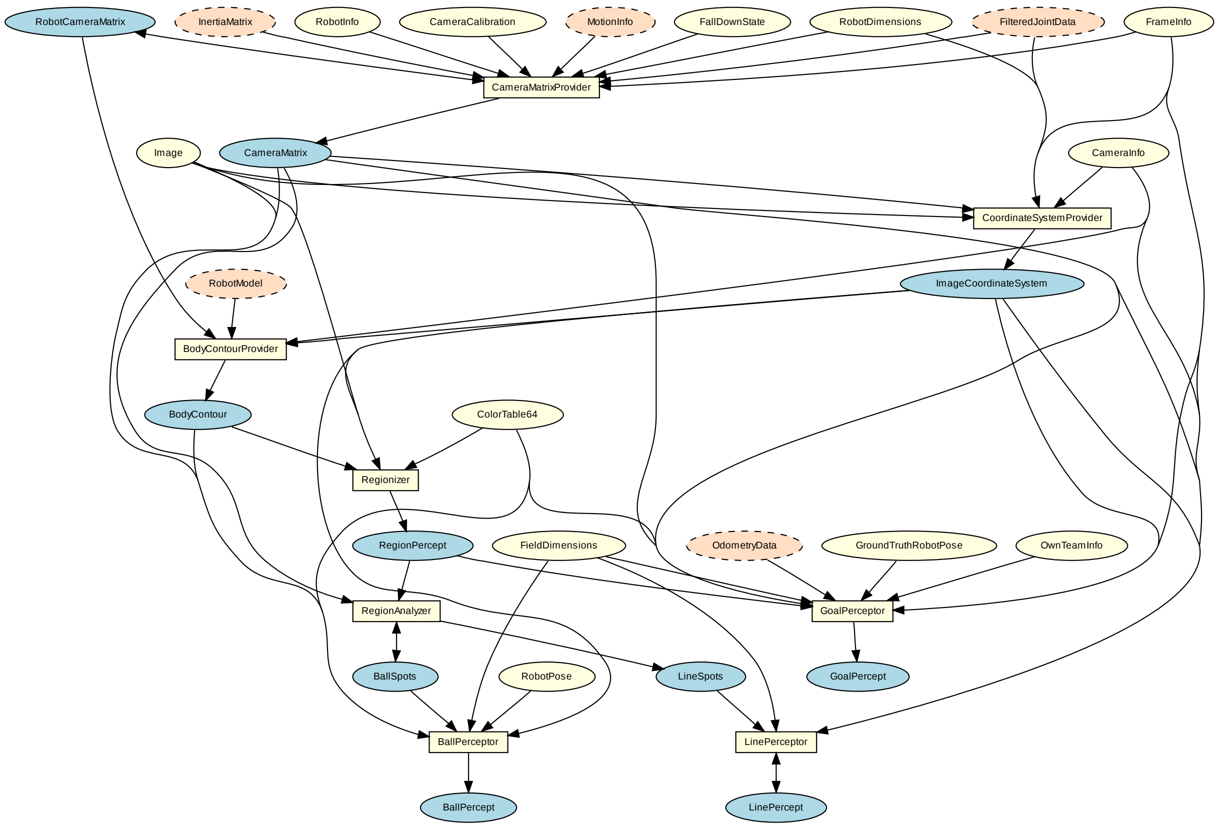


Figure 4.1: Perception module graph

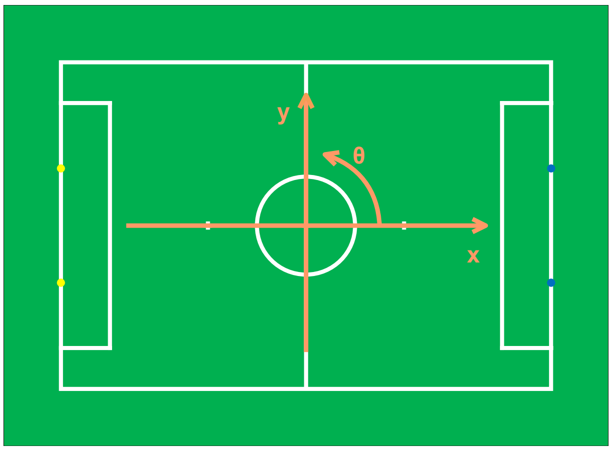


Figure 4.2: Visualization of the global coordinate system

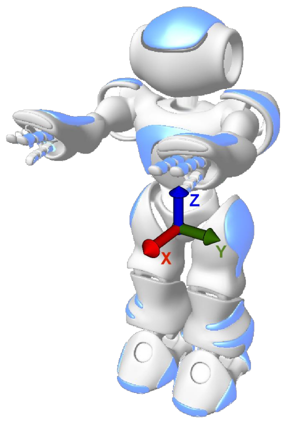


Figure 4.3: Visualization of the robot relative coordinate system

4.1.1.1 Camera Matrix

The *CameraMatrix* is a representation containing the transformation matrix of the lower camera of the Nao (we only use the lower camera) that is provided by the CameraMatrixProvider. It is computed based on the *InertiaMatrix* that represents the orientation and position of a specific point within the robot's torso relative to the ground (cf. sect. 5.1.5). Using the *RobotDimensions* and the current position of the joints the transformation of the camera matrix relative to the inertia matrix is computed (this relative transformation matrix is also provided as *RobotCam-*

eraMatrix, which is used to compute the *BodyContour* (cf. Sect. 4.1.2)). In addition to these fixed parameters some robot-specific parameters from the *CameraCalibration* are integrated that are necessary because the camera cannot be mounted perfectly plain and the torso is not always perfectly vertical (a small variation can lead to significant errors when projecting farther objects onto the field).
The *CameraMatrix* is used for projecting objects onto the field as well as the creation of the *ImageCoordinateSystem* (cf. Sect. 4.1.1.2).

For the purpose of calibrating the mentioned robot-specific parameters there is a debug drawing that projects the field lines into the camera image. To activate this drawing type *vid raw module:-CameraMatrixProvider:calibrationHelper* in the simulator console. This drawing is helpful for calibrating because the real lines and the projected lines only match if the camera matrix and hence the camera calibration is correct (assuming that the real position corresponds to the self-localization of the robot).

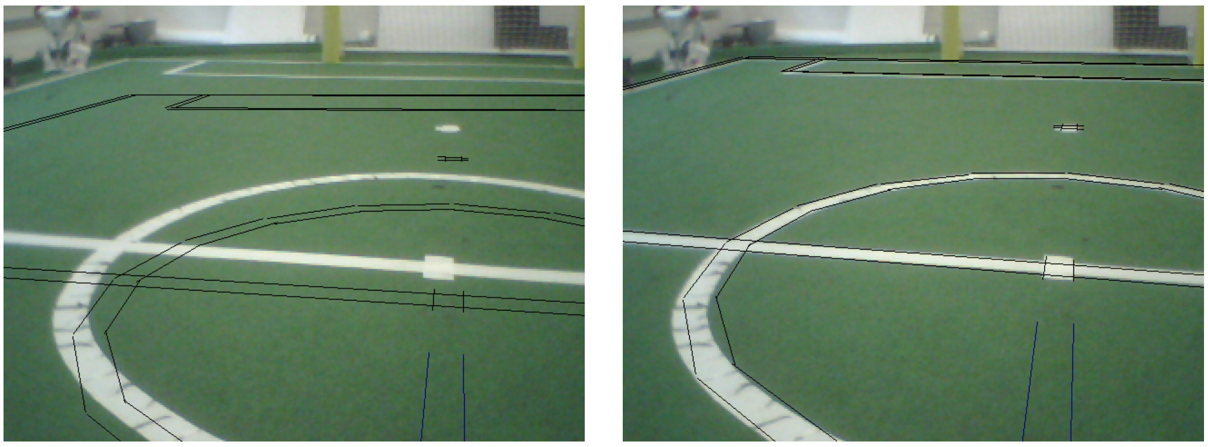


Figure 4.4: Camera calibration using *calibrationHelper*: Projected lines before (left) and after (right) calibration procedure

The calibration procedure is as follows:

1. Connect the simulator to a robot on the field.
2. Place the robot on one of the crosses facing the opposite goal.
3. Run the SimRobot configuration file *CameraCalibration.con* (in the console type `call CameraCalibration`).
4. Modify the parameters of *CameraCalibration* so that the projected lines match the field lines in the image.
5. Move the head to check that the lines match for different rotations of the head.
6. Repeat 4. and 5. until the lines match in each direction.

4.1.1.2 Image Coordinate System

Based on the camera transformation matrix, another coordinate system is provided which applies to the camera image. The *ImageCoordinateSystem* is provided by the module CoordinateSystem-Provider. Its origin of the y-coordinate lies on the horizon within the image (even if it is not

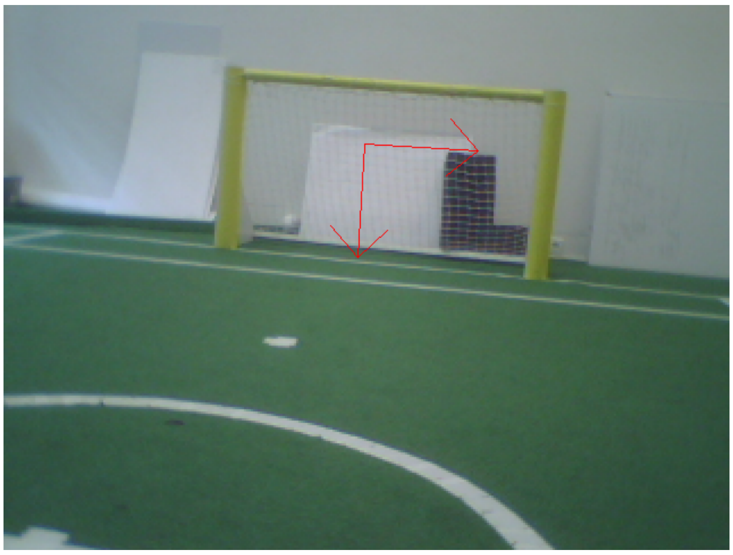


Figure 4.5: Origin of the *ImageCoordinateSystem*

visible in the image). The x-axis points right along the horizon whereas the y-axis points downwards orthogonal to the horizon (cf. Fig. 4.5). For more information see also [23].

The *ImageCoordinateSystem* provides a mechanism to compensate image distortion caused by the rolling shutter that is based on the image recording time and the speed of the head joints. For a detailed description see [20].

4.1.2 BodyContourProvider

If the robot sees parts of its body, it might confuse white areas with field lines and – under certain conditions – red parts with the ball. However, by using forward kinematics, the robot can actually know where its body is visible in the camera image and exclude these areas from image processing. This is achieved by modeling the boundaries of body parts that are potentially visible in 3-D (cf. Fig. 4.6 left) and projecting them back to the camera image (cf. Fig. 4.6 right). The part of that projection that intersects with the camera image or is above it is provided in the representation *BodyContour*. It is used by the image processor as lower clipping boundary. The projection relies on *ImageCoordinateSystem*, i. e., the linear interpolation of the joint angles to match the time when the image was taken. However if joints accelerate or decelerate, the projection may not be accurate, as can be seen in Fig. 4.6 right), where the foot just stopped after a kick. In that case, the clipping region might be too big or too small.

4.1.3 Image Processing

The image processing is split into the three steps segmentation and region-building (cf. Sect. 4.1.3.1), region classification (cf. Sect. 4.1.3.2) and feature extraction (cf. Sect. 4.1.3.3, 4.1.3.4, 4.1.3.5). Since the goals are the only features that are above the horizon and the field border, the goal detection is based on special segments and is not integrated in the region classification.

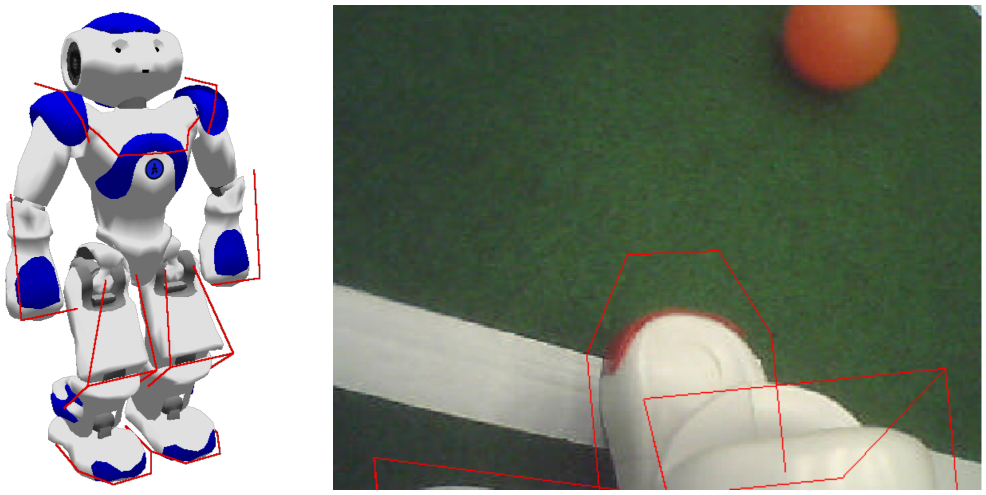


Figure 4.6: Body contour in 3-D (left) and projected to the camera image after kicking the ball (right).

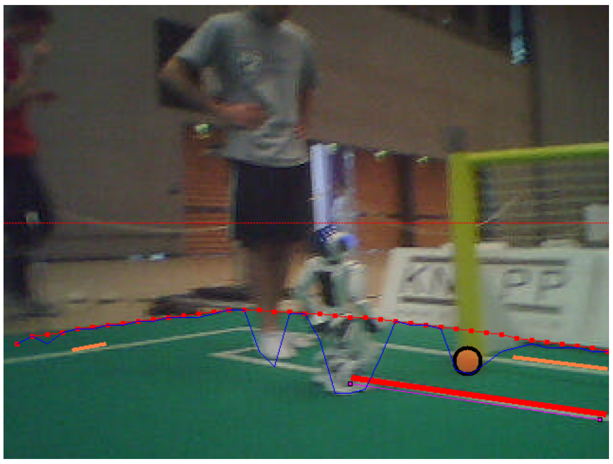


Figure 4.7: Field borders: the small red-dotted line is the horizon, the blue line connects the green points found, the red line is the upper part of the convex hull with a dot for each scan line, the ball in front of the post would be missed without the convex hull.

4.1.3.1 Segmentation and Region-Building

The Regionizer works on the camera image and creates regions based on segments found on scan lines. It provides the *RegionPercept* containing the segments, the regions, the field borders and the horizontal and vertical goal segments.
First the field borders are detected. This is done by running scan lines, starting from the horizon, downwards until a green segment of a minimum length is found. From these points the upper half of the convex hull is used as field border. Using the convex hull ensures that we cannot miss a ball that is in front of a goal post or a robot and therefore has no green above (cf. Fig. 4.7). Since all features that are detected based on regions are on the field (ball and field lines) further segmentation starts from these field border points. Because we do not want to accidentally detect features inside the body of the robot, the *BodyContour* is used to determine the end of each scan line. Now scan lines running from the field border to the bottom of the image, clipped by the *BodyContour*, are used to create segments. To be tolerant against noise in the segmentation, a certain number of different colored pixels in a row may be skipped without starting a new segment. Since we want to create regions based on the segments and our scan

lines are not very close to each other, non-green segments are explored. This means we are trying to find equally colored runs between the current scan line and the last one which touches the segment and starts before or ends after the current segment. This is necessary since a field line that is far away might create segments that do not touch each other because of the distance of the scan lines. However, we still want to be able to unite them to a region (cf. Fig. 4.8).
For the goal detection two special scans are done to detect vertical and horizontal yellow or blue segments above the horizon. The horizontal scan lines are continued below the horizon until there is a scan line in which no yellow or blue segment is found. These segments are stored separately from the normal segments in the *RegionPercept* and are only used by the GoalPerceptor.
Based on these segments regions are created using the algorithm shown in Algorithm 1 and 2. It iterates over all segments (sorted by y- and x-coordinates) and connects the current segment to the regions already created or creates a new one in each iteration. To connect each segment to the regions already found, a function is called that takes as input the segment and a pointer to a segment in the previous column that is the first segment that might touch the segment to be connected. The function returns a pointer to the segment in the last column that might touch the next segment. This pointer is passed again to the function in the next iteration for the next segment. The algorithm is capable of handling overlapping segments, which is needed because the region-building is done on the explored segments. While building the regions, information about neighboring regions is collected and stored within the regions.

Algorithm 1 Region-building

```
$lastColumnPointer \leftarrow NULL$
$firstInColumn \leftarrow NULL$
$s' \leftarrow NULL$
**for all** $s \in segments$ **do**
  **if** $s.color = green$ **then**
    **continue**
  **end if**
  **if** $column(s) \neq column(s')$ **then**
    $lastColumnPointer \leftarrow firstInColumn$
    $firstInColumn \leftarrow s$
  **end if**
  $lastColumnPointer \leftarrow connectToRegions(s, lastColumnPointer)$
  $s' \leftarrow s$
**end for**
```

We do not create green regions, since green is treated as background and is only needed to determine the amount of green next to white regions, which can be determined based on the segments.
Two segments of the same color touching each other need to fulfill certain criteria to be united to a region:

- For white and uncolored regions there is a maximum region size
- The length ratio of the two touching segments may not exceed a certain maximum
- For white regions the change in direction may not exceed a certain maximum (vector connecting the middle of the segments connected to the middle of the next segment is treated as direction)

Algorithm 2 ConnectToRegions(s, lastColumnPointer)

```
if $lastColumnPointer = NULL$ then
  $createNewRegion(s)$
  return $NULL$
end if
while $lastColumnPointer.end < s.exploredStart$ & $column(lastColumnPointer) + 1 = column(s)$ do
  $lastColumnPointer \leftarrow lastColumnPointer.next()$
end while
if $column(lastColumnPointer) + 1 \neq column(s)$ then
  $createNewRegion(s)$
  return $NULL$
end if
if $lastColumnPointer.start \leq s.exploredEnd$ & $lastColumnPointer.color = s.color$ then
  $uniteRegions(lastColumnPointer, s)$
end if
$lastColumnPointer' \leftarrow lastColumnPointer$
while $lastColumnPointer'.end \leq s.exploredEnd$ do
  $lastColumnPointer' \leftarrow lastColumnPointer'.next()$
  if $column(lastColumnPointer') + 1 \neq column(s)$ then
    if $!s.hasRegion$ then
      $createNewRegion(s)$
    end if
    return $lastColumnPointer$
  end if
  if $lastColumnPointer'.start \leq s.exploredEnd$ & $lastColumnPointer'.color = s.color$ then
    $uniteRegions(lastColumnPointer', s)$
  else
    return $lastColumnPointer$
  end if
end while
if $!s.hasRegion$ then
  $createNewRegion(s)$
end if
return $lastColumnPointer$
```

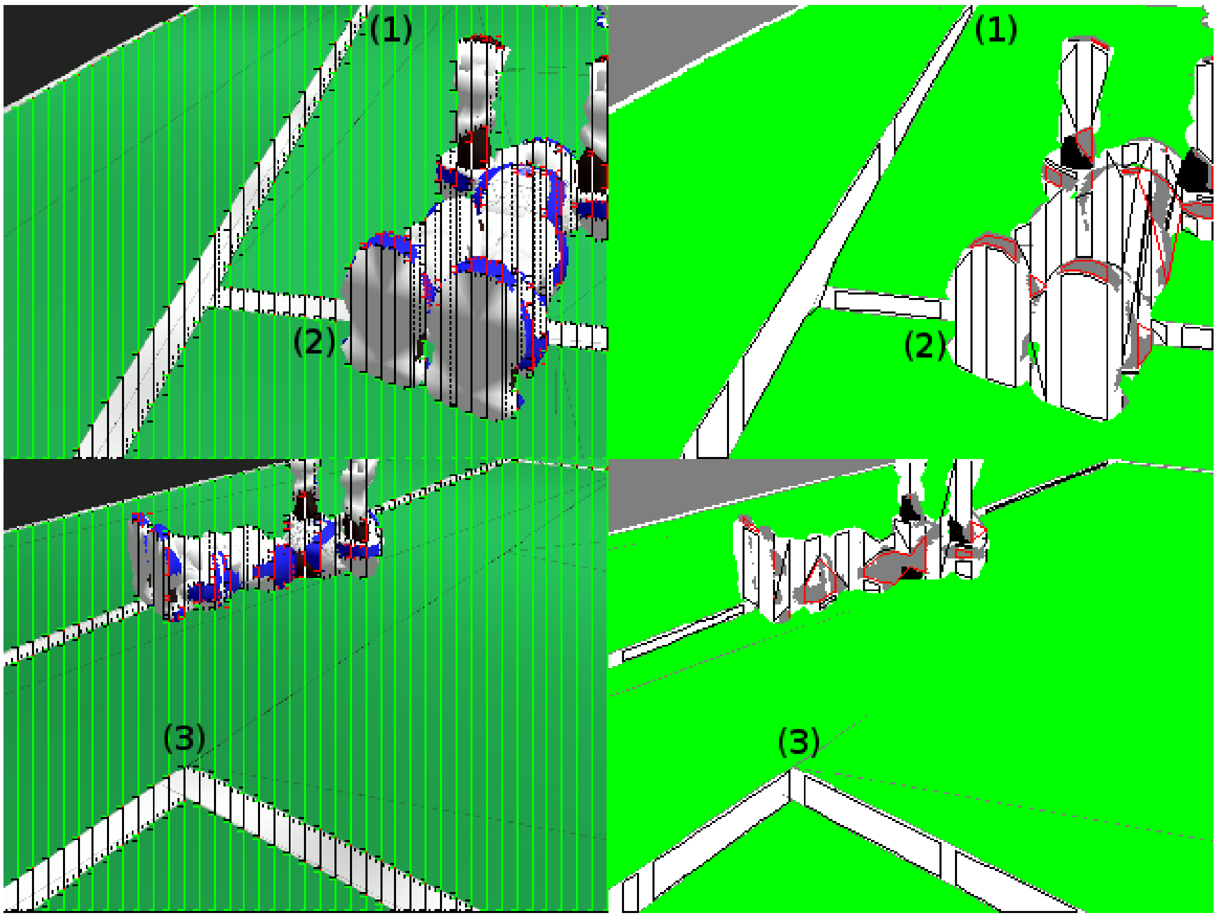


Figure 4.8: Segmentation and region-building: (1) segment is not touching the other, but the explored runs (dotted lines) are touching the other segment, (2) not connected because of length ratio, (3) not connected because of change in direction.

- If two white segments are touching each other and both already are connected to a region, they are not united

These criteria (and all other thresholds mentioned before) are configurable through the file *regionizer.cfg*. For some colors these features are turned off in the configuration file. These restrictions are especially needed for white regions, since we do not want to have a single big region containing all field lines and robots. The result of these restrictions is that we most likely get small straight white regions (cf. Fig. 4.8). For example, at a corner the change in direction of the white segments should exceed the maximum change in direction, and therefore they are not united or if a robot stands on a field line, the length ratio of the two segments is most likely too big and the regions are not united.

4.1.3.2 Region Classification

The region classification is done by the RegionAnalyzer. It classifies the regions within the *RegionPercept* whether they could be parts of a line or the ball and discards all others. It provides the *LineSpots* and *BallSpots*.
The RegionAnalyzer iterates over all regions in the *RegionPercept* and filters all white regions through a filter that determines whether the region could be a part of a line and filters all orange regions through a filter that determines whether the region could be a ball.

A white region needs to fulfill the following conditions to be accepted as a part of a line:

- The region must consists of a certain number of segments, and it must have a certain size. If a region does not have enough children but instead has a size that is bigger than a second threshold it is also accepted as a line. This case occurs for example when a vertical line only creates a single very long segment.
- The axis of orientation must be determinable (since this is the base information passed to further modules).
- The size of neighboring uncolored regions must not exceed a certain size and the ratio of the size of the white region and the neighboring uncolored regions must not exceed a certain ratio (this is because robot parts are most likely classified as uncolored regions).
- A horizontally oriented white region must have a certain amount of green above and below, while a vertically oriented white region must have a certain amount of green on its left and right side.

For each region that was classified as a part of a line the center of mass, the axis of orientation, the size along the axis of orientation, the size orthogonal to the axis of orientation, and the start and end point of the axis of orientation in image coordinates are stored as *LineSpot* in the *LineSpots*. All white regions that did not fulfill the criteria to be a part of a line are filtered again through a very basic filter that determines whether the region could be a part of a robot. The conditions for this filter are the following:

- Does the region have a certain size?
- Does the region have a vertical orientation (since most regions found in robots have a vertical orientation)?
- Does the width/height ratio exceed a certain threshold (since most regions found in robots have a big ratio)?

If all of these conditions are met a *NonLineSpot* is added to the *LineSpots*. These *NonLineSpots* are used in the LinePerceptor to find regions in the image where there are a lot of *NonLineSpots*. These are excluded from line detection.
Orange regions need to have a certain eccentricity to be accepted as BallSpots. For orange regions that are accepted as possible balls the center of mass and the eccentricity is stored as *BallSpot* in the *BallSpots*.
All thresholds are configurable through the file *regionAnalyzer.cfg*.

4.1.3.3 Detecting Lines

The LinePerceptor works on the *LineSpots* provided by the RegionAnalyzer. It clusters the *LineSpots* to lines and tries to find a circle within the line spots not clustered to lines. Afterwards the intersections of the lines and the circle are computed. The results are stored in the *LinePercept*.
To avoid detecting false lines in robots first of all the *NonLineSpots* within the *LineSpots* are clustered to so called *ban sectors*. The *NonLineSpots* are transformed to field coordinates using the 3-D camera equation. Since the *NonLineSpots* are most likely parts of robots that are above the field, this creates a very characteristic scheme in field coordinates (cf. Fig. 4.9). After creating the ban sectors, the LinePerceptor creates line segments from the line spots. For each line

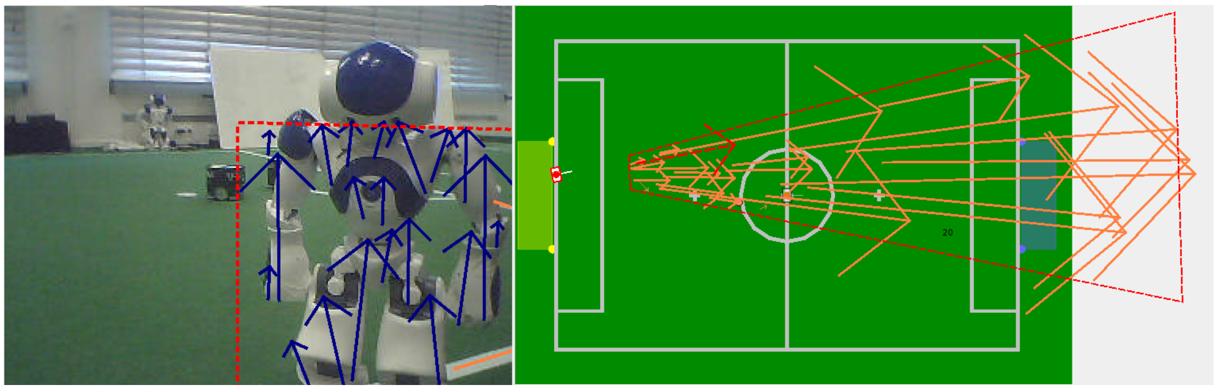


Figure 4.9: Ban sector built from *NonLineSpots*. The red arrow is a false line spot detected in the robot and discarded because of the ban sector.

spot that has a certain width/height ratio the start and end point of the spot is transformed to field coordinates using the 3-D camera equation. If one of the two points is farther away than a certain threshold the segment is discarded, since we can assume no line can be farther away than the diagonal of the field size. The threshold is chosen a little bit smaller than the diagonal of the field size because a line which has almost this distance is too small to be recognized by our vision system. If the spot is similar to the regions that are created by robots (certain width/height ratio, vertical, and a certain length), the LinePerceptor checks whether that spot is inside a ban sector. If that is the case the segment is discarded. For all segments created the Hesse normal form is calculated.
From these segments the lines are built. This is done by clustering the line segments. The clustering is done by the algorithm shown in cf. Algorithm 3. The basic idea of the algorithm is similar to the quality threshold clustering algorithm introduced by Heyer et al. in [8], but it ensures that it runs in the worst-case-scenario in $\mathcal{O}(n^2)$ runtime. Therefore it is not guaranteed to find optimal clusters. Since the number of line spots is limited by the field setup, practical usage showed that the algorithm has an acceptable runtime and delivers satisfiable results. The difference of the directions and distances of the Hesse normal form of two segments need to be less than a certain threshold to be accepted as parts of the same line. Each cluster of segments also needs a segment with a length bigger than a certain threshold. This is necessary to avoid creating lines from small pieces, for example a cross and a part of the circle. The lines are also represented as Hesse normal form.

All remaining line segments are taken into account for the circle detection. For each pair of segments with a distance smaller than a threshold the intersection of the perpendicular from the middle of the segments is calculated. If the distance of this intersection is close to the real circle radius, for each segment a spot is generated which has the distance of the radius to the segment. After the spots are created the same clustering algorithm used for the lines is used to find a cluster for the circle. As soon as a cluster is found which fulfills the criteria to be a circle it is assumed to be the circle (cf. Fig. 4.10). For all remaining line segments which have a certain length additional lines are created. Since it might happen that a circle is detected but single segments on the circle were not recognized as part of it, all lines which are aligned on the circle are deleted. It might also happen that a single line in the image created multiple lines (because the line was not clustered but the single segments were long enough to create lines on their own) therefore lines which are similar (with respect to the Hesse normal form) are merged together. Since it might happen that single segments where not clustered to a line (for example because the line spot is not perfectly aligned with the line, what can be caused by an inaccurate segmentation) the remaining single segments are merge to the lines, if the distance of their start

Algorithm 3 Clustering LineSegments

```
while $lineSegments \neq \emptyset$ do
  $s \leftarrow lineSegments.pop()$
  $supporters \leftarrow \emptyset$
  $maxSegmentLength \leftarrow 0$
  for all $s' \in lineSegments$ do
    if $similarity(s, s') < similarityThreshold$ then
      $supporters.add(s')$
      if $length(s') > maxSegmentLength$ then
        $maxSegmentLength = length(s')$
      end if
    end if
  end for
  if $supporters.size() > supporterThreshold$ and $maxSegmentLength > segmentLengthThreshold$ then
    $createLine(\{s\} \cup supporters)$
    $lineSegments \leftarrow lineSegments \backslash supporters$
  end if
end while
```

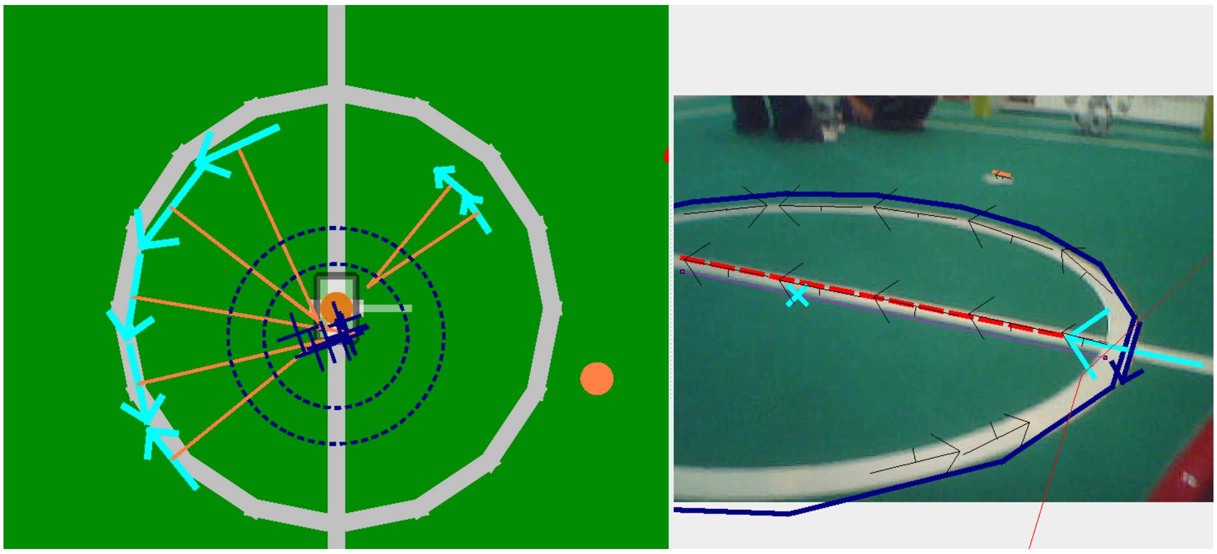


Figure 4.10: Circle detection: blue crosses: circleSpots, blue circles: threshold area, orange lines: perpendiculars of segments

and end point is close to a line. For each resulting line the summed length of the segments must cover a certain percentage of the length of the line. Otherwise the line will be discarded. This avoids creating lines for segments which a far away from each other, for example the cross and a part of the circle.

All resulting lines are intersected. If the intersection of two lines is between the start and end point or close enough to the start/end point of both lines, an intersection is created. There are different types of intersections (L, T and X). Whether an intersection is an L, T or X intersection is determined by the distance of the intersection to the start/end points of the lines.

All thresholds used in the LinePerceptor are configurable through the *lineperceptor.cfg*.

4.1.3.4 Detecting the Goal

The GoalPerceptor works directly on the *RegionPercept* output of the Regionizer. It provides the representation *GoalPercept* that contains information about the goal posts.

The main idea of the GoalPerceptor is to perform a Hough-Transformation on the horizontal goal segments in the *RegionPercepts* to search for long vertical lines in the color of the two goals in the image and to perform some further checks to decide whether these objects might be a goal post or not. If they are supposed to be goal posts the GoalPerceptor also tries to determine the type of the detected goal post, which means to decide whether it is a left or a right post of the goal.
Because of performance issues the GoalPerceptor at first ignores the angle of the goal posts in the image and assumes that in most of the relevant cases the goal posts lie roughly vertical in the image. After one or more long vertical segments have been detected in the image, the precise top and bottom of these percepts are explored by looking into the image (small noise allowed). The next step is to check in these percepts whether they match all the characteristic properties of a goal post and to make sure not to accept blue robot parts or things outside the field as goal posts. The most important checks are:

- Is there enough green below the foot point of the percept?
- Is the bottom of the percept under the calculated horizon and the top above it?
- Does the expected width of the goal post match the width of the percept?
- Is the calculated distance of the goal post within a plausible range?

Percepts that do not fulfill these criteria are discarded. For the remaining percepts a determination of the side is performed. If there are exactly two remaining percepts of the same color this can be done easily. Otherwise the surrounding of the head point of the only remaining percept is scanned in order to try to detect parts of the crossbar and to determine the side of the post.

Since for a robot standing on the field it is impossible to see two goal posts of different colors at the same time. This fact is also used for another plausibility filter. If the GoalPerceptor perceives two goal posts of different colors it discards both, if it sees a complete goal consisting of two clearly visible goal posts and an additional single goal post of the other color it discards the single one and if it detects two complete goals consisting of two goal posts each it discards both of them. After this check the remaining goal posts are inserted into the *GoalPercept*, containing information about their position in the image, their position relative to the robot, their color, their type (which side), the time when they were perceived and the type how their distance to the robot was calculated (by projecting of the bottom point to the field or by using their height in the image).

4.1.3.5 Detecting the Ball

The BallPerceptor requires the representation *BallSpots* that is provided by the module Region-Analyzer. The *BallSpots* is a list containing the center of mass of each orange region which could be considered as a ball.

The BallPerceptor calculates the validity (0...1) for each *BallSpot* in several steps. To determine whether an orange cluster can be considered as a ball, the image is scanned in eight different directions (left and right, up and down, and in all four diagonal directions) starting at the *BallSpot*. If the border of the image is reached or the last five points viewed are not orange,

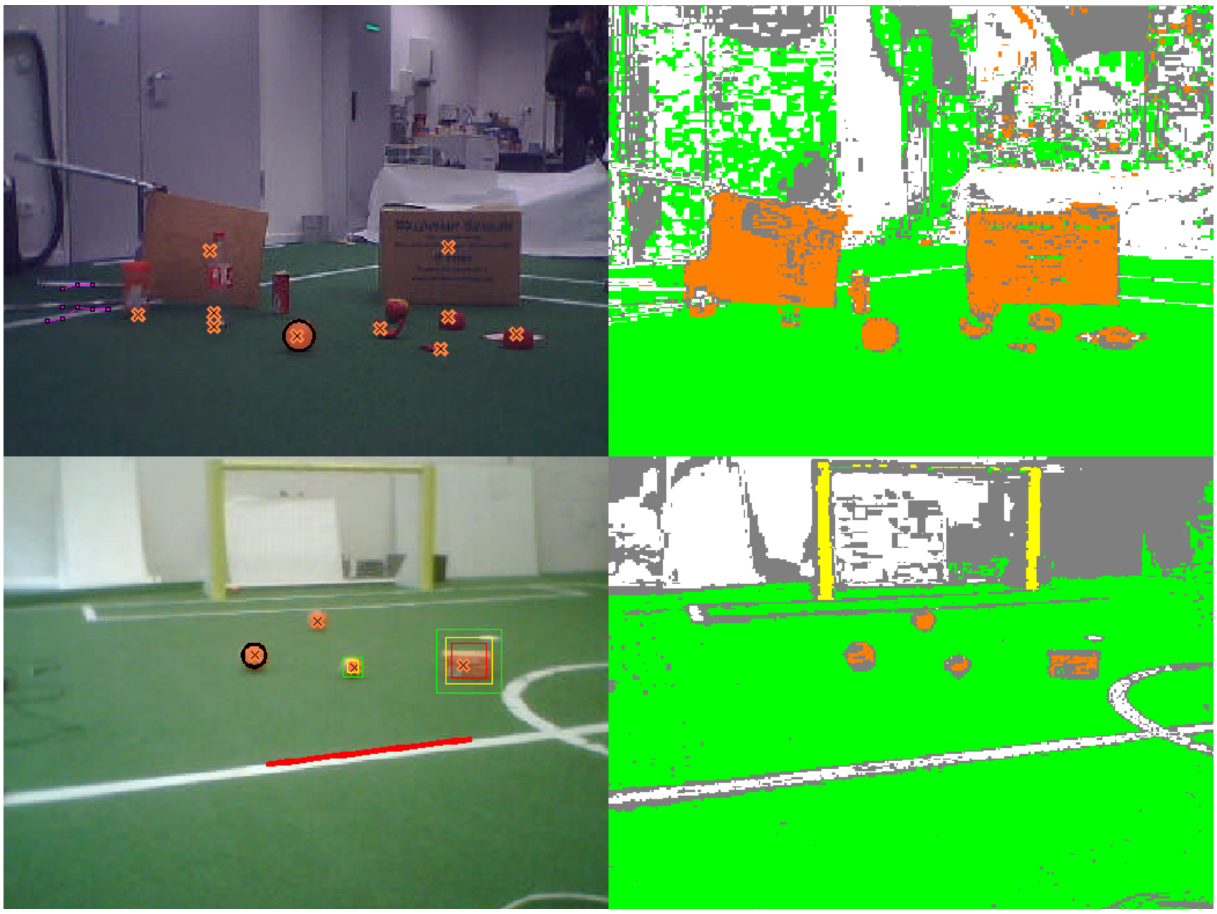


Figure 4.11: Test case for the BallPerceptor. The crosses mark *BallSpots*, the black circle marks the recognized Ball.

the search is stopped and the last orange point is saved. If the ball points found are considered valid, the correct center and the radius of the ball can be calculated. Therefore, we have to distinguish between a ball at the image border and a ball that is completely inside the image.

The center and the radius of a ball that is partially inside the image are calculated using the Levenberg-Marquardt method [1]. First, only hard edge points, i. e., points with a high contrast that are next to a green point are passed to the method. If there are not enough of those points or if the method fails, all hard edge points are taken into account. If that does not work either, all points that are not at the image border are used. If that still does not work out, the last attempt is to use all ball points.

For a ball that is completely inside the image the center and radius can be calculated by taking the two points of the ball which are farthest away from each other. The half of the distance is the radius of the ball and the center of the ball is the average of the x and y coordinates.

If the center of the ball is below the horizon, the offset to the ball is calculated based on its size in the image (for far balls) or its bearing (for close balls), resulting in the world coordinates relative to the robot. Due to this information we can project the ball back to the image. This results in a means for calculating the validity. The validity is calculated from the eccentricity, the projection factor and the quality of the edge points. A *BallSpot* is considered as possible ball if the validity is bigger than a certain threshold, it is considered not to be a ball if the validity is smaller than another threshold. If the validity is between the two thresholds we run another test to determine whether it could be a ball or not.

In this last test we analyze the color classes in the calculated ball and its surrounding area.

For this test we calculate three regions for the *BallSpot*. The first region is the biggest square that fits into the calculated ball. The second region is the smallest square bigger than the complete ball without all points inside the first region. The last region is a square with an edge length of $\sqrt{2} \cdot diameter$ and center point on the center of the ball without any points inside the two other regions (cf. Fig. 4.11). For each region the color classes of the points inside this region are counted.

The counted values are compared to configured thresholds for the first and the third region[1]. For example we expect at least 90 percent of orange points in the first region, but we only allow 2 percent of orange points in the third region. The result of this comparisons is a value between 0 and 1 for each region. These values describe the likelihood that the corresponding region could be part of a real ball and its surrounding area.

Now we know definitely whether a *BallSpot* can be considered as a ball or not. If it is a ball it is added to the list of *ExtendedBallPercepts*. When all *BallSpots* are analyzed, the one with the highest validity is stored in the representation *BallPercept*. Our tests showed that this approach works very well and distinguishes between balls and other objects (cf. Fig. 4.11). that is used by the ParticleFilterBallLocator to provide the *BallModel*. For more information see Sect. 4.2.2.

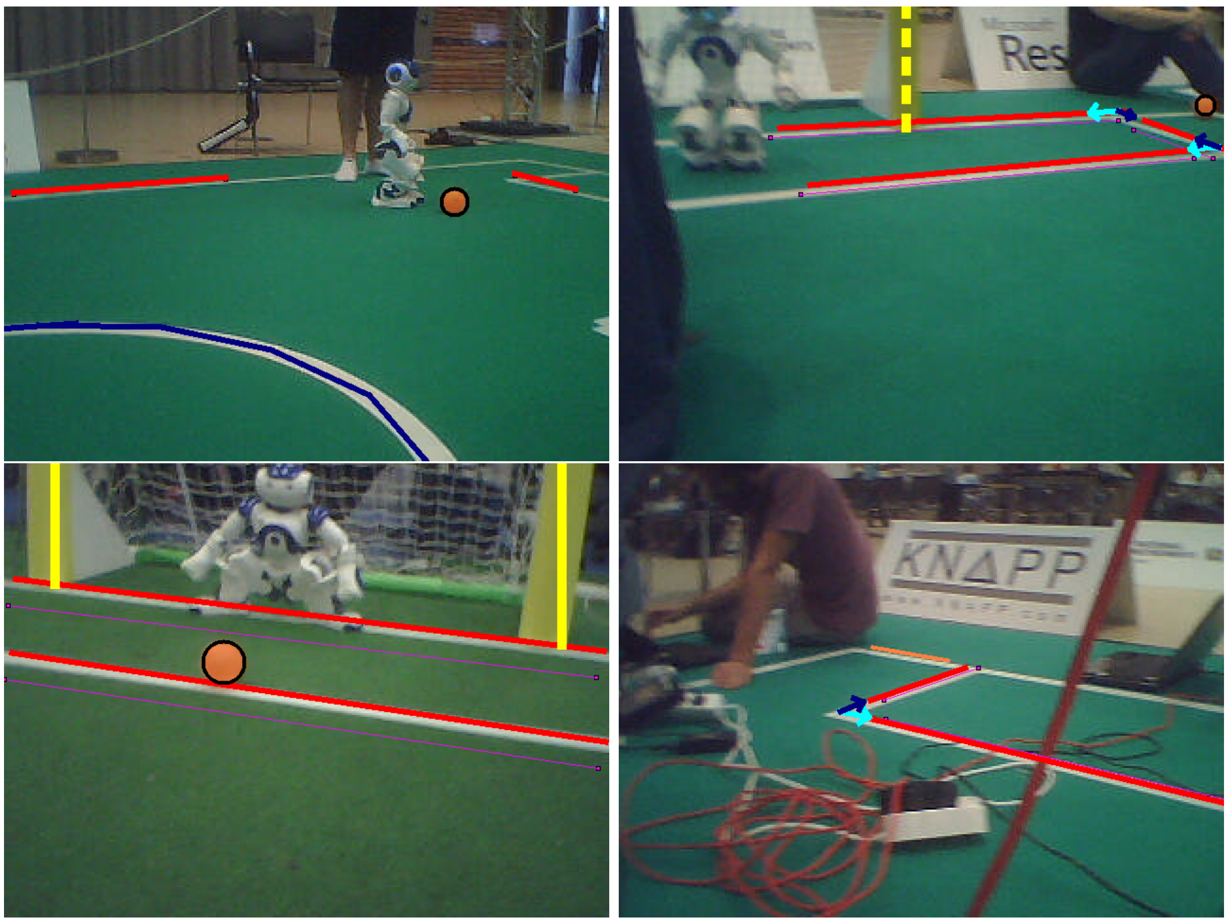


Figure 4.12: Some typical B-Human/RoboCup vision frames

[1]Currently the second region is ignored.

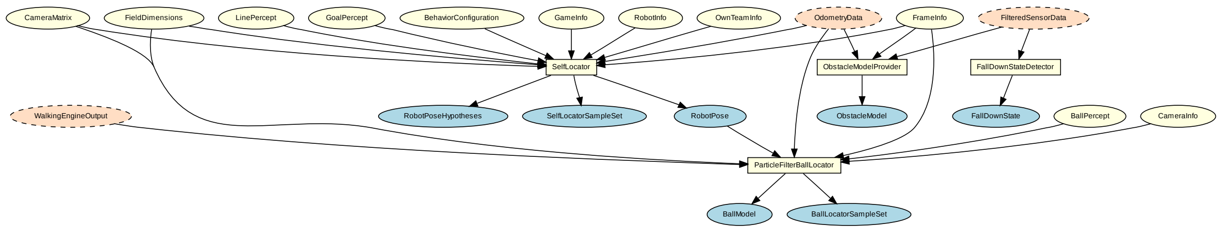


Figure 4.13: Modeling module graph

4.2 Modeling

To compute an estimate of the world state – including the robot's position, the ball's position and velocity, and the presence of obstacles – given the noisy and incomplete data provided by the perception layer, a set of modeling modules is necessary. The modules and their dependencies are depicted in Fig. 4.13.

4.2.1 Self-Localization

For self-localization, B-Human uses a particle filter based on the Monte Carlo method [4] as it is a proven approach to provide accurate results in such an environment [22]. Additionally, it is able to deal with the kidnapped robot problem that often occurs in RoboCup scenarios. For a faster reestablishment of a reasonable position estimate after a kidnapping, the *Augmented MCL* approach by [7] has been implemented. A comprehensive description of our general state estimation implementation – applied to a Humanoid League scenario – is given in [13].

The module providing the *RobotPose* that is a simple pose in 2-D is the SelfLocator. Earlier versions have already shown a reasonable performance during competitions in the Humanoid League in 2007 and 2008.

However, the previous implementation's sensor model has been designed for a Humanoid League field (as described in [19]) by taking only points on the field – lines as well as the base lines of the colored goal backgrounds that do not exist in the SPL – instead of more complex perceptions into account. B-Human's current vision system (cf. Sect. 4.1.3) provides a variety of perceptions that have all been integrated into the sensor model: goal posts (ambiguous as well as unambiguous ones), line segments (of which only the endpoints are matched to the field model), line crossings (of three different types: L, T, and X), and the center circle. During each sensor update of the particle filter, always a fixed number (currently six) of randomly selected perceptions is used, independently of the total number of perceptions; only goal post percepts are always preferred over other ones.

For a precise localization near a goal, it is not only necessary to perceive the goal posts – which are rarely seen to a utilizable extent – but also to avoid confusing the goal net with field lines. Therefore, the SelfLocator has access to a precomputed look-up table which provides the maximum valid distance to a field line for a given sample pose. As all false positives (given a robot position inside the field) resulting from the goal net lie beyond the goal line, this is an effective way of excluding them from the sensor update.

One other change is the integration of a new approach for computing the robot's pose from the sample set. Through the process of sensor resetting [17] (cf. Fig. 4.14b) to overcome kidnapping problems, the probability distribution of the sample set often becomes multimodal (cf. Fig. 4.14a). Our approach is capable of robustly and computationally efficiently tracking different sample clusters without any discretization drawbacks as, e. g., in the popular binning

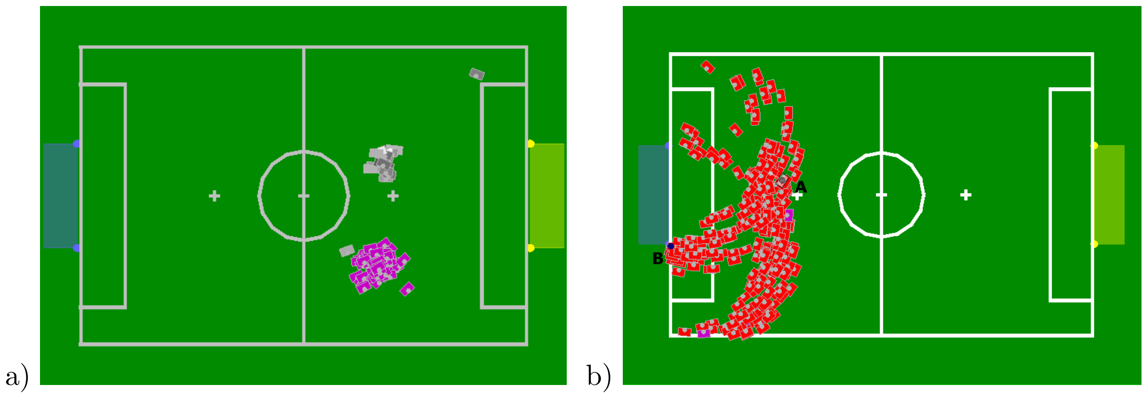


Figure 4.14: a) Multi modal probability distribution: The samples have clustered at two different positions. The cluster selected for pose computation is drawn in magenta. b) Sensor resetting: A robot standing at position A observes the non-unique goalpost at position B. The red boxes denote a sample set that has been fully generated using this single observation. All samples are situated on circles around the two possible goal posts. The different distances to the posts are a result of the sensor model's uncertainty.

.

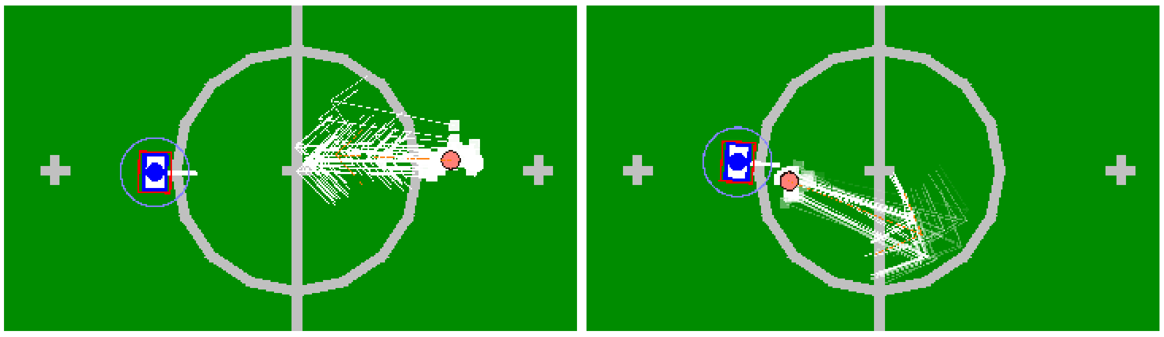


Figure 4.15: Ball samples before (left figure) and after (right figure) bouncing off the robot's shape. The orange circle and arrow show the ball model computed from the probability distribution. The circle around the robot denotes the area in which collision checks are performed, the red rectangle shows the shape the relevant samples are collided with.

approach. A detailed description is given in [15].

4.2.2 Ball Tracking

Estimating the ball's velocity as well as filtering its position is also realized via a particle filter similar to the one used for self-localization. A detailed description is also given in [13]. A probability distribution of a ball in motion is shown in Fig. 4.15. By using the perception described in Sect. 4.1.3.5, the module ParticleFilterBallLocator computes the *BallModel*.

A previous version of this module has already been used by the BreDoBrothers in 2008 and B-Human's Humanoid team in 2007 and 2008 [19]. The most significant improvement in 2009 has been the addition of a model for collisions between ball and robot, as depicted in Fig. 4.15. To be computationally inexpensive, this model is very simplistic and thus reduces the robot's feet to one simple rectangle. Its main purpose is to avoid the ball model traversing the robot during phases in which the robot does not look at the ball. This is especially useful for dribbling behaviors as well as for dealing with some cases of wrong velocity estimates (caused by erroneous measurements) of close balls.

4.2.3 Obstacle Model

Since the vision system is not yet able to robustly detect other robots, the only perception used for obstacle avoidance is the ultrasonic measurement. The possibility of measuring false positives as well as the sensor's rough coverage of the environment in combination with its low update rate lead to development of the ObstacleModelProvider.

To compute a robust model, all ultrasonic measurements are added to a grid (cf. Fig. 4.16b) which rasterizes the robot's local environment. The grid currently used has a resolution of 40×40 cells with each cell having a size of $60mm \times 60mm$. Each cell stores a number n_o of positive obstacle measurements. When receiving a new sensor measurement, the counter n_o of all cells on the front border of the measurement cone becomes increased. The space within this range is assumed to be empty, thus n_o of all cells within the cone becomes decreased. The projection of the cone to the floor is simplified to a triangle (cf. Fig. 4.16a) instead of a sector of a circle. This might cause a model of slightly closer obstacles but can – especially given the sensor noise and the rough rasterization – be neglected in general. To become considered as an occupied cell and thus part of an obstacle, n_o needs to reach a configurable threshold (currently 3). Thereby, single false positives cannot affect the model and the robot does not avoid any nonexistent obstacles. This turned out to be crucial for an efficient playing behavior.

The robot's motion is incorporated by odometry offsets to previous execution cycles. Rotations can be handled efficiently since no grid cells need to be moved, only the angular offset used in the transformations between the robot's and the grid's coordinate systems becomes changed. Translational movements become accumulated until the robot has moved into a different cell. Then the whole grid needs to be shifted. Over time, obstacle cells might leave the sensors' scope and the odometry accumulates a significant error. To avoid keeping any old and inconsistent information, all cell counters become decreased in regular time intervals (currently $3s$) after their last change.

The module's output, the *ObstacleModel*, contains the distances to the closest obstacles in four different sectors, as depicted in Fig. 4.16a. This quite simple representation has been chosen after consultation with the behavior implementers to avoid a specification of circumstantial obstacle avoidance behaviors. Of course, based upon the data contained in the grid, alternative representations would be easy to compute, e. g. angle and distance to closest obstacle or the angle to the next free corridor.

4.2.4 Detecting a Fall

To start an appropriate get-up motion after a fall, the orientation of the robot's body needs to be computed from the measurements of the accelerometers. This task is realized by the FallDownStateDetector that provides the *FallDownState*. By averaging the noisy sensor data, the module robustly detects whether the robot is in an upright position, lying on its front, its back, its left, or its right side. The latter two cases appear to be highly unlikely but are not impossible.

4.2.5 Ground Truth

As source for ground truth data, providing *RobotPose* and *BallModel*, a global tracking system is used. For this purpose, a unique marker is fixed on the robot's head (cf. Fig. 4.17) and tracked by a camera hanging above the field. For this purpose, the standard vision system of the RoboCup Small Size League [25] is used. The system provides the position as well as the rotation (that is fused with the robot's head rotation) of the robot on the field.

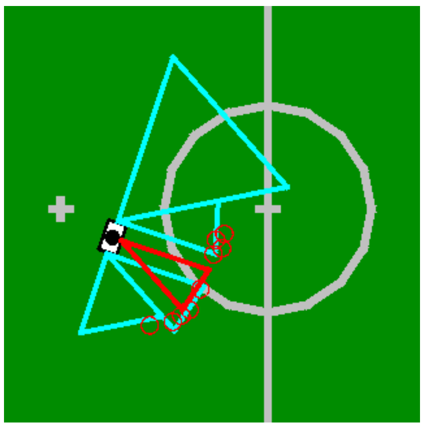
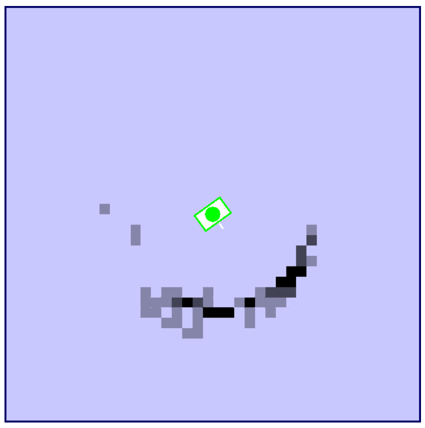
a)  b) 

Figure 4.16: Modeling obstacle information: a) The *ObstacleModel* drawn as cyan polygons relative to the robot. The red circles show the currently occupied cells, the red triangle the current ultrasonic measurement. b) The local occupancy grid of the situation shown in a). The cells' color depends on their counter n_o.

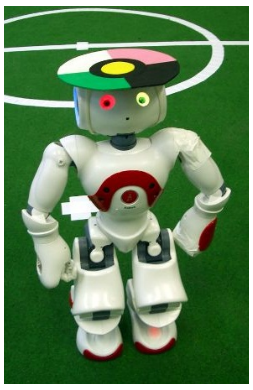
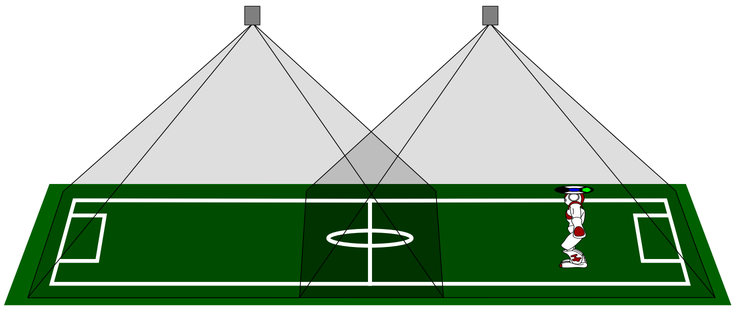
a)  b) 

Figure 4.17: a) A Nao robot equipped with a colored pattern for global tracking. b) Two cameras mounted above the field with overlapping images.

In its default configuration the standard vision system transmits UDP/IP packets to the multicast address 224.5.23.2. The robot and the vision system have to be configured to use the correct network device for this address.

Currently, there is no filtering of the marker pose, which causes instability of the ground truth robot pose in the area near the center line where both cameras provide the position of the marker.

The command *mr GroundTruthRobotPose SSLVision* will enable the SSLVision module to provide the *GroundTruthRobotPose*. The estimated *RobotPose* can still be calculated by the SelfLocator, both representations can be streamed to a connected SimRobot instance or stored in a log file.

Chapter 5

Motion

The *Motion* process comprises the two essential task *Sensing* (cf. Sect. 5.1) and *Motion Control* (cf. Sect. 5.2). *Sensing* contains all modules that are responsible for preprocessing sensor data. Some of the results of *Sensing* are forwarded to the *Cognition* process, but they are also used within the *Motion* process itself for the generation of walking motions. Therefore, the *Sensing* part must be executed before *Motion Control*. Figure 5.1 shows all modules running in the *Motion* process and the representations they provide.

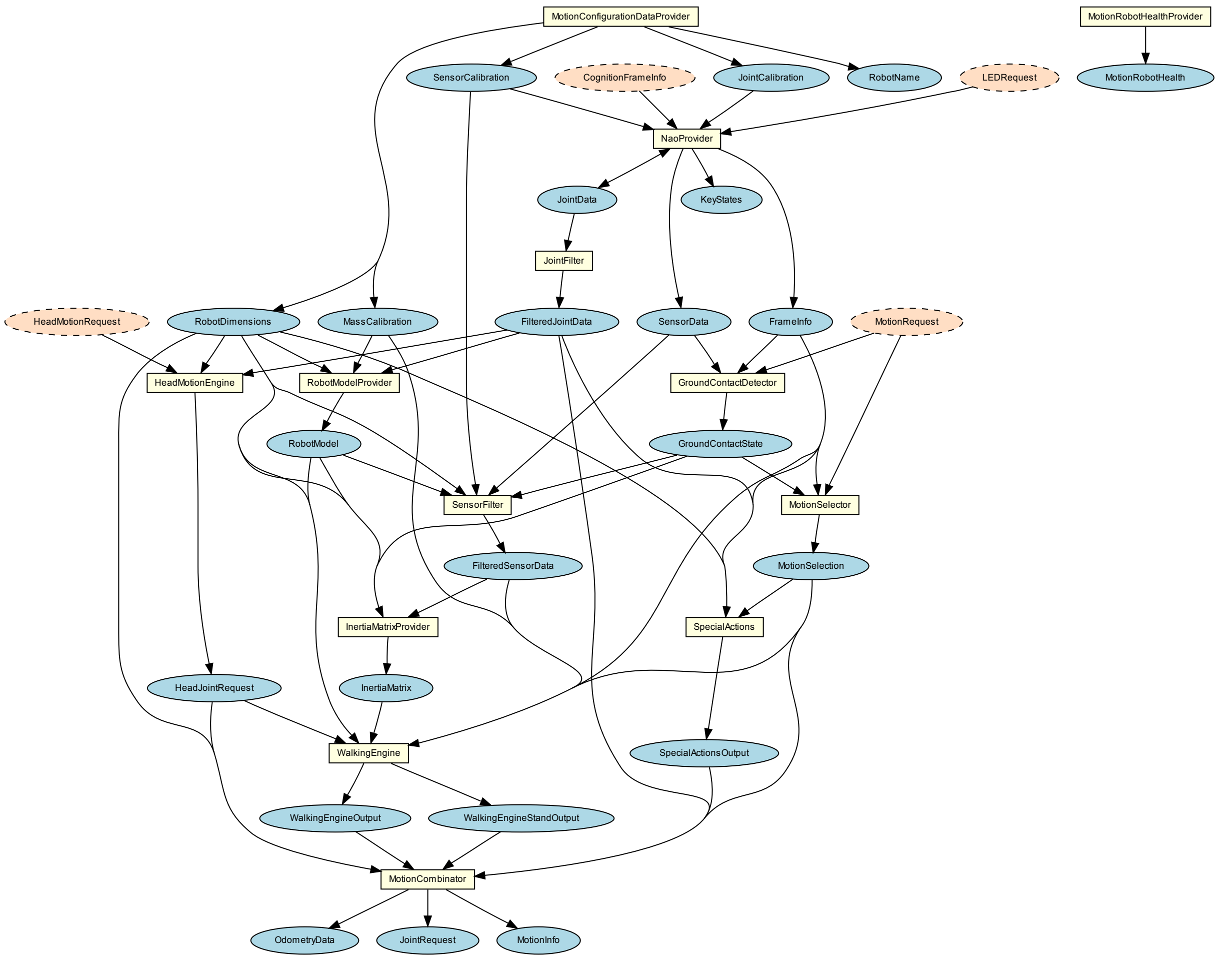


Figure 5.1: All modules and representations in the process *Motion*

5.1 Sensing

An important task of the motion process is to retrieve the measurements of all sensors of the Nao and to preprocess them. The Nao has three acceleration sensors (for the x-, y- and z-direction), two gyroscopes (for the rotation around the x- and y-axes), a sensor for the battery level, eight force sensing resistors, sensors for the load, temperature, and angle of each joint, and an ultrasonic sensor with four different measurement modes. Furthermore, the inertia board of the Nao calculates a crude approximation of the angle of the robot torso relative to the ground that is also accessible in the same way as the other sensors. The NaoProvider receives all sensor values, adds a calibration for each sensor and provides them as *SensorData* and *JointData*. The next step is to make sure that these sensor values do not contain unexpectedly large or small values, which may require a special treatment in other modules. Therefore, the *JointData* passes through the JointFilter module that provides the representation *FilteredJointData* (cf. Sect. 5.1.1) and the *SensorData* passes the SensorFilter module that provides the representation *FilteredSensorData* (cf. Sect. 5.1.4). The module SensorFilter also requires the *GroundContactState*, determined from the unfiltered *SensorData* (cf. Sect. 5.1.2), and the *RobotModel*, created from the representation *JointData* that contains the position of the robot limbs (cf. Sect. 5.1.3). Based on the *Filtered-SensorData* and the *RobotModel*, it is possible to calculate the *InertiaMatrix* that describes the position of the inertia board within the Nao (cf. Sect. 5.1.5).

5.1.1 Joint Data Filtering

The measured joint angles of the Nao are very accurate, so the main task of the JointFilter is to ensure that the *FilteredJointData* does not contain any values indicating missing sensors. Normally, this does not happen on the real robot (which was different on former platforms used by B-Human), but it can still occur with altered module configurations or while replaying log files that do not contain any sensor data.

5.1.2 Ground Contact Recognition

Since it may happen during official soccer matches that a robot is manually placed or it gets lifted because of a penalty, it is useful for several reasons (e. g. localization, behavior) for the robot to know whether it is standing or walking on the ground or not. It also comes in handy when the robot stops moving automatically after it got lifted, since it is much easier to place a standing robot on the field instead of a moving one. The *GroundContactState* that is actually a simple Boolean value indicating whether there is at least one foot on the ground, should not be confused with the *FallDownState* that indicates whether the robot is in a horizontal position.

Besides the SensorFilter, the GroundContactDetector is the only module that uses the unfiltered *SensorData*, since the result of the GroundContactDetector is required by the SensorFilter. So it is not possible for the GroundContactDetector to use the *FilteredSensorData*. Earlier approaches to recognize whether the robot stands upside on the ground were based on the load on the servomotors. This worked well with the lowered walk that was used at the GermanOpen 2009, but since the walk was replaced with a more economic version, the ground contact detection stopped working and was replaced with a method using the force sensing resistors in Nao's feet.

The force sensing resistors tend to lose their calibration very quickly, so they have to be re-calibrated regularly to use them with a static threshold for detecting the ground contact state. To avoid the need of the repeating re-calibrations, the GroundContactDetector observes the value range of the uncalibrated sensors for each foot while discarding particular high or low values.

This results in a range of values that allows to determine the *GroundContactState* with a percentage threshold. The *GroundContactState* does not change until most of the sensors of each foot give matching results for multiple frames.

5.1.3 Robot Model Generation

The *RobotModel* is a simplified representation of the robot. It contains information on the position of the limbs of the robot relative to its torso. Additionally it contains the position of the center of mass (*CoM*) of the robot.

The position of the limbs is represented as the rotation relative to the torso (*RotationMatrix*) and the position of the last hinge before the limb (in total represented as a homogenous transformation matrix). They are determined by consecutively computing the kinematic chain. The position of the joints is taken from the measured position (instead of the desired position). As the inverse kinematic, the implementation is customized for the Nao, i. e., the kinematic chain is not described by a general purpose convention such as Denavit-Hartenberg parameters to save computation time.

The *CoM* is computed by equation (5.1) with n = number of limbs, $\vec{r_i}$ = position of the center of mass of the i-th limb relative to the torso, and m_i = the mass of the i-th limb.

$$\vec{r}_{com} = \frac{\sum_{i=1}^{n} \vec{r_i} m_i}{\sum_{i=1}^{n} m_i} \tag{5.1}$$

$\vec{r_i}$ is computed using the transformation of the hinge and an offset of the *CoM* of a limb relative to its hinge. The offset and the mass of a limb is configurable (*masses.cfg*). The configuration provided was taken from the documentation of the Nao.

5.1.4 Sensor Data Filtering

The SensorFilter module is responsible for three different tasks. It drops broken inertial sensor values, it determines a calibration offset for the gyroscope sensor (for x and y), and it calculates an improved angle of the robot torso relative to the ground using an Unscented Kalman filter (UKF) [10]. The calculations base on *SensorData* and the results are provided as *FilteredSensorData*. Additionally, the module uses the *GroundContactState* for automatic gyroscope calibration and requires the *RobotModel* for estimating the torso's pitch and roll angle.

Dropping of broken inertial sensor values is necessary, because some sensor measurements received cannot be explained by the legal noise of the sensor. This malfunction occurs sporadically and affects most of the sensor values from the inertia board within a single frame (cf. Fig. 5.2). The broken frames are detected through comparing the difference of each value and their predecessor to a predefined threshold. If a broken frame is found that way, all sensor values from the inertial board are ignored. The broken acceleration sensor values are replaced with their predecessor and the broken gyroscope sensor values are not used for the angle estimation.

The gyroscope sensors are hard to calibrate, since their calibration offset depends of the sensor's temperature, which cannot be observed. The temperature changes slowly as long as the robot runs, so that it is necessary to redetermine the calibration offset constantly. Therefore, it is hypothesized that the robot has the same orientation at the beginning and ending of walking phases while all gyroscope values are collected during each phase. If the robot does not walk,

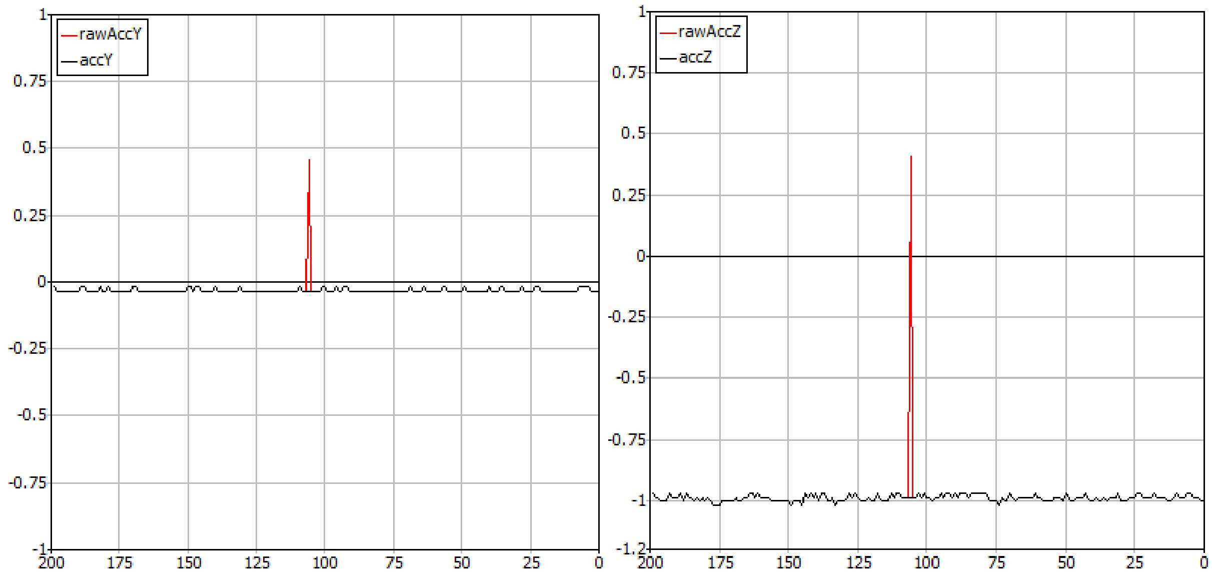


Figure 5.2: A typical broken inertia sensor data frame. The broken data was detected and replaced with its predecessor.

the gyroscope values are collected for one second instead. The average of the collected values is filtered through a simple one-dimensional Kalman filter and used as offset for the gyroscope sensor. The collection of gyroscope values is restricted to slow walking speeds and the ground contact state is used to avoid collecting gyroscope values in unstable situations.

The UKF estimates the orientation of the robot torso (cf. Fig. 5.3) that is represented as three-dimensional rotation matrix. The change of the rotation of the feet relative to the torso in each frame is used as process update. The sensor update is derived from the calibrated gyroscope values. Another sensor update is added from a crude absolute measurement realized under the assumption that the longer leg of the robot rests evenly on the ground as long as the robot stands almost upright. In cases in which this assumption is apparently incorrect, the acceleration sensor is used instead.

It is not only possible to get the orientation from the UKF, but also to get a "filtered" version of the gyroscope values from the change in orientation. These gyroscope values (including a calculated z-gyroscope value that is actually missing on the Nao) are written into the *FilteredSensorData* that contains corresponding values for all readings in the representation *SensorData*.

5.1.5 Inertia Matrix

The *InertiaMatrix* describes the transformation from the projection of the middle of both feet on the ground up to the position of the inertia board within the robot torso. The transformation is represented as *Pose3D*. Additionally, the *InertiaMatrix* contains the change of the position of the inertia board including odometry. Hence, the *InertiaMatrix* is used by the *WalkingEngine* for estimating the odometry offset. The *CameraMatrix* within the *Cognition* process is based on the *InertiaMatrix*.

To calculate the *InertiaMatrix* the vector of each foot from ground to the torso (f_l and f_r) is calculated by rotating the vector from torso to each foot (t_l and t_r), which can be calculated using the kinematic chain, according to the estimated rotation (cf. Sect. 5.1.4), represented as rotation matrix R.

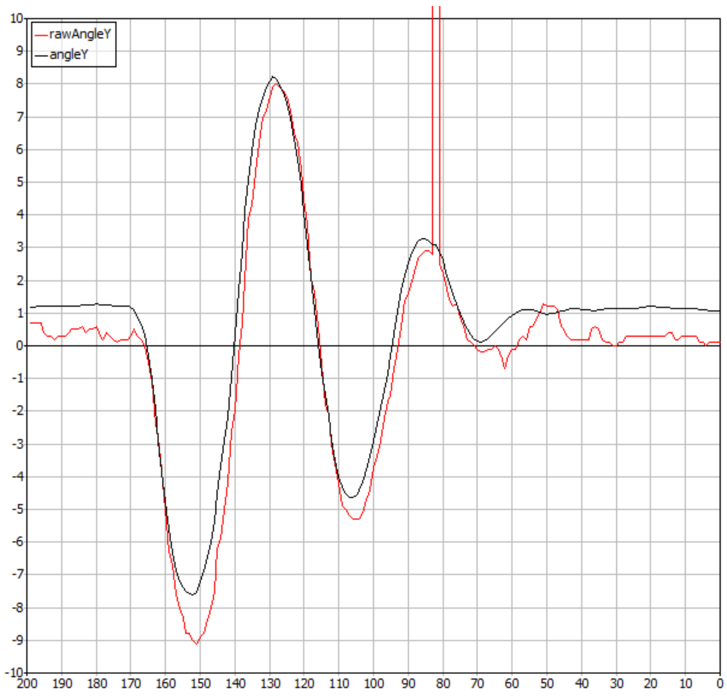


Figure 5.3: The difference between the estimated pitch angle *angleY* and the pitch angle *rawAngleY* provided by the inertia board.

$$f_l = -R \cdot t_l \tag{5.2}$$

$$f_r = -R \cdot t_r \tag{5.3}$$

The next step is to calculate the span between both feet (from left to right) by using f_l and f_r:

$$s = f_r - f_l \tag{5.4}$$

Now it is possible to calculate the translation part of the inertia matrix p_{im} by using the longer leg and the constant offset h that describes the position of the inertia board within the torso. The rotation part is already known since it is equal to R.

$$p_{im} = R \cdot h + \begin{cases} s/2 + f_l & \text{if } (f_l)_z > (f_r)_z \\ -s/2 + f_r & \text{otherwise} \end{cases} \tag{5.5}$$

The change of the position of the inertia position is determined by using the inverted inertia matrix of the previous frame and concatenating the odometry offset, which was calculated by using the change of the span between both feet and the change of the ground foot's rotation, and the new inertia matrix.

5.2 Motion Control

The B-Human motion control system generates all kinds of motions needed to play soccer with a robot. They are split into the different type of motions *Walking*, *Standing* and *SpecialActions*. The walking motion and a corresponding stand are dynamically generated by the WalkingEngine (cf. Sect. 5.2.3). All further motions that are created by static joint angle patterns, are provided by the module SpecialActions (cf. Sect. 5.2.4). Both modules generate joint angles. The WalkingEngine provides the *WalkingEngineOutput* and the *WalkingEngineStandOutput*. The module SpecialActions provides the *SpecialActionsOutput*. According to the *MotionRequest* the MotionSelector (cf. Sect. 5.2.1) calculates which motions to execute and how to interpolate between different motions while switching from one to another. This information is provided in the *MotionSelection*. If necessary, both modules calculate their joint data and the MotionCombinator (cf. Sect. 5.2.5) combines these according to the *MotionSeletion*. The MotionCombinator provides the *JointRequest*, *OdometryData*, and *MotionInfo*.

5.2.1 Motion Selection

While playing soccer with a robot it is necessary to execute different motions. To have smooth transitions between these motions, they are interpolated while switching from one to another.

The MotionSelector determines which motion to execute, taking into account not to interrupt a motion in an unstable situation. To achieve this, the modules WalkingEngine and SpecialActions that actually generate the motions both provide information about when it is possible to leave the motion they generate. In addition, some motions have a higher priority than others, e. g. stand-up motions. These are executed immediately and it is not possible to leave the motion before it finished.

If the *MotionRequest* requires switching from one motion to another and when it is possible to switch the motion, the MotionSelector calculates interpolation ratios for the motion currently executed and the target motion. Both motions are generated and the resulting joint angles are combined within the MotionCombinator (cf. Sect. 5.2.5).

5.2.2 Head Motions

Besides the motion of the body (arms and legs), the head motion is handled separately. In contrast to last year's code this task is encapsulated within a separate module HeadMotionEngine for two reasons: on one hand the walking engine manipulates the center of mass, therefore it is necessary to know the joint angles, and hence the mass distribution of the head *before* execution in order to compensate for head movements, on the other hand the module smoothens the head movement by limiting the speed as well as the maximum angular acceleration of the head.

The module takes the *HeadMotionRequest* generated within behavior control, and produces the *HeadJointRequest*.

5.2.3 Walking and Standing

An essential precondition for biped locomotion is to have a stable system for moving the Zero Moment Point (ZMP) from one foot to the other. Prior approaches by B-Human achieved this by using static trajectories for foot positions relative to the robot torso. The movement into the walking direction was also modeled with the shape and parameters of trajectories. This approach has the underlying disadvantage that it omits the position of the head, arms and legs, where some limbs are even completely independent from the walk itself. Since these limbs have an influence on the ZMP position, they should be taken into account. Especially, since the head of the Nao weighs about 400g, which is almost 10% of the total body weight.

The center of mass (CoM) of the body plays an important role in the generation of the movement for the alternating support leg. Hence, it makes sense to concentrate on the CoM position for the movement generation. The WalkingEngine does not use trajectories for foot positions relative to the torso any longer but for foot positions relative to the CoM instead. The foot positions relative to the CoM and the stance of the other body parts are used for determining foot positions relative to the torso that move the CoM to the desired point. The foot positions relative to the torso allow using inverse kinematics (cf. Sect. 5.2.3.4) for the generation of joint angles.

The *MotionSelection* provides the command to either stand, walk into a defined direction with defined speed, or walk to a position relative to the robot to the WalkingEngine. In all three cases, the joint angles for the stand are initially created as *WalkingEngineStandOutput*. Additionally, when the robot shall move, step sizes for the foot-shifting of the upcoming half-step are calculated continuously. The stand and the step sizes allow creating joint angles that are provided as *WalkingEngineOutput*. Besides the joint angles, the *WalkingEngineOutput* contains additional information (e. g. speed, speed limits, odometry offset, walking phase position, . . .) about the current walk.

Before the creation of walking motions starts, a stance (the stand) that will be used as basis for the foot joint angles at every time of the walking motion is required. This stance is called S and it is created from a foot position relative to the torso using inverse kinematics. S is mirror-symmetrically to x-z-plane of the robot's coordinate system. For the temporal orientation, a phase (p_{phase}) that runs from 0 to (excluding) 1 and repeats permanently, is used. The phase can be separated into two half-phases, in every half-phase of which another support leg is used and the other foot can be lifted up. The size of each half-step is determined at the beginning of each half-phase.

The shifting of the feet is separated into the "shift"- and "transfer"-phases. Both phases last one half-phase, run sequentially and are transposed for both feet. Within the "shift"-phase, the foot is lifted ("lift"-phase) and moved ("move"-phase) to another place on the ground, so that the foot is completely shifted at the end of the phase. The foot shifting is subtracted from the foot position within the "transfer"-phase and also subtracted from the other foot position until this foot is shifted (cf. Fig. 5.8). This alone creates a motion that already looks like a walking motion. But the desired CoM movement is still missing.

So the foot positions relative to the torso (p_{lRel} and p_{rRel}) are calculated as follows:

$$p_{lRel} = \begin{cases} o_l + s_{lLift} \cdot t_{lLift} + s_l \cdot t_{lMove} - s_r \cdot (1 - t_{lMove}) \cdot t_l & \text{if } p_{phase} < 0.5 \\ o_l + s_{lLift} \cdot t_{lLift} + s_l \cdot (1 - t_r) & \text{if } p_{phase} \geq 0.5 \end{cases} \tag{5.6}$$

$$p_{rRel} = \begin{cases} o_r + s_{rLift} \cdot t_{rLift} + s_r \cdot t_{rMove} - s_l \cdot (1 - t_{rMove}) \cdot t_r & \text{if } p_{phase} \geq 0.5 \\ o_r + s_{rLift} \cdot t_{rLift} + s_r \cdot (1 - t_l) & \text{if } p_{phase} < 0.5 \end{cases} \tag{5.7}$$

o_l and o_r are the foot origin positions. s_{lLift} and s_{rLift} are the total offsets used for lifting either the left or the right leg. s_l and s_r are the current step sizes. t_{lLift} and t_{rLift} are parameterized trajectories that are used for the foot lifting. They are parameterized with the beginning (x_l) and the duration (y_l) (cf. Fig. 5.4) of the "lift"-phase. t_{lMove} and t_{rMove} are used for adding the step sizes. They are parameterized with the beginning (x_m) and the duration (y_m) of the "move"-phase (cf. Fig. 5.5). t_l and t_r are shifted cosine shapes used for subtracting the step sizes (cf. Fig. 5.6). The trajectories are defined as follows:

$$t_{lLift} = \begin{cases} \frac{1 - cos((2 \cdot p_{phase} - x_l)/y_l \cdot 2\pi)}{2} & \text{if } 2 \cdot p_{phase} > x_l \wedge 2 \cdot p_{phase} < x_l + y_l \\ 0 & \text{otherwise} \end{cases} \tag{5.8}$$

$$t_{rLift} = \begin{cases} \frac{1 - cos((2 \cdot p_{phase} - 1 - x_l)/y_l \cdot 2\pi)}{2} & \text{if } 2 \cdot p_{phase} - 1 > x_l \wedge 2 \cdot p_{phase} - 1 < x_l + y_l \\ 0 & \text{otherwise} \end{cases} \tag{5.9}$$

$$t_{lMove} = \begin{cases} \frac{1-cos((2\cdot p_{phase}-x_m)/y_m\cdot\pi)}{2} & \text{if } 2\cdot p_{phase} > x_m \wedge 2\cdot p_{phase} < x_m + y_m \\ 1 & \text{if } 2\cdot p_{phase} \geq x_m + y_m \wedge 2\cdot p_{phase} < 1 \\ 0 & \text{otherwise} \end{cases} \tag{5.10}$$

$$t_{rMove} = \begin{cases} \frac{1-cos((2\cdot p_{phase}-1-x_m)/y_m\cdot\pi)}{2} & \text{if } 2\cdot p_{phase}-1 > x_m \wedge 2\cdot p_{phase}-1 < x_m + y_m \\ 1 & \text{if } 2\cdot p_{phase}-1 \geq x_m + y_m \wedge 2\cdot p_{phase}-1 < 1 \\ 0 & \text{otherwise} \end{cases} \tag{5.11}$$

$$t_l = \begin{cases} \frac{1-cos(2\cdot p_{phase}\cdot\pi)}{2} & \text{if } p_{phase} < 0.5 \\ 0 & \text{otherwise} \end{cases} \tag{5.12}$$

$$t_r = \begin{cases} \frac{1-cos(2\cdot p_{phase}\cdot\pi)}{2} & \text{if } p_{phase} \geq 0.5 \\ 0 & \text{otherwise} \end{cases} \tag{5.13}$$

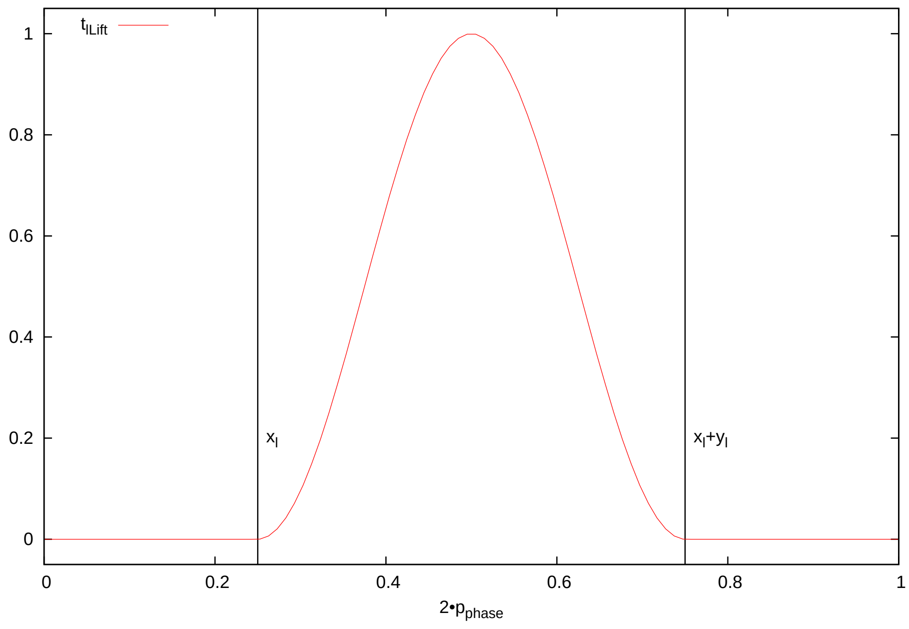


Figure 5.4: The trajectory t_{lLift} that is used for foot lifting. t_{rLift} is similar to t_{lLift} except that it is shifted one half-phase to the right.

The CoM movement has to be performed along the y-axis as well as in walking direction. The CoM is already moving in walking direction by foot shifting, but this CoM movement does not allow walking with a speed that meets our needs. Also, a CoM movement along the z-axis is useful. First of all, the foot positions relative to the CoM are determined by using the foot positions relative to the CoM of stance S and adding the value of a trajectory to the y-coordinate of these positions. Since the calculated foot positions are relative to the CoM, the rotation of the body that can be added with another trajectory should be considered for the calculation of the foot positions. The CoM movement in x-direction is calculated with the help of the step sizes of the currently performed half-steps.

If any kind of body and foot rotations and CoM-lifting along the z-axis are ignored, the desired foot positions relative to the CoM (p_{lCom} and p_{rCom}) can be calculated as follows:

$$p_{lCom} = \begin{cases} -c_S + s_s \cdot t_{com} - \frac{s_l\cdot t_{lin} - s_r\cdot(1-t_{lin})}{2} + o_l & \text{if } p_{phase} < 0.5 \\ p_{rCom} - p_{rRel} + p_{lRel} & \text{otherwise} \end{cases} \tag{5.14}$$

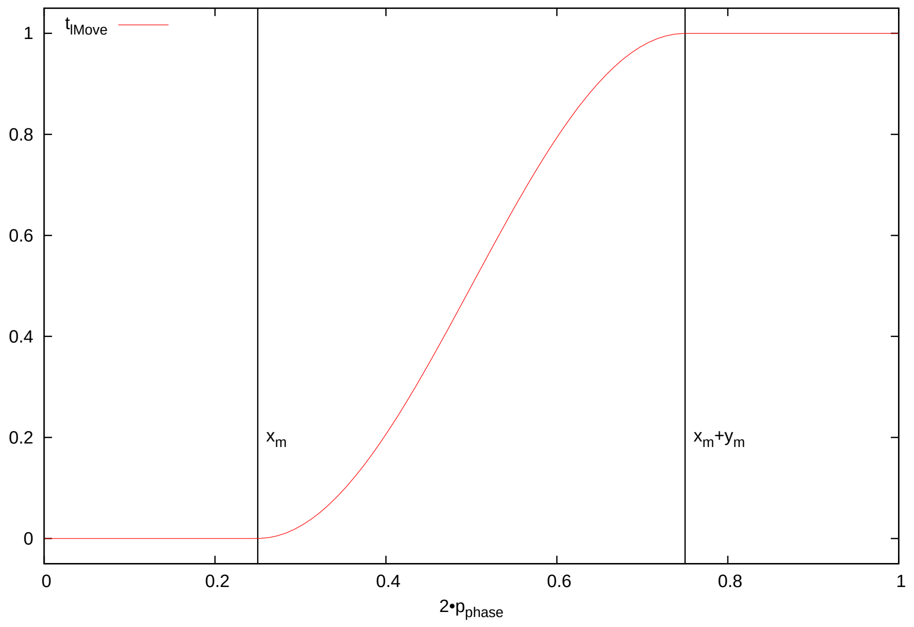


Figure 5.5: The trajectory t_{lMove} that is used for adding step sizes. t_{rMove} is similar to t_{lMove} except that it is shifted one half-phase to the right.

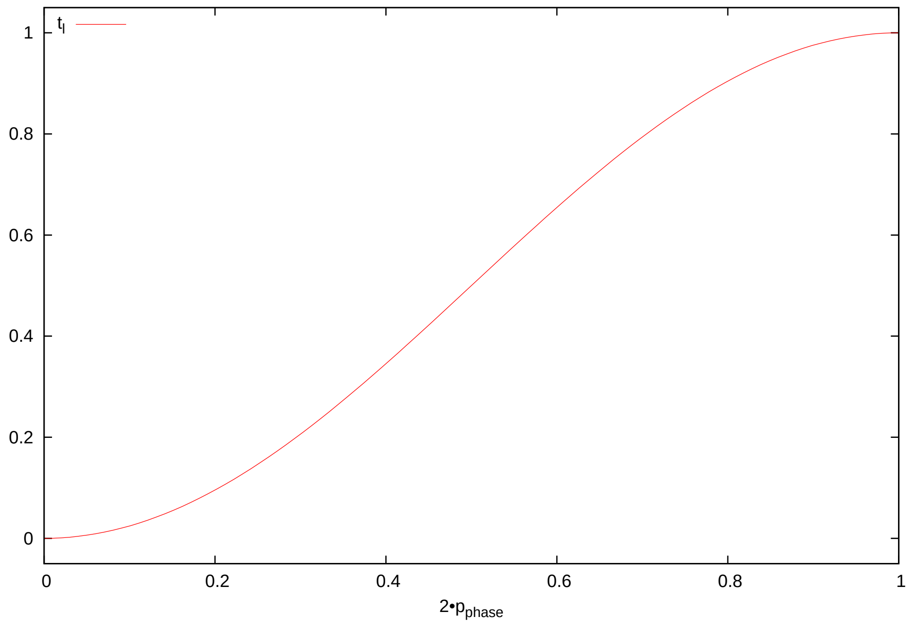


Figure 5.6: The trajectory t_l that is used for subtracting step sizes. t_r is similar to t_l except that it is shifted one half-phase to the right.

$$p_{rCom} = \begin{cases} -c_S + s_s \cdot t_{com} - \frac{s_r \cdot t_{lin} - s_l \cdot (1 - t_{lin})}{2} + o_r & \text{if } p_{phase} \geq 0.5 \\ p_{lCom} - p_{lRel} + p_{rRel} & \text{otherwise} \end{cases} \tag{5.15}$$

c_S is the offset to the CoM relative to the torso of the stance S. s_s is the vector that describes the amplitude of the CoM movement along the y-axis. t_{com} is the parameterized trajectory of the CoM movement. Therefore, a sine ($s(p)$), square root of sine ($r(p)$) and a linear ($l(p)$) component are merged according to the ratios x_c (for $s(p)$), y_c (for $r(p)$) and z_c (for $l(p)$) (cf.

Fig. 5.7). t_{com} is defined as:

$$t_{com} = \frac{x_c \cdot s(p_{phase}) + y_c \cdot r(p_{phase}) + z_c \cdot l(p_{phase})}{x_c + y_c + z_c} \tag{5.16}$$

$$s(p) = sin(2\pi \cdot p) \tag{5.17}$$

$$r(p) = \sqrt{|sin(2\pi \cdot p)|} \cdot sgn(sin(2\pi \cdot p)) \tag{5.18}$$

$$l(p) = \begin{cases} 4 \cdot p & \text{if } p < 0.25 \\ 2 - 4 \cdot p & \text{if } p \geq 0.25 \wedge p < 0.75 \\ 4 \cdot p - 4 & \text{if } p \geq 0.75 \end{cases} \tag{5.19}$$

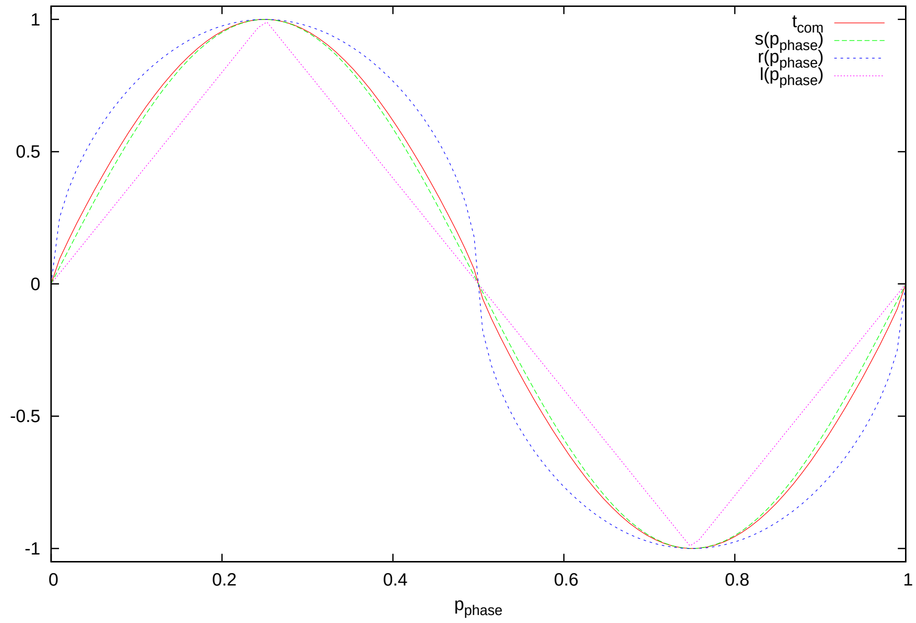


Figure 5.7: The trajectory t_{com} that is used for the CoM movement along the y-axis. It is a composition of $s(p_{phase})$, $r(p_{phase})$ and $l(p_{phase})$.

t_{lin} is a simple linear trajectory used for moving the CoM into the walking direction.

$$t_{lin} = \begin{cases} 2 \cdot p_{phase} & \text{if } p_{phase} < 0.5 \\ 2 \cdot p_{phase} - 1 & \text{if } p_{phase} \geq 0.5 \end{cases} \tag{5.20}$$

Based on the desired foot positions relative to the CoM another offset is calculated and added to the already calculated foot positions relative to the torso (p_{lRel} and p_{rRel}) to achieve the desired foot positions relative to the CoM with coordinates relative to the torso. For the calculation of the additional offset, the fact that the desired CoM position does not change that much within two cycles is exploited, because the offset that was determined in the previous frame is used first. So, given the current leg, arm and head stance and the offset of the previous frame, some foot positions relative to the CoM are determined. The difference between these foot positions and the desired foot positions is added to the old offset to get the new one. The resulting foot positions relative to the CoM are not precise, but the error is pretty small.

The additional offset (e_{new}) can be calculated using the old offset (e_{old}) as follows:

$$e_{new} = \frac{p_{lRel} - p_{lCom} + p_{rRel} - p_{rCom}}{2} - c_{e_{old}, p_{lRel}, p_{rRel}} \tag{5.21}$$

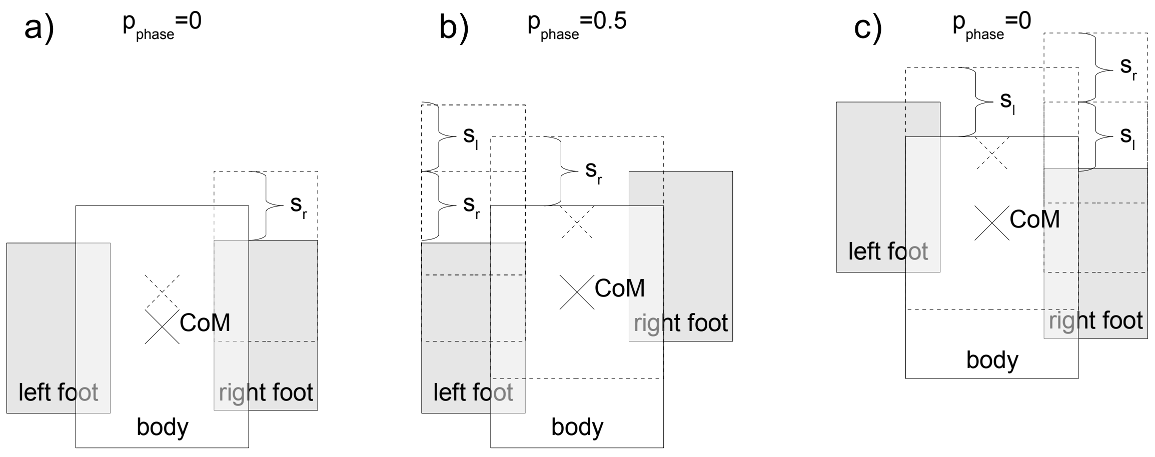


Figure 5.8: Illustration of the foot shifting within three half-phases from a top-down view. At first, the robot stands (*a)*) and then it walks two half-steps to the front (*b)* and *c)*). The support leg (left leg at *a)*) changes in *b)* and *c)*. Between *a)* and *b)*, the step size s_r is added to the right foot position. Between *b)* and *c)*, s_r is subtracted from the right and left foot positions, so that the body moves forwards. Additionally, the new offset $s_r + s_l$ is added to the left foot position. The visualized CoM is the desired CoM position, which differs a lot from the "center of body". So an additional offset has to be added to both foot positions, to move the CoM to the desired position.

$c_{e_{old},p_{lRel},p_{rRel}}$ is the CoM offset relative to the torso given the old offset, the new foot stance, and the stance of the other limbs.

Finally, $p_{lRel} - e_{new}$ and$p_{rRel} - e_{new}$ are the positions used for creating the joint angles, because:

$$p_{lRel} - c_{e_{new},p_{lRel},p_{rRel}} - e_{new} \approx p_{lCom} \quad (5.22)$$

$$p_{rRel} - c_{e_{new},p_{lRel},p_{rRel}} - e_{new} \approx p_{rCom} \quad (5.23)$$

Although former B-Human walking engines [19] could only be invoked by walking speed and direction, the current WalkingEngine allows to set a walking target by using the "target" entry within the *MotionRequest* representation. If a target was set that differs from zero, the WalkingEngine tries to walk as fast as possible to the target specified until it is reached. Therefore, the odometry of the steps executed are reduced form the walking target and the step sizes are calculated according to the distance of the target. The target can be changed at any time while the robot is walking. When the target shall not change any longer and the target that was set at last shall be reached, "target" within the *MotionRequest* representation has to be set to zero. To prematurely stop walking, the target can be replaced with another one or any speed can be set.

5.2.3.1 Balancing

To react on unexpected events and for stabilizing the walk in general, balancing is required. Therefore, the WalkingEngine supports different balancing methods. Most of them are simple p-controllers. The error that is used as input for the controllers is solely determined from the angles of the torso that were estimated by the SensorFilter module (cf. Sect. 5.1.4), and expected angles. The delay between the measured and the desired CoM positions is taken into account by buffering and restoring the desired foot positions and body rotation angles. The different kinds of balancing are:

CoM Balancing. The CoM balancer works by adding an offset to the desired foot positions (p_{lCom} and p_{rCom}). Therefore, the error between the measured and desired foot positions has to be determined, so that the controller can add an offset to these positions according to the determined error. The error is calculated by taking the difference between the desired foot positions and the same positions rotated according to rotation error.

Rotation Balancing. Besides **CoM Balancing**, it is also possible to balance with the body and/or foot rotation. Therefore, angles depending on the rotation error can be added to the target foot rotations or can be used for calculating the target foot positions and rotations. This affects the CoM position, so that the offset computation that is used for moving the foot positions relative to the CoM might compensate the balancing impact. So this kind of balancing probably makes only sense when it is combined with **CoM Balancing**.

Phase Balancing. Balancing by modifying the walking phase is another possibility. Therefore, the measured x-angle, the expected x-angle and the current walking phase position are used to determine a measured phase position. The measured position allows calculating the error between the measured and actual phase position. This error can be used for adding an offset to the current phase position. When the phase position changes in this manner, the buffered desired foot positions and body rotations have to be adjusted, because the modified phase position affects the desired foot positions and body rotations of the past.

Step-Size Balancing. Step-size balancing is the last supported kind of balancing. It works by increasing or decreasing the step size during the execution of half-steps according to the foot position error that was already used for **CoM Balancing**. Applied step-size balancing has the disadvantage that the predicted odometry offset becomes imprecise. This is a serious drawback, when the robot tries to precisely walk behind the ball in order to prepare a shot. Therefore, this kind of balancing can be switch off in the *MotionRequest* by activating the "pedantic" walking mode.

5.2.3.2 Parameters

The WalkingEngine comes along with a lot of required parameters that are read from the file *walking.cfg*. They can also be modified by using the commands *get*/*set* for the *module:WalkingEngine:parameters* modifier. This section gives a brief overview of these parameters. Parameters describing lengths are interpreted in mm, speeds in $\frac{mm}{s}$ and rotations in radians.

stepDuration. The duration of a step (two half-steps) in ms.

stepLift. A height added to each half-step size.

footOrigin. Coordinates relative to the torso for the left and right feet that are used as origin for standing and walking.

footOriginRotation. Rotations added to each foot.

footCenter. The position of the foot center relative to the ankle point. Changing this value may affect the calculation of rotation movements. However, leaving these values zero already works quite well.

footRotation. An additional foot rotation around the x-axis according to the sine of the walking phase.

legHardness. The hardness for all leg servomotors (except `legLeft5` and `legRight5`) that which overwrites the default values declared in *jointHardness.cfg*.

leg5Hardness. The hardness for `legLeft5` and `legRight5`. Experiments showed that reducing this value does not lead to disadvantages in any cases. [11]

bodyOriginTilt. The desired rotation of the body around the y-axis.

bodyRotation. A rotation around the y-axis added to the body according to the sine of the walking phase.

armOriginRotation. The angles for `armLeft0` and `armRight0` (y) and `armLeft1` and `armRight2` (x).

armMoveRotation. Values that are added to the arm rotation according to the performed steps.

coMShift. The amplitude of the CoM along the y-axis.

coMLift. A height added to the desired CoM position to generate a rising CoM depending on the lifting of the foot.

coMShiftShape. The parameters x_c, y_c and z_c of the trajectory used for the CoM position along the y-axis.

coMSpeedOffset. An additional offset as factor of the step size added to the desired CoM position.

coMAccelerationOffset. Not implemented.

coMTransferRatio. The ratio that defines how far the CoM has moved into the walking direction in the moment the support leg changes. The value 0.5 causes the smoothest CoM movement into the walking direction at constant speed.

liftOffset. The height used for foot lifting.

liftPhases. The beginning (x_l) and duration (y_l) of the “lift”-phase as ratios of the half-phase.

movePhases. The beginning (x_m) and duration (y_l) of the “move”-phase in ratios of the half-phase.

maxSpeed. The maximum speed allowed. Speeds exceeding this maximum are cropped down.

maxBackwardSpeed. The maximum speed allowed in x-direction used for backwards speeds.

maxDashSpeed. The maximum speed allowed for speed vectors with only a single component.

maxSpeedChanged. The maximum of acceleration performed in $\frac{mm}{s^2}$. Deceleration is not limited.

maxSpeedEllipsoidClipping. The kind of speed clipping used. If this value is $true$, the requested speed is cropped through the three-dimensional ellipse defined by the maximum speeds, preserving the ratio of speed components. If it is $false$, a rectangle is used instead.

measuredCoMGain. This parameter allows modeling the difference between the desired and the actual body rotation around the x-axis while walking. The measured rotation is scaled with this value before it is used for determining the rotation error.

measuredCoMDelay. The delay between setting and measuring servomotor angles. It is about 80 ms on the Nao.

balanceStep. A factor for the step-size controller.

balanceNextStep. A factor for adopting the step size offset, added through the step-size controller, to the size of the next half-step.

balanceCoM. The p-factors for the CoM controller.

balancePhaseAcceleration. The p-factor for positive phase controlling.

balancePhaseDeceleration. The p-factor for negative phase controlling.

balanceBodyRotation. The p-factors for the body rotation controller.

balanceFootRotation. The p-factors for the foot rotation controller.

balanceArm. The p-factors for some funny looking controlling with the arm stance.

balanceMaxInstability. The maximum instability allowed. If this value is exceeded, the WalkingEngine stops walking and enforces a stand.

balanceMaxInstabilityStandTime. The duration in ms declaring, how long the WalkingEngine stops walking when the **balanceMaxInstability** was exceeded.

balanceAirFoot. Declares whether the foot rotation balancing is applied on the lifted foot.

balancePhaseOld. Another p-factor for a phase controller that uses another (older) method for calculating the error. The other phase controller is inactive if this value is nonzero.

odometryScale. Some factors for scaling the calculated odometry offset.

odometryUseInertiaMatrix. Declares whether the odometry offset is calculated by using the *InertiaMatrix* or by using the desired step size. The former method should be more accurate.

5.2.3.3 Parameter Optimization

Since it may require a lot of effort to find good parameters, the application of optimization algorithms for this purpose is very popular. Therefore, the B-Human WalkingEngine module supports basic parameter optimization using a simple *Particle Swarm Optimization* (PSO) algorithm [3]. The `WalkingEngine::initOptimization()` method within the WalkingEngine source code initializes the parameter vector including minimum and maximum values. The `WalkingEngine::updateOptimization()` method applies suggested parameter values that are produced by the PSO algorithm. The optimization can be started by using the *module:WalkingEngine:optimize* debug response. When this debug response is active, the WalkingEngine adopts a set of new parameters (using the `updateOptimization` method) each time a new walking session is invoked. After 5 seconds, the WalkingEngine stops walking and evaluates the walking parameters last used according to the instability that was measured using the sum of the rotation errors within the previous seconds. When the walk stops, the robot can be repositioned on the field. Because of the ground contact detection, the robot will not restart walking until it stands on the ground again. Besides the *optimize* debug response, another debug response called *module:WalkingEngine:continuousOptimize* exists, which causes a similar behavior, but instead of stopping walking, the walking parameters are changed on the fly.

To optimize parameters for a simple forward walk, activate the debug response and set the *MotionRequest* to the desired speed. For instance:

```
dr module:WalkingEngine:optimize
set representation:MotionRequest { motion = walk; specialActionRequest = {
    specialAction = playDead; mirror = false; }; walkRequest = { speed = {
    rotation = 0; translation = { x = 200; y = 0; }; }; target = { rotation = 0;
    translation = { x = 0; y = 0; }; }; pedantic = false; }; }
```

The best evaluated parameter vector of the optimization can be requested by using the following command:

```
get module:WalkingEngine:bestParameters
```

But doing an effective optimization may require some modifications of the WalkingEngine to choose some parameters and their value range and to customize the duration of each run.

The parameter optimization was used to find optimal parameters for the alternating support legs that are mainly effected by **bodyRotation** and **coMShift**. So these two parameters were optimized while the robot was walking on the spot. The results, which were different for each robot, were averaged and rounded to keep a single set of parameters. This resulting parameter set (the values in *walking.cfg*) was used at RoboCup 2009.

5.2.3.4 Inverse Kinematic

Solving the inverse kinematics problem analytically for the Nao is not straightforward because of two special circumstances:

- The axes of the hip yaw joints are rotated by 45 degrees.
- These joints are also mechanically connected among both legs, i. e., they are driven by a single servo motor.

The target of the feet is given as homogenous transformation matrices, i. e., matrices containing the rotation and the translation of the foot in the coordinate system of the torso. To explain our solution we use the following convention: A transformation matrix that transforms a point p_A given in coordinates of coordinate system A to the same point p_B in coordinate system B is named $A2B$, so that $p_B = A2B \cdot p_A$. Hence the transformation matrix that describes the foot position relative to the torso is $Foot2Torso$ that is given as input. The coordinate frames used can are depicted in Fig. 5.2.3.4.

The position is given relative to the torso, i. e., more specifically relative to the center point between the intersection points of the axes of the hip joints. So first of all the position relative to the hip is needed[1]. This is a simple translation along the y-axis[2]

$$Foot2Hip = Trans_y\left(\frac{l_{dist}}{2}\right) \cdot Foot2Torso \tag{5.24}$$

[1]The computation is described for one leg. Of course, it can be applied to the other leg as well.

[2]The elementary homogenous transformation matrices for rotation and translation are noted as $Rot_{<axis>}(angle)$ resp. $Trans_{<axis>}(translation)$.

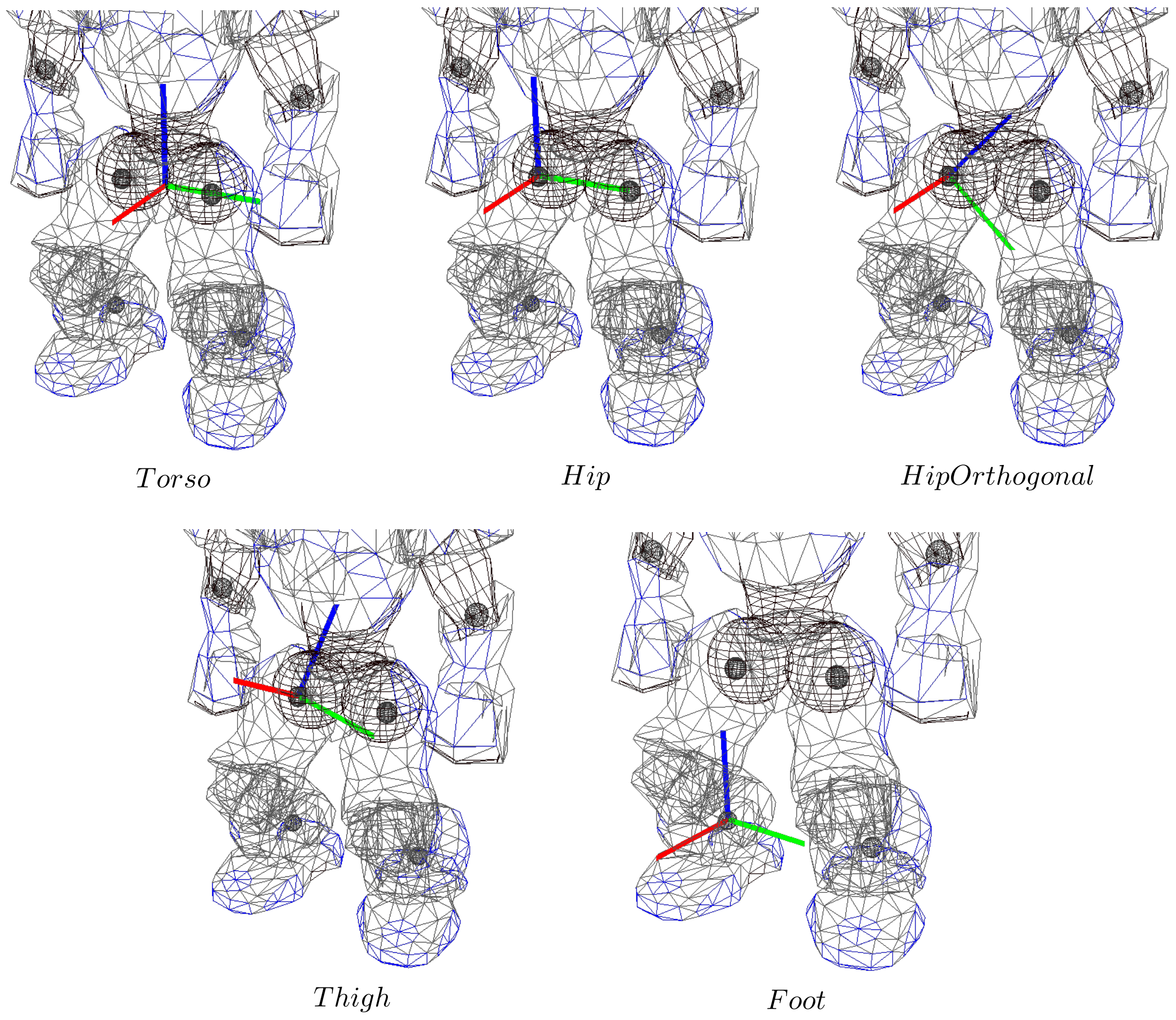


Figure 5.9: Visualization of coordinate frames used in the inverse kinematic. Red = x-axis, green = y-axis, blue = z-axis.

with l_{dist} = distance between legs. Now the first problem is solved by describing the position in a coordinate system rotated by 45 degrees, so that the axes of the hip joints can be seen as orthogonal. This is achieved by a rotation around the x-axis of the hip by 45 degrees or $\frac{\pi}{4}$ radians.

$$Foot2HipOrthogonal = Rot_x(\frac{\pi}{4}) \cdot Foot2Hip \tag{5.25}$$

Because of the nature of the kinematic chain, this transformation is inverted. Then the translational part of the transformation is solely determined by the last three joints and hence they can be computed directly.

$$HipOrthogonal2Foot = Foot2HipOrthogonal^{-1} \tag{5.26}$$

The limbs of the leg and the knee form a triangle, in which an edge equals the length of the translation vector of $HipOrthogonal2Foot$ (l_{trans}). Because all three edges of this triangle are known (the other two edges, the lengths of the limbs, are fix properties of the Nao) the angles of the triangle can be computed using the law of cosines (5.27). Knowing that the angle enclosed by the limbs corresponds to the knee joint, that joint angle is computed by equation (5.28).

$$c^2 = a^2 + b^2 - 2 \cdot a \cdot b \cdot \cos\gamma \tag{5.27}$$

$$\gamma = \arccos \frac{{l_{upperLeg}}^2 + {l_{lowerLeg}}^2 - {l_{trans}}^2}{2 \cdot l_{upperLeg} \cdot l_{lowerLeg}} \tag{5.28}$$

Because γ represents an interior angle and the knee joint is being streched in the zero-position, the resulting angle is computed by

$$\delta_{knee} = \pi - \gamma \tag{5.29}$$

Additionally the angle opposite to the upper leg has to be computed, because it corresponds to the foot pitch joint:

$$\delta_{footPitch1} = \arccos \frac{{l_{lowerLeg}}^2 + {l_{trans}}^2 - {l_{upperLeg}}^2}{2 \cdot l_{lowerLeg} \cdot l_{trans}} \tag{5.30}$$

Now the foot pitch and roll joints combined with the triangle form a kind of pan-tilt-unit. Their joints can be computed from the translation vector using atan2.[3]

$$\delta_{footPitch2} = \text{atan2}(x, \sqrt{y^2 + z^2}) \tag{5.31}$$

$$\delta_{footRoll} = \text{atan2}(y, z) \tag{5.32}$$

where x, y, z are the components of the translation of $Foot2HipOrthogonal$. As the foot pitch angle is composed by two parts it is computed as the sum of its parts.

$$\delta_{footPitch} = \delta_{footPitch1} + \delta_{footPitch2} \tag{5.33}$$

After the last three joints of the kinematic chain (viewed from the torso) are determined, the remaining three joints that form the hip can be computed. The joint angles can be extracted from the rotation matrix of the hip that can be computed by multiplications of transformation matrices. For this purpose another coordinate frame $Thigh$ is introduced that is located at the end of the upper leg, viewed from the foot. The rotation matrix for extracting the joint angles is contained in $HipOrthogonal2Thigh$ that can be computed by

$$HipOrthogonal2Thigh = Thigh2Foot^{-1} \cdot HipOrthogonal2Foot \tag{5.34}$$

where $Thigh2Foot$ can be computed by following the kinematic chain from foot to thigh.

$$Thigh2Foot = Rot_x(\delta_{footRoll}) \cdot Rot_y(\delta_{footPitch}) \cdot Trans_z(l_{lowerLeg}) \cdot Rot_y(\delta_{knee}) \cdot Trans_z(l_{upperLeg}) \tag{5.35}$$

To understand the computation of those joint angles, the rotation matrix produced by the known order of hip joints (yaw (z), roll (x), pitch (y)) is constructed (the matrix is noted abbreviated, e. g. c_x means $\cos \delta_x$).

$$Rot_{Hip} = Rot_z(\delta_z) \cdot Rot_x(\delta_x) \cdot Rot_y(\delta_y) = \begin{pmatrix} c_y c_z - s_x s_y s_z & -c_x s_z & c_z s_y + c_y s_x s_z \\ c_z s_x s_y + c_y s_z & c_x c_z & -c_y c_z s_x + s_y s_z \\ -c_x s_y & s_x & c_x c_y \end{pmatrix} \tag{5.36}$$

[3]atan2(y, x) is defined as in the C standard library, returning the angle between the x-axis and the point (x, y).

The angle δ_x can obviously be computed by $\arcsin r_{21}$.[4] The extraction of δ_y and δ_z is more complicated, they must be computed using two entries of the matrix, which can be easily seen by some transformation:

$$\frac{-r_{01}}{r_{11}} = \frac{\cos\delta_x \cdot \sin\delta_z}{\cos\delta_x \cdot \cos\delta_z} = \frac{\sin\delta_z}{\cos\delta_z} = \tan\delta_z \tag{5.37}$$

Now δ_z and, using the same approach, δ_y can be computed by

$$\delta_z = \delta_{hipYaw} = \mathrm{atan2}(-r_{01}, r_{11}) \tag{5.38}$$

$$\delta_y = \delta_{hipPitch} = \mathrm{atan2}(-r_{20}, r_{22}) \tag{5.39}$$

At last the rotation by 45 degrees (cf. eq. 5.25) has to be compensated in joint space.

$$\delta_{hipRoll} = \delta_x - \frac{\pi}{4} \tag{5.40}$$

Now all joints are computed. This computation is done for both legs, assuming that there is an independent hip yaw joint for each leg.

The computation described above can lead to different resulting values for the hip yaw joints. From these two joint values a single resulting value is determined, in which the interface allows to set the ratio. This is necessary, because if the values differ, only one leg can realize the desired target, and normally the support leg is supposed to reach the target position exactly. With this fixed hip joint angle the leg joints are computed again. To face the six parameters with the same number of degrees of freedom, a virtual foot yaw joint is introduced, which holds the positioning error provoked by the fixed hip joint angle. The decision to introduce a foot *yaw* joint was mainly taken because an error in this (virtual) joint has a low impact on the stability of the robot, whereas other joints (e. g. foot pitch or roll) have a huge impact on stability. The computation is almost the same as described above, with the difference that, caused by the fixed hip joint angle and the additional virtual foot joint, the computation is done the other way around, because the imagined pan-tilt-unit is now fixed at the hip and the universal joint is represented by the foot.

This approach can be realized without any numerical solution, which has the advantage of a constant and low computation time and a mathematically exact solution instead of an approximation.

Based on this inverse kinematic solver the module InverseKinematicEngine was built. It is controlled via simulator console (`get inverseKinematic`) and just sets the joint angles according to the given position. Among other things it is useful for creating special actions (cf. sect. 5.2.4), because the feet can be set to build a perfect plane, and to see the impact of specific relative foot positions on the robot, e. g., when developing walking motions.

5.2.4 Special Actions

Special actions are hardcoded motions that are provided by the module SpecialActions. By executing a special action, different target joint values are sent consecutively, allowing the robot to perform actions such as kicking or standing up. Those motions are defined in *.mof* files that

[4]The first index, zero based, denotes the row, the second index denotes the column of the rotation matrix.

are located in the folder *Src/Modules/MotionControl/mof*. A *.mof* file starts with the unique name of the special action, followed by the label *start*. The following lines represent sets of joint angles, separated by a whitespace. The order of the joints is as follows: head (pan, tilt), left arm (shoulder pitch/roll, elbow yaw/roll), right arm (shoulder pitch/roll, elbow yaw/roll), left leg (hip yaw-pitch/roll/pitch, knee pitch, ankle pitch/roll), and right leg (hip yaw-pitch[5]/roll/pitch, knee pitch, ankle pitch/roll). A '*' does not change the angle of the joint (keeping, e. g., the joint angles set by the head motion control), a '–' deactivates the joint. Each line ends with two more values. The first decides whether the target angles will be set immediately (the value is 0), forcing the robot to move its joints as fast as possible, or whether the angles will be reached by interpolating between the current and target angles (the value is 1). The time this interpolation takes is read from the last value in the line. It is given in milliseconds. If the values are not interpolated, the robot will set and hold the values for that amount of time instead.

It is also possible to change the hardness of the joints during the execution of a special action, which can be useful, e. g., to achieve a stronger kick while not using the maximum hardness as default. This is done by a line starting with the keyword *hardness*, followed by a value between 0 and 100 for each joint (in the same sequence as for specifying actual joint angles). In the file *Config/hardness.cfg* default values are specified. If only the hardness of certain joints should be changed, the others can be set to '*'. This will cause those joints to use the default hardness. After all joint hardness values, the time has to be specified that it will take to reach the new hardness values. This interpolation time runs in parallel to the timing resulting from the commands that define target joint angles. Therefore, the hardness values defined will not be reached if another hardness command is reached before the interpolation time has elapsed.

Transitions are conditional statements. If the currently selected special action is equal to the first parameter, the special action given in the second parameter will be executed next, starting at the position of the label specified as last parameter. Note that the currently selected special action may differ from the currently executed one, because the execution costs time. Transitions allow defining constraints such as *to switch from A to B, C has to be executed first*. There is a wildcard condition *allMotions* that is true for all currently selected special actions. There also is a special action called *extern* that allows leaving the module SpecialActions, e. g., to continue with walking. *extern.mof* is also the entry point to the special action module. Therefore, all special actions have to have an entry in that file to be executable. A special action is executed line by line, until a transition is found the condition of which is fulfilled. Hence, the last line of each *.mof* file contains an unconditional transition to *extern*.
An example of a special action:

```
motion_id = stand
label start
"HP HT AL0 AL1 AL2 AL3 AR0 AR1 AR2 AR3 LL0 LL1 LL2 LL3 LL4 LL5 LR0 LR1 LR2 LR3 LR4 LR5 Int Dur
*  *  0   -50 -2 -40  0   -50 -2  -40 -6  -1  -43 92  -48 0   -6  -1  -43 92  -48 -1  1   100
transition allMotions extern start
```

To receive proper odometry data for special actions, they have to be manually set in the file *Config/odometry.cfg*. It can be specified whether the robot moves at all during the execution of the special action, and if yes, how it has moved after completing the special action, or whether it moves continuously in a certain direction while executing the special action. It can also be specified whether the motion is stable, i. e., whether the camera position can be calculated correctly during the execution of the special action. Several modules in the process *Cognition* will ignore new data while an unstable motion is executed to protect the world model from being impaired by unrealistic measurements.

[5]Ignored

5.2.5 Motion Combination

The MotionCombinator requires the interpolation ratios of each motion in execution that are provided by the module MotionSelector (cf. Sect. 5.2.1). Using these ratios, the joint angles generated by the WalkingEngine and the SpecialActions module are merged together. The ratios are interpolated linearly. The interpolation time between different motions depends on the requested target motion.

The MotionCombinator merges the joint angles together to the final target joint angles. If there is no need to interpolate between different motions, the MotionCombinator simply copies the target joint angles from the active motion source into the final joint request. Additionally it fills the representations *MotionInfo* and *OdometryData* that contain data such as the current position in the walk cycle, whether the motion is stable, and the odometry position.

Chapter 6

Behavior Control

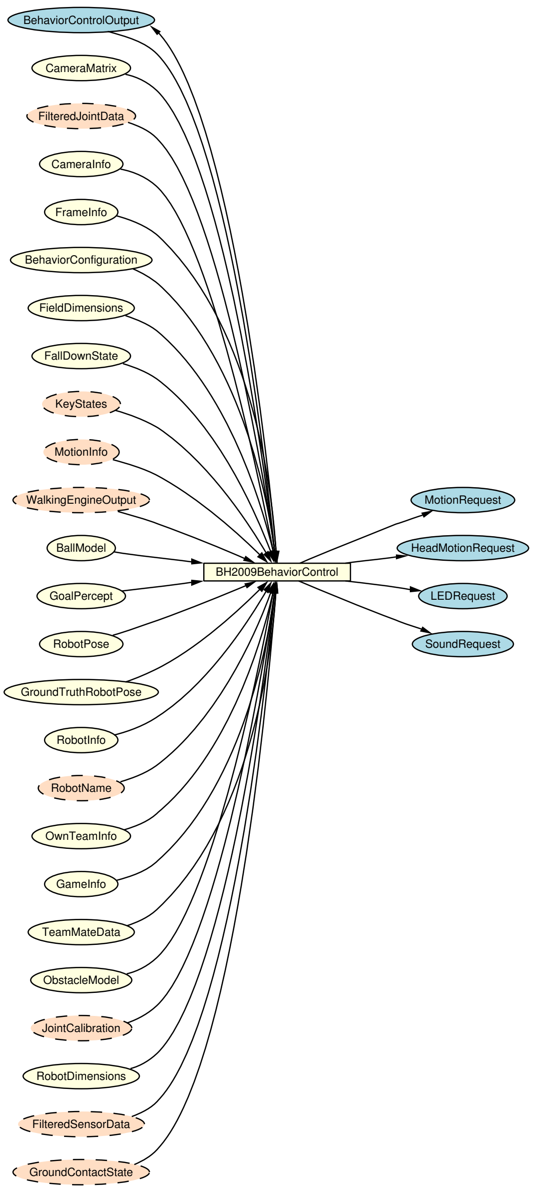


Figure 6.1: Behavior Control module graph

The part of the B-Human system that makes the decisions is called *Behavior Control*. The behavior was modeled using the Extensible Agent Behavior Specification Language (XABSL) [18]. The module provides the representations *MotionRequest*, *HeadMotionRequest*, *LEDRequest* and *SoundRequest* (cf. Fig. 6.1).

This chapter gives a short overview of XABSL and how it is used in a simple way. Afterwards, it is shown how to set up a new behavior. Both issues will be clarified by an example. Finally, the behavior used by B-Human at the RoboCup 2009 is explained in detail.

6.1 XABSL

XABSL is a programming language that is designed to model an agent behavior. To work with it, it is important to know its general structure. In XABSL following base elements are used: *options*, *states*, *decisions*, *input symbols*, and *output symbols*.
A behavior consists of options that are arranged in an option graph. There is a single option to start the whole behavior from which all other options are called; this is the root of the option graph. Each option describes a specific part of the behavior such as a skill or a head motion of the robot, or it combines such basic features. For this description each option consists of several states. Each option starts with its *initial state*. Inside a state, an action can be executed and optionally a decision can be made. An action can consist either of the modification of output symbols (for example head motion requests or walk requests), or a call of another option. A decision comprises conditions and transitions the latter of which are changes of the current state within the same option.

This structure is clarified with an example:

```
option example_option
{
  initial state first_state
  {
    decision
    {
      if(boolean_expression)
        goto second_state;
      else if(boolean_expression)
        goto third_state;
      else
        stay;
    }
    action
    {
      output_symbol = input_symbol * 3
    }
  }

  state second_state
  {
    action
    {
      secondOption();
    }
  }

  state third_state
```

```
    {
      decision
      {
        if(boolean_expression)
          goto first_state;
        else
          stay;
      }
      action
      {
        output_symbol = input_symbol < 0 ? 10 : 50;
      }
    }
  }
```

A special element within an option is the common decision. It consists of conditions which are checked all the time, independently of the current state, and it is always positioned at the beginning of an option. Decisions within states are only “else-branches” of the common decision, because they are only evaluated if no common decision is satisfied.

```
  option example_common_decision
  {
    common decision
    {
      if(boolean_expression)
        goto first_state;
      else if(boolean_expression)
        goto second_state;
    }
    initial state first_state
    {
      decision
      {
        else if(boolean_expression)
          goto second_state;
        else
          stay;
      }
      action
      {
        output_symbol = input_symbol * 3
      }
    }

    state second_state
    {
      decision
      {
        else if(boolean_expression)
          goto first_state;
        else
          stay;
      }
      action
      {
        output_symbol = input_symbol < 0 ? 10 : 50;
      }
    }
  }
```

Options can have parameters. The parameters have to be defined in a sub-option. Then these symbols can be passed from a superior option to the sub-option. Within the sub-option they can be used similar to input symbols by using an @ in front of the parameter name:

```
option example_superior_option
{
  initial state first_state
  {
    action
    {
      example_suboption(first_parameter = first_input_symbol, second_parameter = 140);
    }
  }
}

option example_suboption
{

  float @first_parameter [-3000..3000] "mm";
  float @second_parameter [-2000..2000] "mm";

  initial state first_state
  {
    action
    {
      output_symbol = @first_parameter - @second_parameter;
    }
  }
}
```

It is possible to define a *target state* within an option. When the option arrives at this target state the superior option has the possibility to query this status and to react on it. It is queried by the special symbol `action_done`:

```
option example_superior_option
{
  initial state first_state
  {
    decision
    {
      if(action_done)
       goto second_state;
    }
    action
    {
      example_suboption();
    }
  }
  state second_state
  {
    action
```

```
      {
        output_symbol = input_symbol'';
      }
    }
  }

  option example_suboption
  {

    initial state first_state
    {
      decision
      {
        if(boolean_expression)
          goto second_state;
        else
          stay;
      }
      action
      {
        output_symbol = @first_parameter - @second_parameter;
      }
    }

    target state second_state
    {
      action
      {
      }
    }
  }
```

Input and output symbols are needed to create actions and decisions within a state. Input symbols are used for the decisions and output symbols are used for the actions. Actions may only consist of symbols and simple arithmetic operations. Other expressions cannot be used in XABSL. All symbols are implemented in the actual robot code and range from math symbols to specific robot symbols.

6.2 Setting up a new Behavior

To set up a new behavior it is necessary to create a new folder in *Src/Modules/BehaviorControl*. This folder will contain the new behavior. To structure the behavior it is advisable to create some subfolder, such as folders for *Options* and *Symbols* (this is not mandatory). The option folder can be divided into subfolders such as skills, head motions, or roles. Inside the folder *Symbols*, all symbols shall be placed. To create symbols, a header file, a source file, and a Xabsl file are necessary. The files shall be used for groups of symbols such as head symbols, ball symbols, and so on. In this way it is easier to locate symbols later.

After creating all symbols needed it is necessary to create a file called *agents.xabsl* in the behavior folder, where all options needed are listed. This file is also important to get the behavior started later. Next to the *agents.xabsl* the following files have to be available in the newly created behavior folder: *<name>BehaviorControl.cpp*, *<name>BehaviorControl.h*, and *<name>BehaviorControlBase.h*. To understand how these files work, look at the comments in the corresponding files available in the *BH2009BehaviorControl* folder. It is important to include all symbols needed within the *<name>BehaviorControl.cpp*. After these files were created correctly, the *<name>BehaviorControl.h* has to be added to the *CognitionModules.h* that can be found in *Src/Modules*.

After this preparation, it is possible to write new options by creating the corresponding *.xabsl* files in the *Options* folder (or subfolders). An option consists of a name and all states needed that call each other or another option. Each new option has to be added to the file *agents.xabsl*, otherwise it cannot be used. When all options are implemented, one or more agents have to be created at the end of *agents.xabsl* that consists of a name and the option the agent shall start with. In the file *Locations/<location>/behavior.cfg* it will be selected, which agent will be started. At last it is necessary to modify the file *Cognition.cpp* that can be found in *Src/Processes/CMD*. In the method `handleMessage` the own behavior has to be added with a logical OR operator.

6.3 Behavior Used at Robocup 2009

In the following the behavior used at Robocup 2009 will be described, the option graph of which is shown in Fig. 6.2. It can be split into several parts which are explained hierarchically.

The root option of the whole behavior is the option *pre-initial* that is used to suppress the immediate stand up after starting the software. When the chest button is pressed the first time after starting the software, the robot stands up (this mechanism is suppressed for a simulated robot). Afterwards the option *start_soccer* becomes active that is the actual root option of the behavior. It executes the options *head_control*, *body_control*, *display_control*, *button_interface*, *penalty_control* and *role_selector* in parallel.

The *button_interface* is used to switch between the different game states by using the chest button and foot bumpers of the robot. The role selector sets the dynamically corresponding to the current game situation. According to the game state and the dynamic role, the *body_control* is used to invoke the corresponding options (e. g. the *initial*, *ready*, or *set* state and the different roles such as *striker*, *supporter*, *defender*, and *keeper*). The *head_control* executes a specific head control mode. The option *penalty_control* is responsible for the correct behavior relating to the penalty mode.

There is one more option *display_control* that is used to get an overview of the running behavior.

All options including the several roles are described in detail in the following sections.

6.3.1 Button Interface

This option includes several decisions, one for each possible change of the game state (initial, playing, penalized) (cf. Fig. 6.3). These game states can be changed through the chest button.

Besides the game state changes, the button interface includes decisions for changing the team color and the kickoff team. These can be changed by pressing the left or the right foot button, respectively. All changes are signalized acoustically.

Next to the button interface, the game state can also be controlled by the official GameController that overwrites all settings made via the button interface. The button interface is implemented according to the rules of 2009.

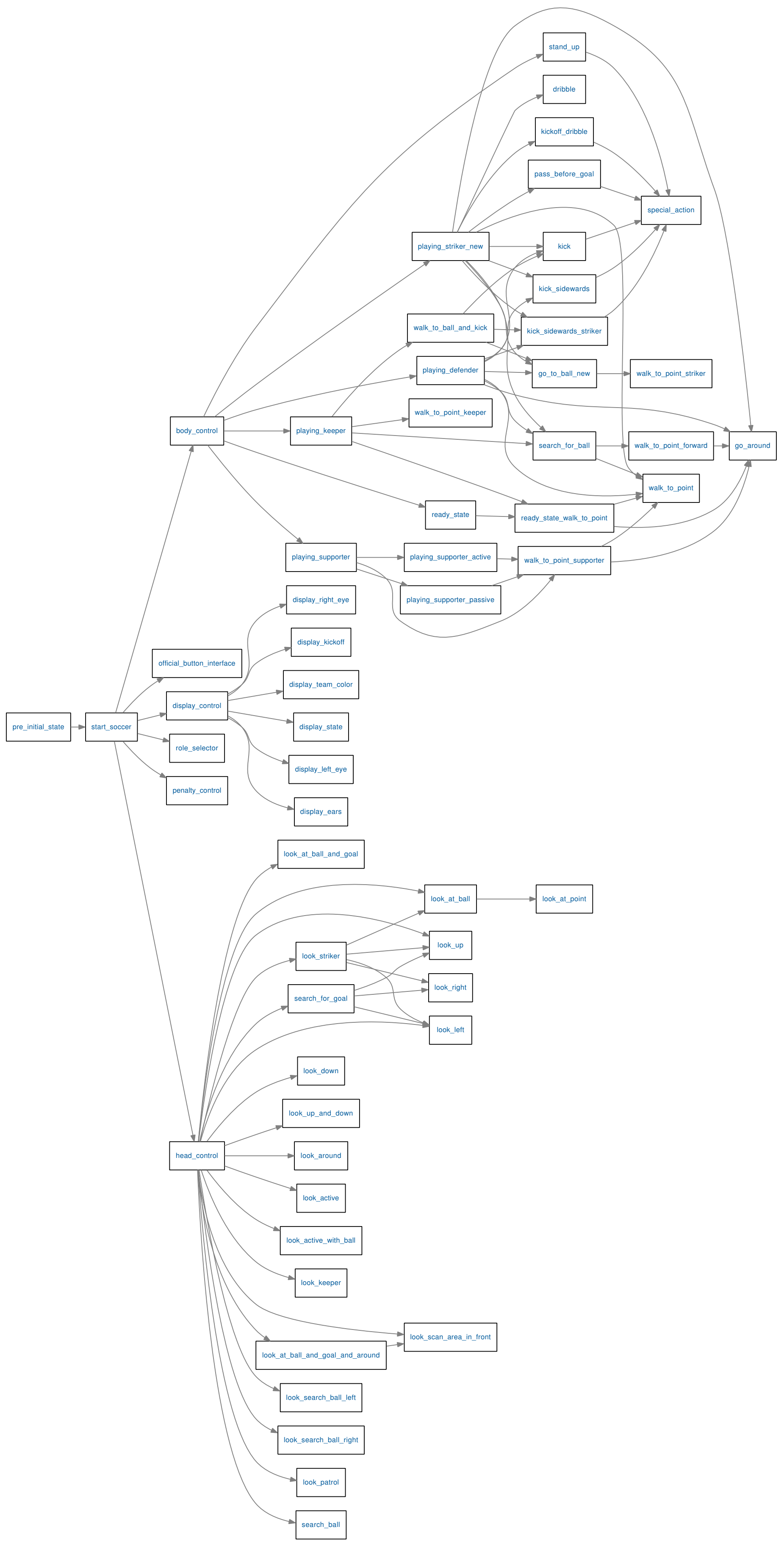


Figure 6.2: Option graph of behavior used at RoboCup 2009

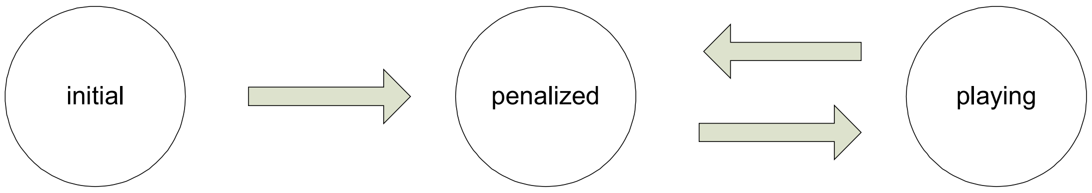


Figure 6.3: Button Interface

6.3.2 Body Control

The body control is used to invoke options that let the robot act. The execution of these options is dependent on the game states and the current role.

There are following reachable states:

state_initial. In this state the robot looks straight ahead without any further action.

state_ready. In the ready state the robots walk to their legal kickoff positions. At the beginning of this state the robots stand and look around to improve the localization. Afterwards the robots begin to walk to their positions. Each robot has another target position depending on its role. Moreover, the target positions differ for the attacking and defending team.

Because the robots can be placed by the robot handlers anywhere on the field, the target positions depend on the current positions of the individual robots and on the positions of their teammates. Specifically, the striker has the higher priority in reaching its position quickly, which means that its decision only depends on its own position on the field whereas the supporter chooses its target position depending on the striker's target, when the other team has kickoff. In case of kickoff, the supporter also chooses its target according to its own position, because the striker walks to the center of the field. The purpose is to prevent collisions generally and to distribute the robots preferably homogenously, i. e. it would be a disadvantage if all robots stood on the same side.

Besides this, obstacle avoidance and the fastest possible positioning are still important. The obstacle avoidance will be described in detail in Sect. 6.3.5.1. Fast positioning is achieved by a kind of path planning, in this case by simply handling large distances in another way than small distances to the target point. This is realized by turning around first and walking straight forward if the target point is far away and walking omni-directionally, when coming closer to the target.

state_set. While the set state is active the robot stands on its position and looks around (more details can be found in Sect. 6.3.3).

state_playing. When the game state changes to playing, the robots begin to play soccer. Therefore the behavior executed is dependent on the robots' roles, e. g., the supporter executes another behavior than the striker.

state_finished. The half-time or the game is over and the robot simply stops and looks straight ahead.

state_penalized. When a robot is penalized, it looks down without any further action until it is unpenalized again.

6.3.3 Head Control

The head control option controls the requests for the angles of the head joints, and therefore the direction in which the robot looks. In our behavior it is an independent option that runs in parallel to the body control option. In general, it takes requests from body control, and executes predefined motion patterns. The requests are passed from body control via the *head.control_mode* symbol. As long as the symbol does not change, the selected head control option is executed continuously. The problem with designing and choosing head controls is that the visual information provided by our vision system is required from many different software modules. GoalPerceptor, BallPerceptor, and LinePerceptor provide input for the modeling modules (Sect. 4.2) that provide localization and the ball position. However, providing images from all relevant areas on the field is often mutually exclusive, i. e., when the ball is located directly in front of the robot's feet, it cannot simultaneously look at the goal. In addition to only being able to gather some information at a time, speed constraints come into play, too. The solution to move the head around very fast to look at important areas more often proves impractical, since not only the images become blurred above a certain motion speed, but also because due to the weight of the head a high motion speed has a negative influence on the robot's walk, too. With these known limitations, we had to design many head control options for a variety of needs. Commonly used head controls are described below.

look_at_ball_and_goal_and_around. This option is used when there is no need for a specialized one. It looks at all important parts of the field to localize and play soccer. In a cycle, it looks at the ball three times, once at one goal, and to its left and right in the remaining time. If the ball is not where it is supposed to be at its calculated position, a scan of the area in front of the robot is performed, to reduce the need to switch the current behavior to *search_for_ball* (Sect. 6.3.5.1).

look_at_ball. Just looks at the estimated ball position. This is used when the robot is near the ball and aligns to the opponent goal and needs to find the perfect position for a kick. To reach such a position, we want to collect information about the ball position over a longer period of time, which this head control provides.

look_scan_area_in_front. Scans the area in front of the robot and looks for the ball. This is used to quickly look whether the ball is somewhere in front of the robot, after the ball was not where it was supposed to be, and when the robot returns to the field after a penalty.

look_up_and_down. Moves the head up and down in a continuous motion while panning to the left or right by 35 degrees. This option is used when the robot turns around to search for the ball. The left or right pan is equal to the turning direction, so that the robot actually looks ahead in the direction it turns to. This gives it enough time to stop turning and to walk around the ball without turning back if necessary.

6.3.4 Role Selector

The role selector is used to change the roles of the robots on the field dynamically. The available roles are *striker*, *supporter*, *defender*, and *keeper* (and *undefined* in the case that a robot is penalized). In general, the role selection is dependent on the number of field players, their current roles, and the current game situation. Additionally, the role selection is generally different for the *ready* state and the *playing* state (for the *initial* state and the *set* state, the role selection of the playing state is taken, which does not matter, because the robots are only standing anyway). As the role selection is mainly important for the *playing* state, it is described first.

Naturally, a robot that is penalized must be excluded from the role selection, and hence it gets the special role *undefined* independently of its environment. For the remaining robots the role selection mainly depends on the number of the robots that are currently on the field, which can be any number between zero and three. This variation can occur because of broken or penalized robots.

If no robot is on the field, no role selection needs to be done.

For the case that only one robot is on the field, this robot always gets the striker's role, independently of the native role and the game situation, because the striker is the only offensive role, whereas all other roles are defensive and particularly inactive until the ball enters their defense area.[1]

If two robots are on the field, generally a more defensive tactic is chosen. One player takes a defensive role (defender or keeper) whereas the other player becomes striker. The decision about the defensive role depends on whether the robot with player number 1 (per definition the keeper) is on the field. If this robot is available, this robot becomes keeper, and the other robot becomes striker. Otherwise the roles *defender* and *striker* are assigned based on their position on the field. The player that is closer to the own goal gets the defender's role and the other robot is striker.

The last possible case is that all three robots are on the field, which get the roles *striker*, *supporter*, and *keeper*. Whether this is an offensive or defensive tactic depends on the currently executed behavior of the supporter (cf. Sect. 6.3.5.2). Naturally the robot with player number 1 is keeper, whereas the supporter's and striker's role are assigned to the other robots. The decision about which robot becomes striker respectively supporter depends on their distance to the ball, i. e. the closer robot becomes striker. To suppress an oscillation of those two roles when both robots have almost the same distance to the ball, a hysteresis is introduced, i. e., the roles only change if the ball distances differ more than a specific tolerance. If the difference of the ball distances of both robots *is* within this tolerance, the roles do not change. If, for any reason, both robots are strikers and the mentioned difference is within the tolerance (e. g. synchronization problems, etc.), then the native roles of the robots determine the dynamic role. This is done to prevent a collision of two strikers trying to walk towards the ball. Additionally, there is a case where both field players are striker, namely if the previous striker has not seen the ball for a long time but assumes the ball close to itself. To prevent a game stuck because no teammate is allowed to walk towards the ball, the supporter also becomes striker in this case.

Besides the role selection for the *playing* state there is a special handling for the *ready* state. This role selection is only based on the native roles and the number of players on the field. The player with player number 1 is always the keeper, even if it is the only player on the field, because the keeper has a defined kickoff position. If only one field player[2] is available (no matter whether there also is a keeper or not), this player is striker. If there are two field players, the assignment of roles depends on whether a keeper is on the field or not. If the keeper is on the field, the field players are striker and supporter, otherwise the field players are striker and defender. In both cases the striker's role is determined by the native role, the other robot gets the remaining role.

6.3.5 Different Roles

In the following sections the behavior of the different roles is described in detail.

[1]According to this description the supporter is neither offensive nor defensive, and can only exist along with the striker and the keeper.

[2]In this case *field player* is meant to be any player except the keeper (the player with number 1).

6.3.5.1 Striker

The main task of the striker is to go to the ball and to kick it into the opponent goal. To achieve this simple-sounding target several situations need to be taken into account. All situations will be discussed in the following separate sections.

search_for_ball is executed when the ball has not been seen for a certain time. Right before its execution, the head control has done a full scan of the area in front of the robot, to rule out the possibility that the ball is there. Therefore the option starts directly with turning the robot around to search for the ball. It turns in the direction where the ball was seen last, and the head control looks ahead in the turning direction with a 35 degrees pan, while looking up and down continuously. After a whole turn the robot enters a patrolling mode. In this state, it walks between some positions and searches for the ball from there, as a clear view on the ball may be obstructed by other robots, but it may be visible from other positions (cf. Fig. 6.4). Certain roles do not patrol though, i. e., the keeper just turns around and does not leave its goal unguarded, while the defender scans the area in front of it for some time and then turns around. As some roles may not see the ball for longer periods of time, e. g., the keeper because the ball is hard to detect in the opponent half of the field, they rely on the position communicated by teammates and keep their position instead of searching the ball.

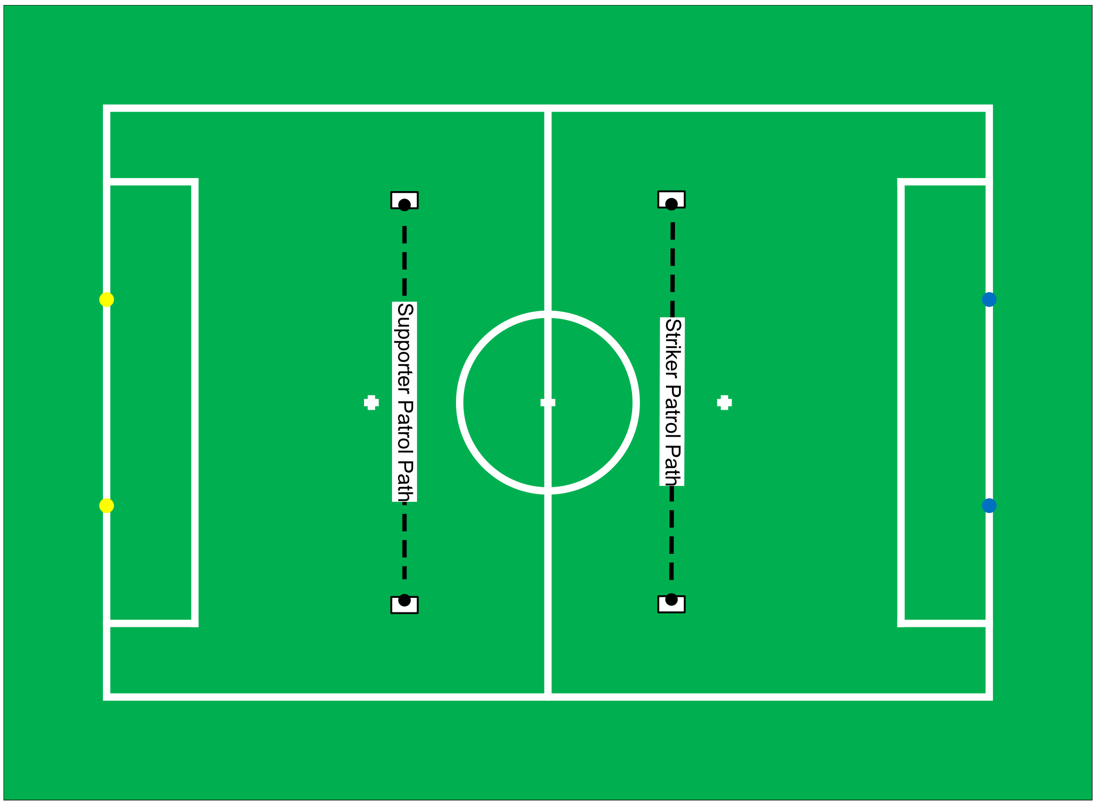


Figure 6.4: Patrol paths for striker and supporter roles. Robots at the end of the lines depict stop positions.

kickoff. Another special situation is the kickoff. When the own team has kickoff, it is not allowed to score a goal until the ball has left the center circle. This means that a normal "go-to-ball-and-kick" must not be executed, instead another behavior needs to be used. The kickoff behavior differs for different number of robots on the field. When only one or two robots are on the field, the striker executes a short diagonal kick to get the ball out of the center circle. In this case the foot used for the kick is only dependent on the position of the ball. The diagonal kick is executed in this situation, because there is no supporting player on the field who could receive a pass (only keeper or defender, or no other robot is on the field).

In the other case, when all three robots are on the field, a sideways kick is executed by the striker. The direction of the kick is dependent on the supporter's position so that the striker always passes towards the supporter. In the case that the ball does not leave the center circle for any reason the striker walks towards the ball and kicks it again either diagonally or sideways until it leaves the center circle.

go_to_middle. The striker's behavior contains a special handling for the case that the ball is near the own penalty area or when the keeper walks towards the ball. In this case, the striker walks to the middle line of the field. The striker turns towards the field which means that the robot is turned to the left side when it stands on the right and vice versa. This special behavior has two reasons: On the one hand, collisions of the striker with its teammates are avoided. On the other hand the striker has a better starting position for the expected following situation (the keeper should kick the ball away from the own goal into the opponent half).

dribble. Normally, the robot does not dribble during the game, because aligning to a kick and kicking is more effective and fast enough in most cases. The only situation when the striker dribbles is, when the ball is lying on the opponent goal line. In this case the robot aligns behind the ball and dribbles it into the goal by walking against the ball.

pass_before_goal. When the robot and the ball are near the opponent goal but the angle between the goalposts is too small to hit the goal with sufficiently high probability, the robot does not kick directly towards the goal. Instead, it performs a slow kick to the side towards the center of the field. For the case that a supporter is on the field, it receives this pass. Otherwise, the striker walks to the ball to kick directly, under the assumption that the angle between the goalposts is large enough then.

kick_to_side. Besides the situations mentioned above, in which a slow sideways kick is used, there is one more situation, when this kick is performed, namely when the striker reaches the ball and has no possibility to kick forward towards the opponent goal because an obstacle such as another robot is detected right in front of it. In this case the robot has different possibilities to react. If three robots are on field the striker executes a sideways kick towards the supporter.

If only one or two robots are on the field, the position of the striker determines the further action. When the robot is near the side border of the field it executes a slow sideways kick towards the center of the field. This kick is slower than the sideways kick used for passing because there is no team player receiving a pass. The striker has to walk after the ball, which should not take too much time (otherwise the ball could be lost to the opponent). When the striker resides near the center of the field it executes a diagonal kick towards the free direction. When the obstacle is shown more on the right the striker kicks diagonally to the left and vice versa.

go_around. It is possible that the striker encounters an obstacle on its way to the ball. In this case it needs to go around the obstacle without touching it. The obstacles are measured by an ultrasonic sensor (for details see Sect. 4.2.3). When the distance to an obstacle is very small, the striker walks sideways until it is possible to pass it. The direction in which the striker avoids depends on the measurements of the obstacle and the target position (normally the position of the ball).

First of all a preference for the walking direction is determined only based on the measurements on the left and on the right side without distinguishing the different sectors on one side. For this preferred side, it is checked whether the obstacle is detected outside. If so, the robot *must* walk to the opposite side not to touch the obstacle. Otherwise if the obstacle is detected more centrally, the target position determines the side to walk to. This means, that there is a possibility that the robot walks to the side where the obstacle is detected assuming that there is enough free space left. The decision on the direction for avoidance is only done once to prevent an oscillation.

The avoidance of the striker is regulated. The more the obstacle is detected in the center, the more the robot walks sideways. The more the obstacle gets to the side, the more the robot walks forward. Hence the striker can walk closely diagonally around an obstacle.

go_to_ball. When no special situation is active (such as kickoff, obstacle avoidance, or behavior near the goal) and the ball is not directly in front of the feet of the striker, it walks towards the ball, which is done in different ways for different situations. If the ball and goal are in front of the robot it walks behind the ball to be able to perform a kick forward towards the goal. To reach the ball in a preferably short time it first turns and then walks straight towards the ball when the distance to it is quite high. When the distance becomes smaller the robot begins walking omni-directionally towards a position behind the ball oriented towards the goal with the ball directly in front of its feet (similar to the positioning in *state_ready*).

The other possibility is that the striker's position is between ball and goal or on a level with the ball, so that it normally needed to walk around the ball completely. In this case the robot walks next to the ball instead of behind the ball. The side the robot walks to is dependent on the angle to the ball and the goal. The "path-planning" is similar to the situation to get behind the ball, except that the target position is oriented towards the ball and in a right angle to the opponent goal.

kick_sidewards. The sideways kick is used, when the striker stands in a right angle towards the opponent goal and directly in front of the ball (cf. *go_to_ball*). In this case, it shall execute a strong sideways kick. It is possible that the robot needs to walk closer to the ball to improve its position. When the ball is directly in front of the robot it looks at the ball for a short time to make sure that the ball is lying in front of it, and then it kicks towards the opponent goal.

kick. After the robot walked behind the ball, it needs to execute a forward kick. It might be necessary to improve the position, because either the ball does not lie centered in front of one foot or the robot is not precisely oriented towards the opponent goal, which is done within this option.

Because it is not always possible to get the ball lying precisely centered in front of one foot, two different kicks are used. One kick is used when the ball lies correctly in front of one foot, whereas the other kick is used when the ball lies more outside of the center of the foot (this kick is weaker compared to the other one).

6.3.5.2 Supporter

The supporter's role is used only when there are three robots on the field. Its task is to support its team members. There are two different modes for the supporter, an active and a passive one. The supporter is active when the ball is on the opponent side of the field. In that case the striker is supported, otherwise the keeper.

The following two situations relate to both, the active and the passive supporter.

kickoff_behavior. During the own kickoff the supporter waits outside the center circle for the striker's pass. The side depends on where the supporter was placed before the kickoff. When the kickoff is over, i. e. the ball has left the center circle, the usual behavior of the supporter starts.

wait_for_keeper. When the keeper starts walking towards the ball to kick it out, the supporter walks to a point between the center line and the penalty area to get out of the way. After the keeper has kicked out the ball, the supporter switches to the active or passive behavior again.

Active supporter

The active supporter is constantly trying to reach a position that enables the robot to score a goal, if the striker itself has no chance and has to pass the ball.

go_to_good_position. Most of the time the supporter tries to go to a position relative to the striker. That position is behind the striker with an offset to the side the striker is not at. The intention is that if the striker decides to pass the ball in the supporter's direction, there is a good chance to score a goal.

To avoid an oscillation, the supporter stays on the same side when the striker is close to the middle. Fig. 6.5 illustrates the position of the active supporter next to the striker.

wait. To avoid a collision, the supporter compares its own position to the one of the striker. If the distance falls below a certain threshold, the robot stops. If the distance gets even smaller, i. e. the striker is walking towards the supporter, the supporter walks in the opposite direction.

go_to_good_position_in_front_of_opponent_goal. When the ball is close to one of the opponent corners of the field, the supporter stops following the striker. Instead the robot walks to a point in front of the opponent goal and waits for a pass from the striker. Fig. 6.6 illustrates the position of the active supporter in front of the opponent goal.

Passive Supporter

The passive supporter is responsible for supporting the keeper defending the goal. There are two different situations the passive supporter can be in. Either the keeper is inside the goal or it was placed at the center line and is on its way back to the goal.

align_to_team_mate. If the keeper is inside its goal the supporter aligns to it, depending on the keeper's position. If the keeper is on the right side of the goal, the supporter tries to cover the left side, otherwise the right side of the goal. Fig. 6.7 illustrates the position of the passive supporter next to the keeper.

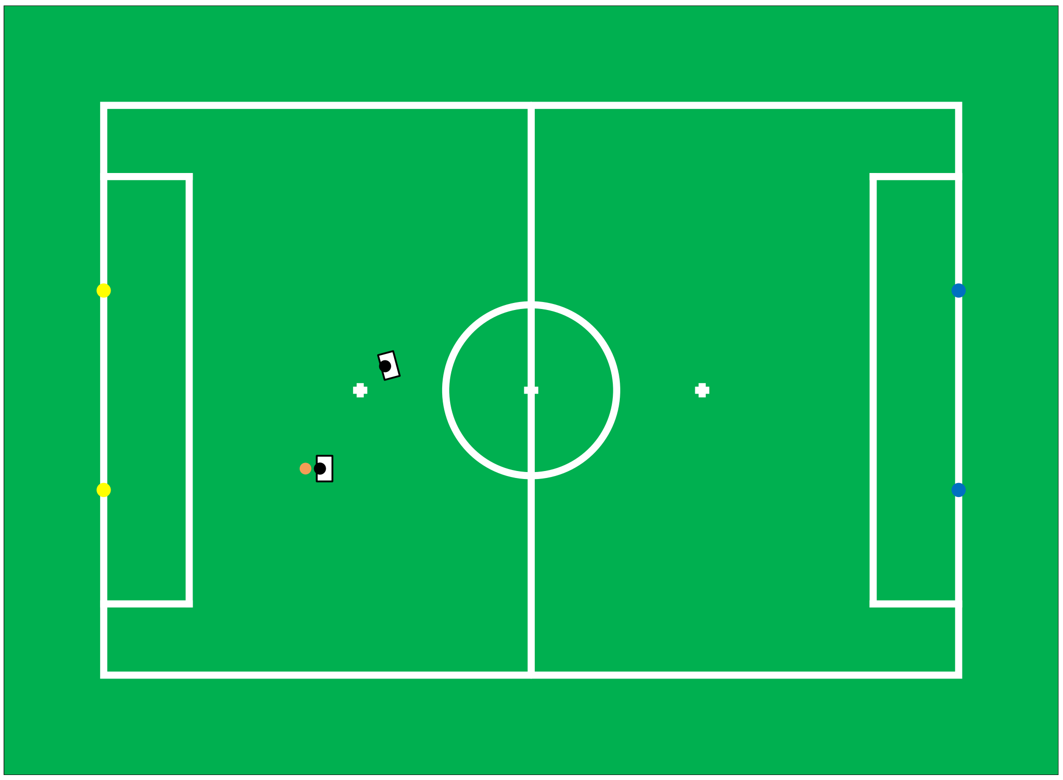


Figure 6.5: Positioning of the active supporter next to the striker

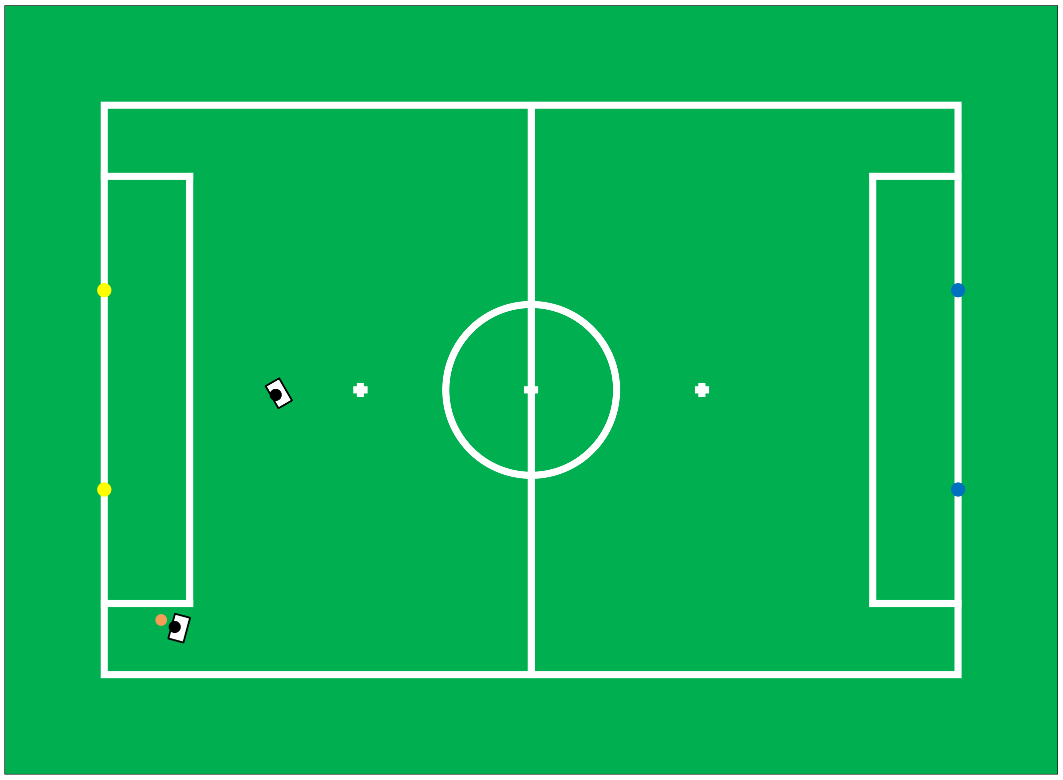


Figure 6.6: Positioning of the active supporter in front of the opponent goal

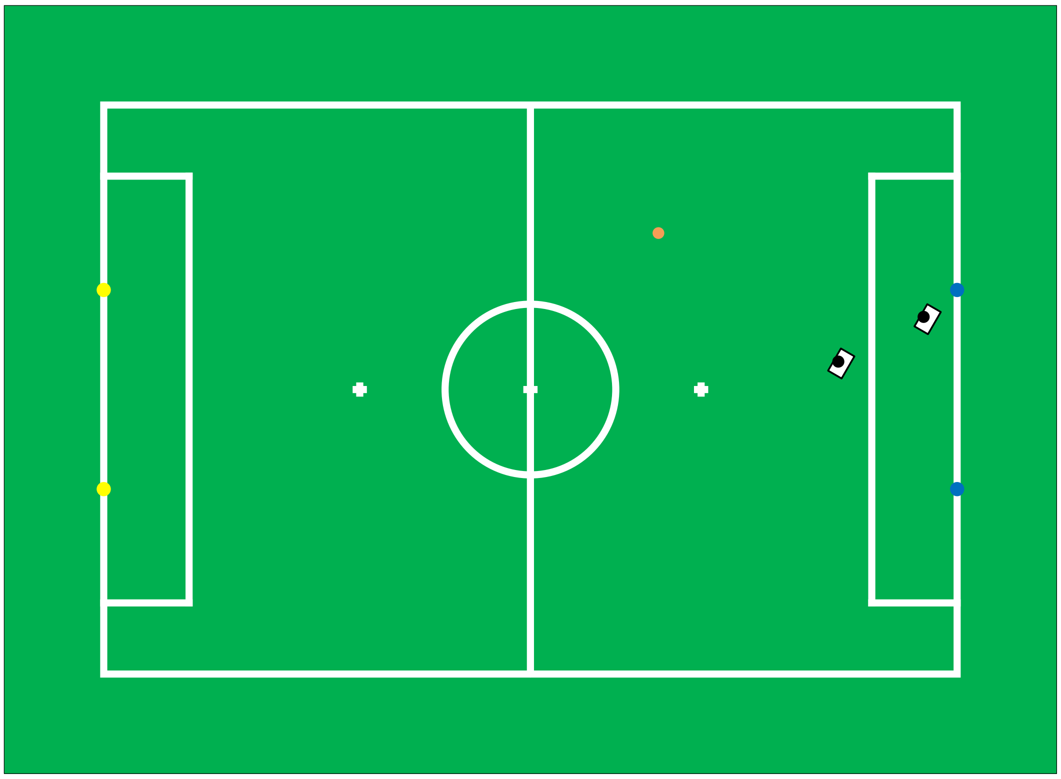


Figure 6.7: Positioning of the passive supporter next to the keeper

align_to_ball. If the keeper is not inside its goal, the supporter positions itself between the ball and its own goal. The position should prevent from shots that go straight towards the goal. When the keeper has returned to the goal, the supporter starts aligning to its team mate again.

6.3.5.3 Defender

The defender's role is only used when the keeper is out of the field. This role is necessary to have a field player near the own goal to be able to defend it when only two robots are on the field. In general the defender has a fixed x-position on the field, which means that the robot only walks right and left on an imaginary line (cf. Fig. 6.8). The position on that line is dependent on the position of the ball because the robot shall always be located between ball and goal to cover as much as possible of the goal when a kick of the opponent team is executed. However, the positioning of the defender is not its only task. The positioning only takes place if the ball is not near or behind the defender and consequently not near the own goal. If the ball was not seen for a long time, the robot searches for the ball (cf. *search_for_ball*). Moreover the robot's task is to walk to the ball if it is lying near or behind the imaginary line mentioned. The aim of walking to the ball is to kick it away from the own half of the field towards the opponent goal.

On the way to the ball it is possible that the defender encounters an obstacle such as another robot. In this case, the defender needs to walk around the obstacle. For this task the same obstacle avoidance as for the striker is used (cf. Sect. 6.3.5.1). The "go-to-ball-and-kick"-behavior is also borrowed from the striker, as it has been proven to be very quick and reliable. This also includes a quite precise alignment towards the goal that is necessary because it would be a disadvantage to simply kick the ball away, as the ball would probably leave the field and would be put back again closer to the own goal.

If an obstacle is detected right in front of the defender when it wants to kick, it executes a similar behavior as the striker does in this situation when no supporter is available, i. e. it will execute a diagonal kick or a kick to the side instead of kicking straight forward. This is useful in this situation, because otherwise the ball could bounce from the obstacle and roll behind the defender towards the own goal.

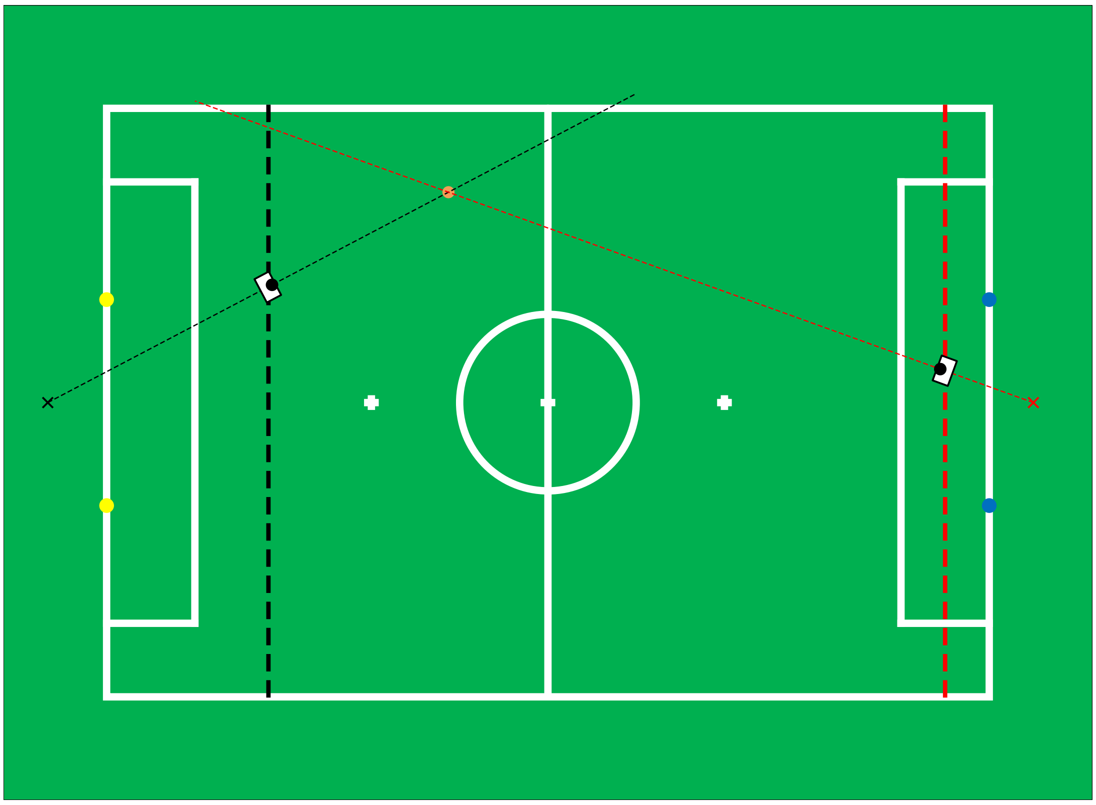


Figure 6.8: Imaginary lines for position of defender and keeper

6.3.5.4 Keeper

The keeper is the only field player that is allowed to walk into the own penalty area and whose main task is to defend the own goal. Hence the keeper mainly stays inside the own penalty area. There are only two situations in which the robot leaves the own goal. Either the keeper is taken out of the field or it walks towards the ball to kick it away. When the robot was taken from the field and put back on the middle line, the keeper walks directly towards its own goal. Naturally, the time to reach the own goal should take as less time as possible. To achieve this, the robot walks straight forward to the own goal as long as the distance is large, and starts walking omni-directionally when it comes closer to the target to reach the desired orientation. This speed-up compensates the disadvantage that the robot is probably directed away from the ball. In this case the keeper avoids obstacles in the same way as the striker and defender.

The second situation when the robot leaves the own goal is when the ball is close to the own goal and has stopped moving. In this case the robot walks towards the ball in order to kick it away. The kind of walking to the ball is the same as for the striker and defender, with the difference that the keeper does not align with the same accuracy in order to kick the ball away from the own goal as soon as possible. The keeper does not avoid obstacles in this case. It only tries to walk towards the ball without paying attention to obstacles. The advantage is that the keeper does not clear the way towards the goal for the opponent, additionally even if it touches another robot and is penalized, it will be put back on the field without the obligatory 30-seconds-penalty.

In any other situation the robot will not leave its position within the own penalty area, and it executes different actions according to different situations. Normally the keeper aligns on an imaginary horizontal line inside the penalty area to position between the ball and the center of the goal to cover a preferably large part of it (cf. Fig. 6.8). This behavior is similar to the defender (cf. Sect. 6.3.5.3), but in contrast the keeper sits down when it has reached a good position (cf. Fig. 6.9). This is useful to cover much more of the own goal compared to a standing robot. That sitting position is executed until the ball position changes significantly so that the keeper needs to change its position. The stand-up or sit-down motion is not executed immediately after recognizing a ball position outside the "tolerance region": when the ball position changes the robot checks the new position of the ball again after a short time period and then decides whether it stands up or stays sitting down and vice versa. This is necessary to avoid an unwanted stand-up-sit-down cycle, because of a short period of uncertainty in the ball position or self-localization.

There is another special handling for the case that the robot recognizes the ball moving fast towards it: if the robot is already sitting, it stays in this position, even if it assumes that the ball will pass. B-Human did not implement a diving motion, because of the possible damage to the robot. Since the ball is not always rolling straight, covering a large part of the goal is a better strategy than standing up, even if it seems that the ball will pass by.

Another special situation for the keeper is when it has not seen the ball for a long period of time. In this case, the reaction of the keeper is dependent on the team mates. When the own field players assume the ball near the own goal, or they have not seen it either for a longer period of time, the keeper starts searching for the ball (cf. Sect. 6.3.5.1). In all other cases, the keeper walks to the center of the goal on the imaginary line and sits down assuming that the ball is not close to the own goal.

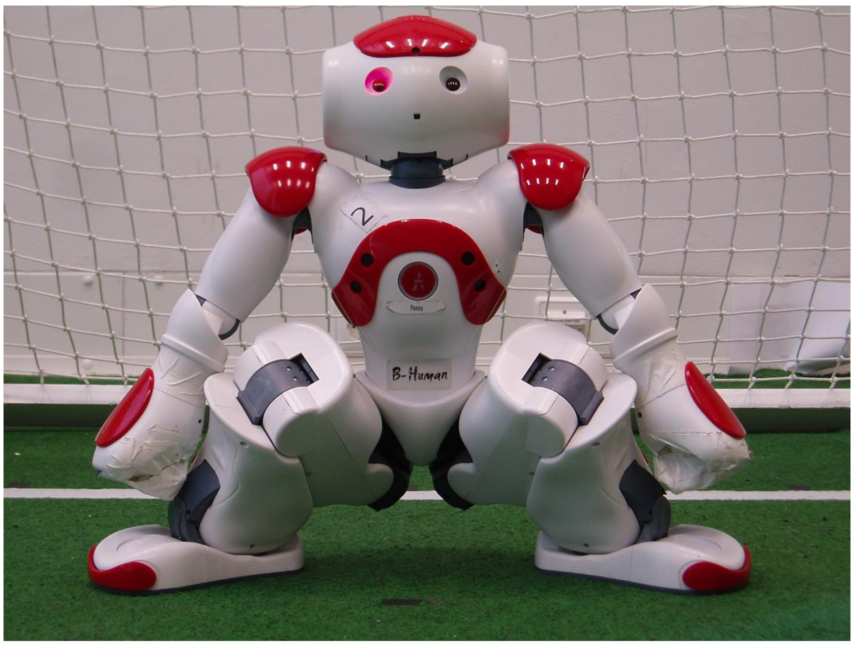


Figure 6.9: Sitting position of the keeper

6.3.6 Penalty Control

Penalty control is responsible to stop the robot's movement while penalized. It is the last option to be called within *start_soccer*. Therefore, it can overwrite the movement requests made by the body and head control options. It is implemented as a separate option (and not separately in body control and head control) to have a single place easily accessible to change penalty behavior. Basically, when the robot is not penalized, this option does nothing. When the robot is penalized, it sets the motion request to *stand* and lets the robot look down. In the first two seconds after the robot is unpenalized, it keeps control over the robot's movement and lets it walk straight forward to get back onto the field. More importantly, the robot looks at a predefined angle of 45 degrees to its left. This is done to help localize the robot, as the particles in the self-locator are evenly distributed at the two possible reentry points on the field, and if the goal color on one side can be determined, the robot knows exactly on which side of the field the robot was put back into the game. After that, the robot's head is moved from left to right to quickly check whether the ball can be seen in front of it.

6.3.7 Display Control

The LEDs of the robot are used to show information about the internal state of the robot, which is useful when it comes to debugging the code.

6.3.7.1 Right eye

Color	**Role**	**Additional information**
Blue	Keeper	blinks if the ball comes up to the keeper
White	Defender	
Green	Supporter	blinks if the supporter is active
Red	Striker	

6.3.7.2 Left eye

Color	**Information**
White	Ball was seen
Green	Goal was seen
Red	Ball and goal were seen

6.3.7.3 Torso (chest button)

Color	**State**
Off	Initial, Finished
Blue	Ready
Green	Playing
Yellow	Set
Red	Penalized

6.3.7.4 Feet

- The right foot shows the team color. If the team is currently the red team the color of the LED is red, otherwise blue.

- The left foot shows whether the team has kickoff or not. If the team has kickoff the color of the LED is white, otherwise it is switched off.

6.3.7.5 Ears

- The right ear shows the battery level of the robot. For each 10% of battery loss, an LED is switched off. Additionally the LEDs that are still on start blinking when the distance to a detected obstacle is less than 30 cm.
- The left ear shows the number of players connected by the wireless. If no LED is on, no player, if half of the LEDs are on one player, and if all LEDs are on two players are connected to the current robot.

Chapter 7

Challenges

For the RoboCup 2009 Technical Challenge, B-Human has developed approaches for all three sub-challenges. Although we did not score in the Any Ball Challenge as well as in the Passing Challenge, both implementations are almost complete and accomplished the tasks many times during tests on different fields and using different start configurations.

7.1 Any Ball Challenge

The aim of the Any Ball Challenge is to manipulate five unknown balls different from the orange standard ball. For this challenge, we had to adapt the modules we are already using to recognize a ball and to develop a new module that enables the robot to keep track of more than one ball on the field.

A major problem of the challenge is that the color and the texture of the ball are not known in advance. Thus, we adapted the RegionAnalyzer and the BallPerceptor not to work on white and orange anymore but on the color class *NONE* that includes all colors except for green, blue, and yellow. Thereby, we do not need to specify any real color class to lines or balls. This is the smallest color table we can work with (cf. Fig. 7.1).

Another problem was the management of five different balls on the field without losing track of them. In this challenge, there is only one moving robot on the field, therefore we can assume that a ball does not move until we kick or dribble it. For this reason, we decided to track the balls' positions only by their absolute position on the field without using a more complex modeling as the one used in the ParticleFilterBallLocator. Whenever the AnyBallPerceptor detects one or more balls in the image, these balls are added to a list of all known balls. If there already is a ball close to the absolute position for the seen ball in the list of all known balls, the new ball is not added but the existing ball information is updated instead. This list of detected balls is managed by the AnyBallSelector. The AnyBallSelector also selects the ball that is reported to the behavior as the next ball to shoot. This selection is done by calculating the distance between the robot and each ball and selecting the nearest one.

Our tests showed that our self-localization is precise and stable enough to rely on it, so the only major problem in identifying balls by their absolute position on the field and maintaining a list of seen balls were false positives and shot balls. To overcome these problems, the AnyBallSelector deletes balls after a certain period of time of not being seen and deletes a ball immediately if the behavior reports that this ball was shot. Thus, the AnyBallSelector is forced to select a new ball for the behavior after the previous one was shot.

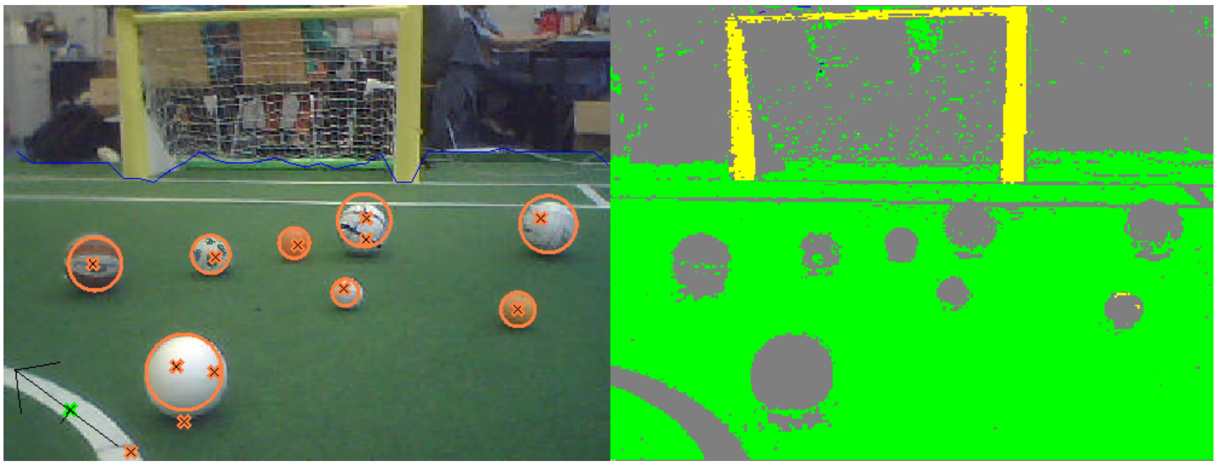


Figure 7.1: Example for the AnyBallPerceptor: possible balls (marked by orange crosses) and accepted balls (marked by orange circles) are shown in the left image. The right image results from a full color segmentation based on a rudimentary color table.

Since the rules specify that the balls are distributed within an area of 3 m × 2 m around the center line, the robot's initial behavior is to follow that line. In case of recognizing a ball, the robot walks towards it and kicks it in the direction of the goal that is nearest to it. After each kick, the robot will return to the center line and continue to follow it (including a turn at the end of line), unless it sees another ball.

7.2 Passing Challenge

The intention of the passing challenge is to encourage teams to develop passing and catching skills. Besides that, a robot who participates in this challenge has to follow some rules. First of all, the robots are not allowed entering the middle area of the field that is marked by two invisible lines. These lines are parallel to the middle line and tangential to the center circle. The second rule is that none of the participating robots is allowed leaving the field during the challenge. As the two robots are set to different penalty areas for the start, these rules implicate that the robots have to stay in their half of the field. The last rule is about achieving points: The team obtains a point if one of the robots passes the ball over the middle area and the other robot touches it. When the third pass was played, the challenge is finished successfully.

With that in mind, the behavior for the passing challenge has been divided into two roles: the passive robot that waits for the ball to roll along and the active robot that goes to the ball and passes it to the other robot. Both robots are passive players at the beginning, because when a ball has never been seen before or has not been seen for a very long time, none of the robots should go to a ball position that is probably wrong.

As each robot starts as a passive player that has never seen the ball each robot starts with turning and searching for it. If there is a ball on the field, one of the robots should discover it shortly after the beginning. The recognition of the ball causes the robot to change its behavior depending on its own position and the position of the ball. The position of the ball is known to the player when the ball was seen by itself or by the other robot (that can communicate it). If one the robots decides that the ball is not within its own half of the field, its role stays passive. This causes the robot to approach a probably good receiving point and to turn itself towards the direction from which the ball is expected. The behavior of the robot switches back to turning

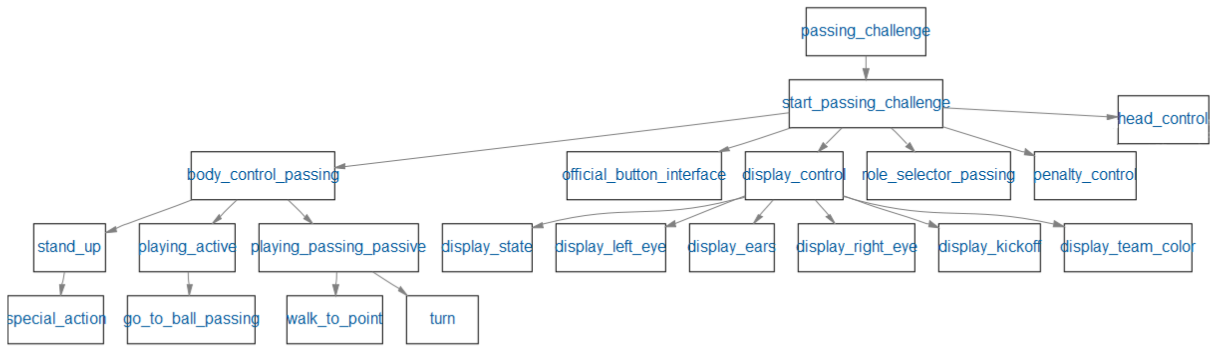


Figure 7.2: The passing challenge behavior option graph

and searching for the ball if the ball was not seen for more than six seconds.

The searching for the ball is an adapted version of the one used by the soccer behavior. It causes the robot to scan the area in front of it faster as it does in the original version and to rotate to find the ball if it was not seen for a while. As the desire is to find the ball fast, the direction of the rotation is derived from the last known position of the ball.

In the case that one robot discovers the ball within its own half of the field, its role switches to become the active player. In this role, the robot has to approach the ball and to pass it to the other robot. As this challenge only takes three minutes, the robot should arrive at the ball already in the desired passing angle to waste no time for adjustments. For this purpose, we implemented a *go-to-ball* behavior that gets the desired passing angle and the position of the ball as input and approaches the ball in a bow if the direct way leads to a wrong passing angle. When the robot reaches the target position, the ball is passed into the other half of the field.

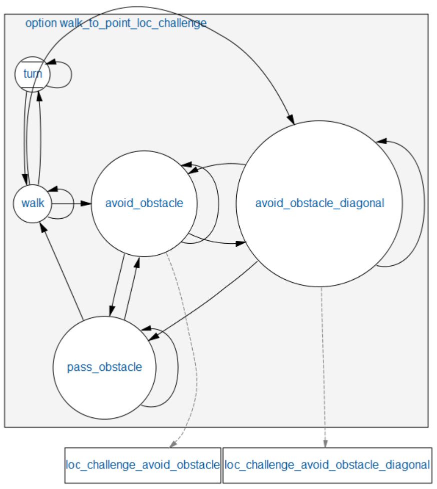
The current passing action is the key of this challenge because the ball needs to arrive almost exactly at the position where the other robot is located. If the pass is too long, the ball possibly leaves the field, which is one of the conditions to fail the challenge. If the pass is too short and the ball stays in the middle area – that no robot is allowed to enter – the challenge is also over. This implies that the passing action depends on the ball position as well as of the position of the active player. For that reason, an early version of a dynamical motion engine that allows a parameterization of the kicking force, was used.

In the case of a successful pass, which means no fail conditions were satisfied, the role of the active player switches back to the passive role, when the robot discovers the ball in the other half of the field or if the ball was not seen by the robot for more than ten seconds. The successful pass causes the other robot, the receiver, to switch its role to active because the ball has to be in its own half of the field. As the robot approaches the ball and passes the ball back to the other half of the field a point is scored. This cycle lasts until the challenge ends.

The skeletal structure of this behavior is based on the default soccer behavior to obtain the official button control, the LED assignment and the reaction to the GameController.

7.3 Localization with Obstacle Avoidance Challenge

For this challenge, no new modules have been implemented since both necessary core capabilities have already been developed for the soccer competition (cf. Sect. 4.2.1 and Sect. 4.2.3). Except for a single parameter change making the obstacle model more sensitive to definitely avoid

a) 
option walk_to_point_loc_challenge
turn
walk
avoid_obstacle
avoid_obstacle_diagonal
pass_obstacle
loc_challenge_avoid_obstacle
loc_challenge_avoid_obstacle_diagonal

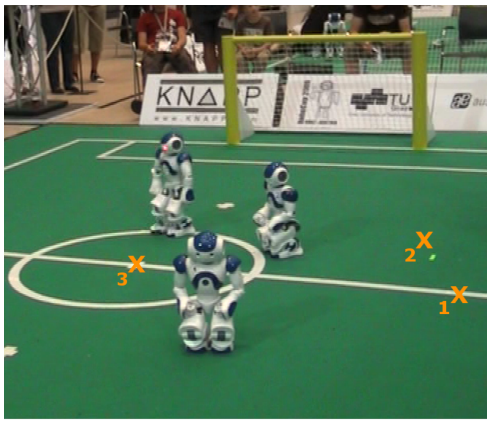
b) 

Figure 7.3: Localization with Obstacle Avoidance Challenge: a) State machine describing the option for walking and obstacle avoidance. b) B-Human accomplishing the challenge at RoboCup 2009, the numbered crosses denote the target positions in visiting order.

overlooking any robot, no changes of these components have been necessary.

The overall behavior for this challenge has been specified as follows: initially, the robot turns on the spot and scans its environment for ten seconds. After this period of time, it is very likely to have a reasonable pose estimate for almost every possible start configuration. Given the robot's position and the three points to navigate to, the shortest metric path (not including any necessary rotations) is computed. This path remains fixed during the whole challenge to avoid any behavior instabilities resulting from temporary localization inaccuracies. For walking from point to point and avoiding obstacles, an option different to the one used in the soccer behavior had to be implemented, as described in the following paragraph and depicted in Fig. 7.3a. Being near a target position, the robot slows down and tries to reach the point as precisely as possible; this phase of improvement is limited to seven seconds to avoid wasting too much time.

To keep all obstacles within the range of the ultrasonic sensors and to move with reasonable speed, the robot does only walk forward (with slight rotational corrections) and – in case of huge rotational deviations – turns on the spot. In case of an obstacle at medium distance (less than 500 mm), the robot tries to avoid it by walking diagonally. If the obstacle is too close to the robot (less than 350 mm), only sideways motion is allowed anymore. After having passed the obstacle (i. e. being more than 600 mm away), some additional forward steps are done to leave it behind. This approach is more time-consuming than the according soccer option but significantly reduces the risk of any collisions that are allowed in soccer but strongly penalized in this challenge. The test runs in several different configurations of the challenge setup – most of them have been much more spacious than the one used at RoboCup 2009 (cf. Fig. 7.3b) – also showed that time would not be a major problem. Since the approach is reactive and does not involve any planning, it is of course not capable of leaving any local minima.

Chapter 8

SimRobot

8.1 Introduction

The B-Human software package uses the physical robotics simulator SimRobot [16, 14] as front end for software development[1]. The simulator is not only used for working with simulated robots, but it also functions as graphical user interface for replaying log files and connecting to actual robots via LAN or WLAN.

8.2 Scene View

The scene view (cf. Fig. 8.1 right) appears if the *scene* is opened from the scene graph. The view can be rotated around two axes, and it supports several mouse operations:

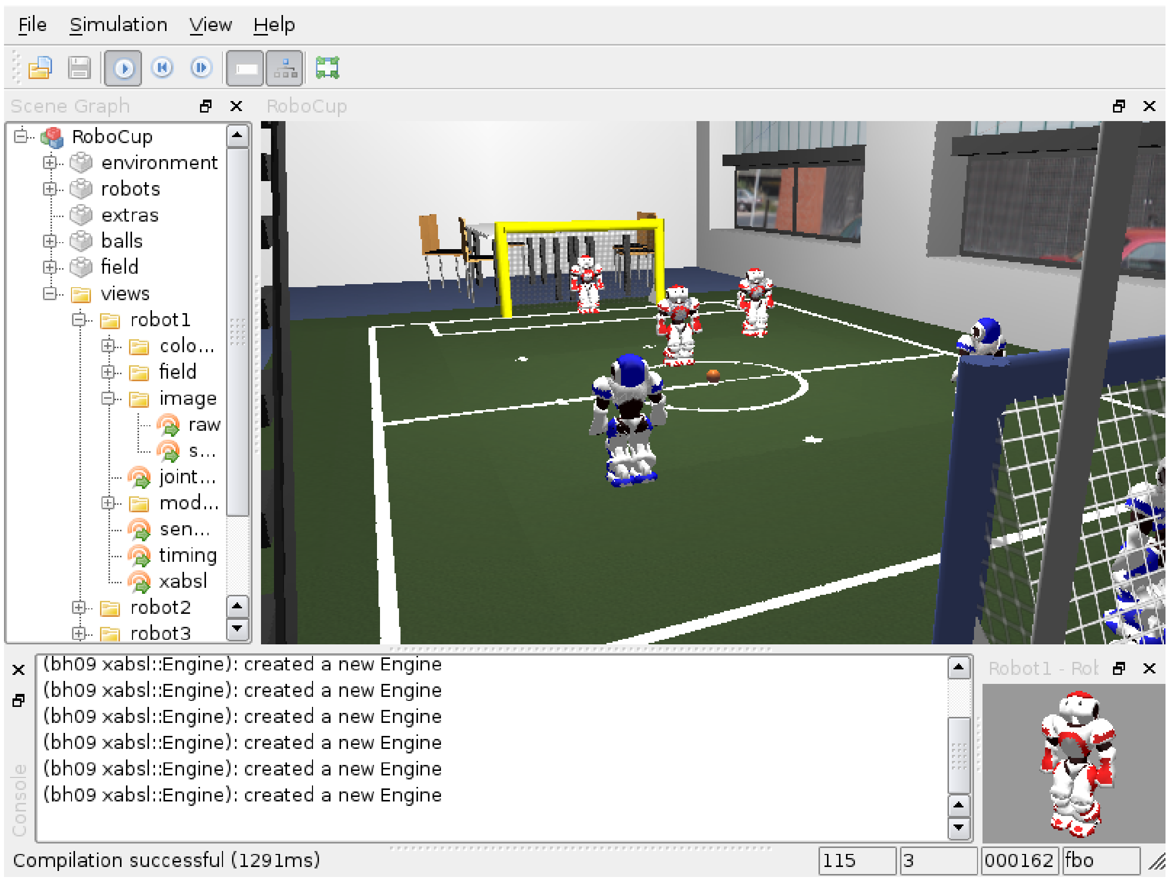
- Left-clicking an object allows dragging it to another position. Robots and the ball can be moved in that way.
- Left-clicking while pressing the *Shift* key allows rotating objects around their body centers.
- Select an *active* robot by double-clicking it. Robots are active if they are defined in the compound *robots* in the scene description file (cf. Sect. 8.4).

  Robot console commands are sent to the selected robot only (see also the command *robot*).

8.3 Information Views

In the simulator, *information views* are used to display debugging output such as debug drawings. Such output is generated by the robot control program, and it is sent to the simulator via *message queues*. The views are interactively created using the console window, or they are defined in a script file. Since SimRobot is able to simulate more than a single robot, the views are instantiated separately for each robot. There are nine kinds of views related to information received from robots: *image views*, *color space views*, *field views*, the *Xabsl view*, the *sensor data view*, the *joint data view*, *plot views*, the *timing view*, and *module views*. Field, image, and plot

[1]SimRobot currently becomes revised in preparation for a stand-alone release in 2010. Therefore, the functionality of sensors and actuators that are *not* used by the models contained in the B-Human software release cannot be guaranteed.



```
(bh09 xabsl::Engine): created a new Engine
(bh09 xabsl::Engine): created a new Engine
(bh09 xabsl::Engine): created a new Engine
(bh09 xabsl::Engine): created a new Engine
(bh09 xabsl::Engine): created a new Engine
(bh09 xabsl::Engine): created a new Engine
```

Compilation successful (1291ms)

Figure 8.1: SimRobot with the tree view on the left and two scene views on the right. The console window is shown at the bottom.

views display debug drawings or plots received from the robot, whereas the other views visualize certain color channels, the current color table, specific information about the current state of the robot's behavior, its sensor readings, the timing of the modules it executes, or the module configuration itself. All information views can be selected from the scene graph (cf. Fig. 8.1 left).

8.3.1 Image Views

An image view (cf. left of Fig. 8.2) displays information in the system of coordinates of the camera image. It is defined by giving it a name and a background image using the console command *vi* and by adding debug drawings to the view using the command *vid* (cf. Sect. 8.5.3).

For instance, the view *raw* is defined as:

```
vi image raw
vid raw representation:LinePercept:Image
vid raw representation:BallPercept:Image
vid raw representation:GoalPercept:Image
```

If color table editing is activated, image views will react to the following mouse commands (cf. commands *ct on* and *ct off* in Sect. 8.5.3):

Left mouse button. The color of the pixel or region selected is added to the currently selected

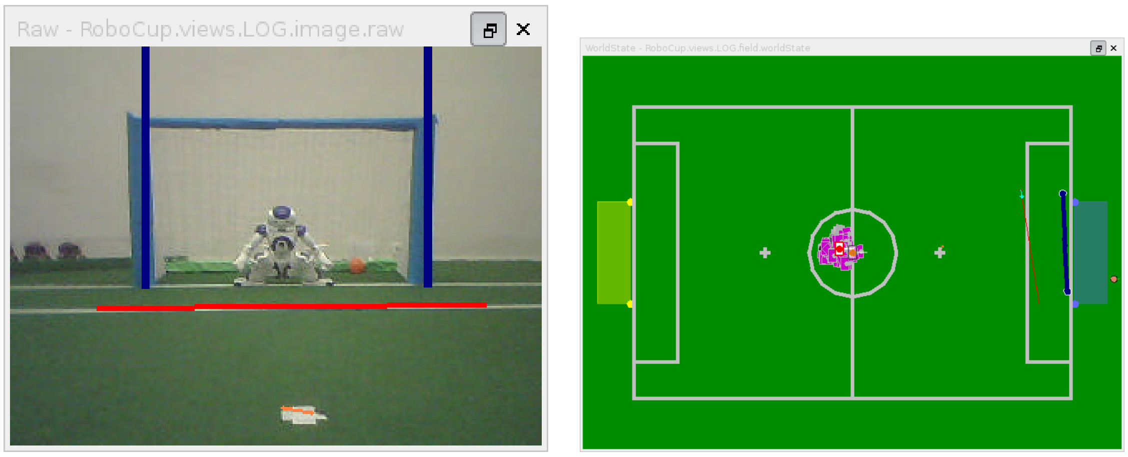

Raw - RoboCup.views.LOG.image.raw
WorldState - RoboCup.views.LOG.field.worldState

Figure 8.2: Image view and field view with several debug drawings

color class. Depending on the current configuration, the neighboring pixels may also be taken into account and a larger cube may be changed in the color table (cf. commands *ct imageRadius* and *ct colorSpaceRadius* in Sect. 8.5.3). However if a region is selected, the *imageRadius* is ignored.

Left mouse button + Shift. If only a single pixel is selected, the color class of that pixel is chosen as the current color class. It is a shortcut for *ct <color>* (cf. Sect. 8.5.3). If a region is selected, all colors of pixels in that region that are not already assigned to a color class are assigned to the selected color class. Thus all colors in a certain region can be assigned to a color class without destroying any previous assignments.

Left mouse button + Ctrl. Undoes the previous action. Currently, up to ten steps can be undone. All commands that modify the color table can be undone, including, e. g., *ct clear* and *ct load* (cf. Sect. 8.5.3).

Left mouse button + Shift + Ctrl. The color of the pixel selected is deleted from its color class. Depending on the current configuration, the neighboring pixels may also be taken into account and a larger cube is changed in the color table (cf. commands *ct imageRadius* and *ct colorSpaceRadius* in Sect. 8.5.3). However if a region is selected, the *imageRadius* is ignored.

8.3.2 Color Space Views

Color space views visualize image information in 3-D (cf. Fig. 8.3). They can be rotated by clicking into them with the left mouse button and dragging the mouse afterwards. There are three kinds of color space views:

Color Table. This view displays the current color table in YCbCr space. Each entry that is assigned to a certain color class is displayed in a prototypical color. The view is useful while editing color tables (cf. Fig. 8.3 down right).

Image Color Space. This view displays the distribution of all pixels of an image in a certain color space (*HSI*, *RGB*, *TSL*, or *YCbCr*). It can be displayed by selecting the entry *all* for a certain color space in the scene graph (cf. Fig. 8.3 top right).

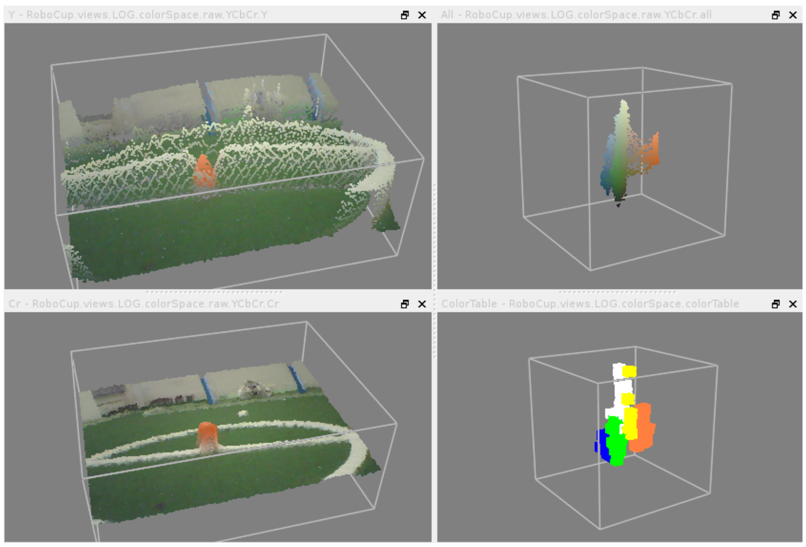

Y - RoboCup.views.LOG.colorSpace.raw.YCbCr.Y
All - RoboCup.views.LOG.colorSpace.raw.YCbCr.all
Cr - RoboCup.views.LOG.colorSpace.raw.YCbCr.Cr
ColorTable - RoboCup.views.LOG.colorSpace.colorTable

Figure 8.3: Color channel views, image color space view, and color table view

Image Color Channel. This view displays an image while using a certain color channel as height information (cf. Fig. 8.3 left).

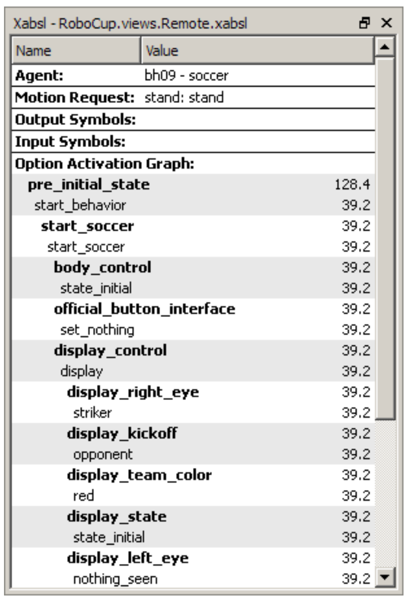
While the color table view is automatically instantiated for each robot, the other two views have to be added manually for the camera image or any debug image. For instance, to add a set of views for the camera image under the name *raw*, the following command has to be executed:

```
v3 image raw
```

8.3.3 Field Views

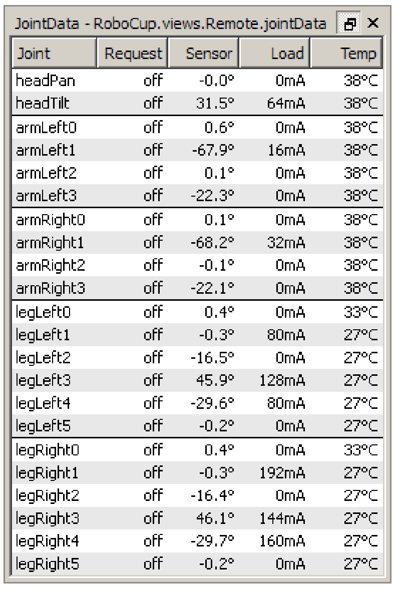
A field view (cf. right of Fig. 8.2) displays information in the system of coordinates of the soccer field. It is defined similar to image views. For instance, the view *worldState* is defined as:

```
vf worldState
vfd worldState fieldPolygons
vfd worldState fieldLines
vfd worldState module:SelfLocator:samples
vfd worldState representation:RobotPose

# ground truth view layers
vfd worldState representation:GroundTruthRobotPose
# from now, relative to ground truth robot pose
vfd worldState origin:GroundTruthRobotPose
vfd worldState representation:GroundTruthBallModel

# views relative to robot
# from now, relative to estimated robot pose
vfd worldState origin:RobotPose
vfd worldState representation:BallModel
vfd worldState representation:BallPercept:Field
vfd worldState representation:LinePercept:Field
vfd worldState representation:GoalPercept:Field
```



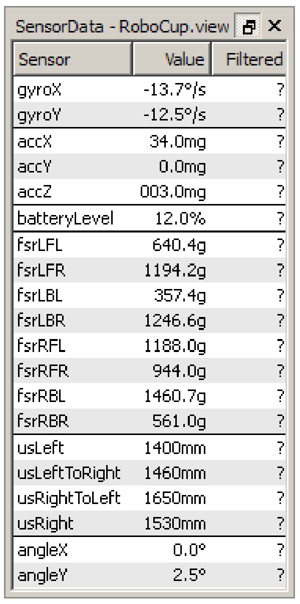
Xabsl - RoboCup.views.Remote.xabsl

Name	Value
Agent:	bh09 - soccer
Motion Request:	stand: stand
Output Symbols:	
Input Symbols:	
Option Activation Graph:	
pre_initial_state	128.4
start_behavior	39.2
start_soccer	39.2
start_soccer	39.2
body_control	39.2
state_initial	39.2
official_button_interface	39.2
set_nothing	39.2
display_control	39.2
display	39.2
display_right_eye	39.2
striker	39.2
display_kickoff	39.2
opponent	39.2
display_team_color	39.2
red	39.2
display_state	39.2
state_initial	39.2
display_left_eye	39.2
nothing_seen	39.2



JointData - RoboCup.views.Remote.jointData

Joint	Request	Sensor	Load	Temp
headPan	off	-0.0°	0mA	38°C
headTilt	off	31.5°	64mA	38°C
armLeft0	off	0.6°	0mA	38°C
armLeft1	off	-67.9°	16mA	38°C
armLeft2	off	0.1°	0mA	38°C
armLeft3	off	-22.3°	0mA	38°C
armRight0	off	0.1°	0mA	38°C
armRight1	off	-68.2°	32mA	38°C
armRight2	off	-0.1°	0mA	38°C
armRight3	off	-22.1°	0mA	38°C
legLeft0	off	0.4°	0mA	33°C
legLeft1	off	-0.3°	80mA	27°C
legLeft2	off	-16.5°	0mA	27°C
legLeft3	off	45.9°	128mA	27°C
legLeft4	off	-29.6°	80mA	27°C
legLeft5	off	-0.2°	0mA	27°C
legRight0	off	0.4°	0mA	33°C
legRight1	off	-0.3°	192mA	27°C
legRight2	off	-16.4°	0mA	27°C
legRight3	off	46.1°	144mA	27°C
legRight4	off	-29.7°	160mA	27°C
legRight5	off	-0.2°	0mA	27°C



SensorData - RoboCup.view

Sensor	Value	Filtered
gyroX	-13.7°/s	?
gyroY	-12.5°/s	?
accX	34.0mg	?
accY	0.0mg	?
accZ	003.0mg	?
batteryLevel	12.0%	?
fsrLFL	640.4g	?
fsrLFR	1194.2g	?
fsrLBL	357.4g	?
fsrLBR	1246.6g	?
fsrRFL	1188.0g	?
fsrRFR	944.0g	?
fsrRBL	1460.7g	?
fsrRBR	561.0g	?
usLeft	1400mm	?
usLeftToRight	1460mm	?
usRightToLeft	1650mm	?
usRight	1530mm	?
angleX	0.0°	?
angleY	2.5°	?

Figure 8.4: Xabsl view, sensor data view and joint data view

```
vfd worldState representation:ObstacleModel
vfd worldState module:ObstacleModelProvider:us
vfd worldState representation:MotionRequest

# back to global coordinates
vfd worldState origin:Reset
```

Please note that some drawings are relative to the robot rather than relative to the field. Therefore, special drawings exist (starting with *origin:* by convention) that change the system of coordinates for all drawings added afterwards, until the system of coordinates is changed again.

8.3.4 Xabsl View

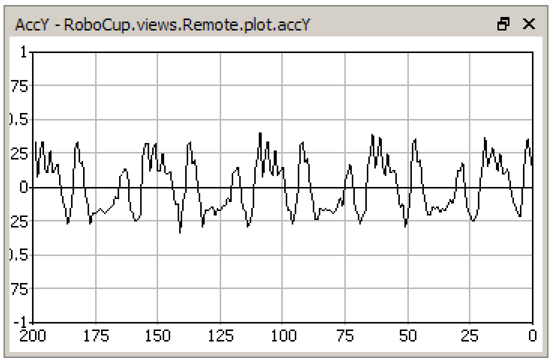
The Xabsl view is part of the set of views of each robot. It displays information about the robot behavior currently executed (cf. left view in Fig. reff:simXabslSensorJointData). In addition, two debug requests have to be sent (cf. Sect. 8.5.3):

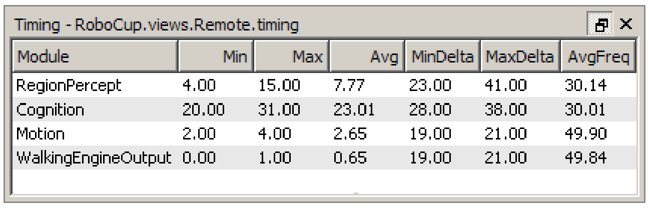
```
# request the behavior symbols once
dr automatedRequests:xabsl:debugSymbols once

# request continuous updates on the current state of the behavior
dr automatedRequests:xabsl:debugMessages
```

The view can also display the current values of input symbols and output symbols. The symbols to display are selected using the console commands *xis* and *xos* (cf. Sect. 8.5.3).

8.3.5 Sensor Data View

The sensor data view displays all the sensor data taken by the robot, e. g. accelerations, gyro measurements, pressure readings, and sonar readings (cf. right view in Fig. 8.4). To display this information, the following debug requests must be sent:




Timing - RoboCup.views.Remote.timing

Module	Min	Max	Avg	MinDelta	MaxDelta	AvgFreq
RegionPercept	4.00	15.00	7.77	23.00	41.00	30.14
Cognition	20.00	31.00	23.01	28.00	38.00	30.01
Motion	2.00	4.00	2.65	19.00	21.00	49.90
WalkingEngineOutput	0.00	1.00	0.65	19.00	21.00	49.84

Figure 8.5: The plot view and the timing view

```
dr representation:SensorData
dr representation:FilteredSensorData
```

8.3.6 Joint Data View

Similar to sensor data view the joint data view displays all the joint data taken by the robot, e. g. requested and measured joint angles, temperature and load (cf. middle view in Fig. 8.4). To display this information, the following debug requests must be sent:

```
dr representation:JointRequest
dr representation:JointData
```

8.3.7 Plot Views

Plot views allow plotting data sent from the robot control program through the macro `PLOT` (cf. Fig. 8.5 left). They keep a history of the values sent of a defined size. Several plots can be displayed in the same plot view in different colors. A plot view is defined by giving it a name using the console command *vp* and by adding plots to the view using the command *vpd* (cf. Sect. 8.5.3).

For instance, the view on the left side of Figure 8.5 was defined as:

```
vp accY 200 -1 1
vpd accY module:SensorFilter:accY blue
```

8.3.8 Timing View

The timing view displays statistics about the speed of certain modules (cf. Fig. 8.5 right). It shows the minimum, maximum, and average runtime of the execution of a module in milliseconds. In addition, the average frequency with which the module was executed (in Hz), as well as the minimum and maximum time difference between two consecutive executions of the module (in milliseconds) is displayed. All statistics sum up the last 100 invocations of the module. The timing view only displays information on modules the debug request for sending profiling information of which was sent, i. e., to display information about the speed of the module that generates the representation *Foo*, the console command *dr stopwatch:Foo* must have been

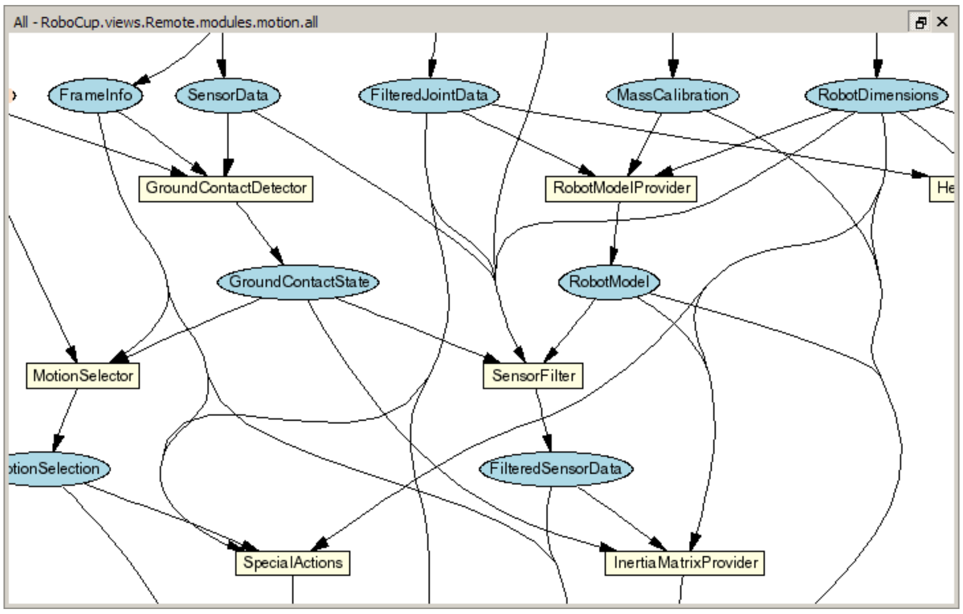


Figure 8.6: The module view showing part of the modules in the process *Motion*

executed. Please note that time measurements are limited to full milliseconds, so the maximum and minimum execution durations will always be given in this precision. However, the average can be more precise.

8.3.9 Module Views

Since all the information about the current module configuration can be requested from the robot control program, it is possible to automatically generate a visual representation. The graphs such as the one that is shown in Figure 8.6 are generated by the program *dot* from the *Graphviz* package [5] in the background. Modules are displayed as yellow rectangles and representations are shown as blue ellipses. Representations that are received from another process are displayed in orange and have a dashed border. If they are missing completely due to a wrong module configuration, both label and border are displayed in red. The modules of each process can either be displayed as a whole or separated into the categories that where specified as the second parameter of the macro `MAKE_MODULE` when they were defined. There is a module view for the process *Cognition* and its categories *Infrastructure*, *Perception*, *Modeling* and *BehaviorControl*, and one for the process *Motion* and its categories *Infrastructure*, *Sensing* and *MotionControl*.

8.4 Scene Description Files

The language of scene description files is RoSiML [6]. In the main *.ros* files, such as *BH2009.ros*, three compounds can be edited:

<Compound name="robots">. This compound contains all *active* robots, i. e. robots for which processes will be created. So, all robots in this compound will move on their own. However, each of them will require a lot of computation time.

<Compound name="extras">. Below the compound *robots*, there is the compound *extras*.

It contains *passive* robots, i. e. robots that just stand around, but that are not controlled by a program. Passive robots can be activated by moving their definition to the compound *robots*.

<Compound name="balls">. Below that, there is the compound *balls*. It contains the balls, i. e. normally a single ball, but it can also contain more of them if necessary, e. g., for a technical challenge that involves more than one ball.

A lot of scene description files can be found in *Config/Scenes*. Please note that there are two types of scene description files: the ones required to simulate one or more robots (about 3 KB in size, but they include larger files), and the ones that are sufficient to connect to a real robot or to replay a log file (about 1 KB in size).

8.5 Console Commands

Console commands can either be directly typed into the console window or they can be executed from a script file. There are three different kinds of commands. The first kind will typically be used in a script file that is executed when the simulation is started. The second kind are *global commands* that change the state of the whole simulation. The third type is *robot commands* that affect currently *selected robots* only (see command *robot* to find out how to select robots).

8.5.1 Initialization Commands

sc <name> <a.b.c.d>. Starts a wireless connection to a real robot. The first parameter defines the *name* that will be used for the robot. The second parameter specifies the ip address of the robot. The command will add a new robot to the list of available robots using *name*, and it adds a set of views to the scene graph. When the simulation is reset or the simulator is exited, the connection will be terminated.

sl <name> <file>. Replays a log file. The command will instantiate a complete set of processes and views. The processes will be fed with the contents of the log file. The first parameter of the command defines the *name* of the virtual robot. The name can be used in the *robot* command (see below), and all views of this particular virtual robot will be identified by this name in the tree view. The second parameter specifies the name and path of the log file. If no path is given, *Config/Logs* is used as a default. Otherwise, the full path is used. *.log* is the default extension of log files. It will be automatically added if no extension is given.

When replaying a log file, the replay can be controlled by the command *log* (see below). It is even possible to load a different log file during the replay.

su <name> <number>. Starts a UDP team connection to a remote robot with a certain player number. Such a connection is used to filter all data from the team communication regarding a certain robot. For instance it is possible to create a field view for the robot displaying the world model it is sending to its teammates. The first parameter of the command defines the *name* of the robot. It can be used in the *robot* command (see below), and all views of this particular robot will be identified by this name in the scene graph. The second parameter defines the number of the robot to listen to. It is required that the command *tc* (see below) is executed before this one.

tc <port> <subnet>. Listens to the team communication on the given UDP *port* and broadcasts messages to a certain *subnet*. This command is the prerequisite for executing the *su* command.

8.5.2 Global Commands

call <file>. Executes a script file. A script file contains commands as specified here, one command per line. The default location for scripts is the directory from which the simulation scene was started, their default extension is *.con*.

cls. Clears the console window.

dt off | on. Switches simulation dragging to real-time on or off. Normally, the simulation tries not to run faster than real-time. Thereby, it also reduces the general computational load of the computer. However, in some situations it is desirable to execute the simulation as fast as possible. By default, this option is activated.

echo <text>. Print text into the console window. The command is useful in script files to print commands that can later be activated manually by pressing the *Enter* key in the printed line.

help [<pattern>], ? [<pattern>]. Displays a help text. If a pattern is specified, only those lines are printed that contain the pattern.

ro stopwatch (off | <letter>) | (sensorData | robotHealth | motionRequest | linePercept) (off | on). Set a release option sent by team communication. The *release options* allow sending commands to a robot running the actual game code. They are used to toggle switches that decide which additional information is broadcasted by the robot in the team communication. *stopwatch* activates time measurements of all stopwatches in the process *Cognition* the name of which begin with the letter specified as additional parameter. If *sensorData* is active, the robot broadcasts the representation of the same name, containing the charge level of the battery and the temperature of all joints. However, this package is rather huge. *robotHealth* activates sending the *RobotHealth* package that contains compact information about the battery, process execution frame rates, and the highest temperature of all joints. *motionRequest* sends the motion request currently executed by the robot. *linePercept* transmits the *LinePercept*. Again, this representation is rather large. Both can be visualized in the field view.

robot ? | all | <name> {<name>}. Selects a robot or a group of robots. The console commands described in the next section are only sent to *selected robots*. By default, only the robot that was created or connected last is selected. Using the *robot* command, this selection can be changed. Type *robot ?* to display a list of the names of available robots. A single simulated robot can also be selected by double-clicking it in the scene view. To select all robots, type *robot all*.

st off | on. Switches the simulation of time on or off. Without the simulation of time, all calls to `SystemCall::getCurrentSystemTime()` will return the real time of the PC. However if the simulator runs slower than real-time, the simulated robots will receive less sensor readings than the real ones. If the simulation of time is switched on, each step of the simulator will advance the time by 20 ms. Thus, the simulator simulates real-time, but it is running slower. By default this option is switched off.

# <text>. Comment. Useful in script files.

8.5.3 Robot Commands

bc <red%> <green%> <blue%>. Defines the background color for 3-D views. The color is specified in percentages of red, green, and blue intensities.

ci off | on. Switches the calculation of images on or off. The simulation of the robot's camera image costs a lot of time, especially if several robots are simulated. In some development situations, it is better to switch off all low level processing of the robots and to work with ground truth world states, i. e., world states that are directly delivered by the simulator. In such cases there is no need to waste processing power by calculating camera images. Therefore, it can be switched off. However, by default this option is switched on. Note that this command only has an effect on simulated robots.

ct off | on | undo | <color> | load <file> | save <file> | send [<ms> | off] | sendAndWrite | clear [<color>] | shrink [<color>] | grow [<color>] | imageRadius <number> | colorSpaceRadius <number> | smart [off]. This command controls editing the current color table. The parameters have the following meaning:

on | off. Activates or deactivates mouse handling in image views. If activated, clicking into an image view modifies the color table (cf. Sect. 8.3.1). Otherwise, mouse clicks are ignored.

undo. Undoes the previous change to the color table. Up to ten steps can be undone. All commands that modify the color table can be undone, including, e. g., *ct clear* and *ct load*.

<color>. Selects the given color as current color class.

(load | save) <file>. Loads or saves the color table. The default directory is the current location. The default extension is *.c64*.

(clear | shrink | grow) [<color>]. Clears, grows, or shrinks a certain color class or all color classes.

send [<ms> | off] | sendAndWrite. Either sends the current color table to the robot, or defines the interval in milliseconds after which the color table is sent to the robot automatically (if it has changed). *off* deactivates the automatic sending of the color table. *sendAndWrite* sends the color table to the robot, which then will write it permanently on its flash drive.

imageRadius <number>. Defines the size of the region surrounding a pixel that is clicked on that will be entered into the color table. 0 results in a 1×1 region, 1 in a 3×3 region, etc. The default is 0.

colorSpaceRadius <number>. Defines the size of the cube that is set to the current color class in the color table for each pixel inserted. 0 results in a $1 \times 1 \times 1$ cube, 1 in a $3 \times 3 \times 3$ cube, etc. The default is 2.

smart [off]. Activates a smart color selection mechanism or deactivates it. This only affects the behavior when selecting a region of the image by mouse. If the mechanism is activated, the simulator adds only colors to the color table within a range around the average color of the selected region. The range can be changed by using the *ct colorSpaceRadius* command. The mechanism is on by default.

dr ? [<pattern>] | off | <key> (off | on | once). Sends a debug request. B-Human uses debug requests to switch *debug responses* on or off at runtime. Type *dr ?* to get a list of all available debug requests. The resulting list can be shortened by specifying a search pattern after the question mark. Debug responses can be activated permanently or only once. They are deactivated by default. Several other commands also send debug requests, e. g., to activate the transmission of debug drawings.

get ? [<pattern>] | <key> [?]. Shows debug data or shows its specification. This command allows displaying any information that is provided in the robot code via the `MODIFY` macro. If one of the strings that are used as first parameter of the `MODIFY` macro is used as parameter of this command (the *modify key*), the related data will be requested from the robot code and displayed. The output of the command is a valid *set* command (see below) that can be changed to modify data on the robot. A question mark directly after the command (with an optional filter pattern) will list all the modify keys that are available. A question mark after a modify key will display the type of the associated data structure rather than the data itself.

jc hide | show | (motion | [press | release] <button>) <command>). Sets a joystick command. If the first parameter is a number, it is interpreted as the number of a joystick button. Legal numbers are between 1 and 32. Any text after this first parameter is part of the second parameter. The *<command>* parameter can contain any legal script command which will be executed in every frame while the corresponding button is pressed. Button numbers may be prefixed with *press* or *release* to restrict the execution to the corresponding event. The commands associated with the 26 first buttons can also be executed by pressing *Ctrl+Shift+A*... *Ctrl+Shift+Z* on the keyboard. If the first parameter is *motion*, the remaining text defines a command that is executed whenever the readings of the analog joystick change. Within this command, $1...$6 can be used as placeholders for up to six joystick axes. The scaling of the values of these axes is defined by the command *js* (see below). If the first parameter is *show*, any command executed will also be printed in the console window. *hide* will switch this feature off again, and *hide* is also the default.

js <axis> <speed> <threshold>. Set axis maximum speed and ignore threshold for command *jc motion*. *axis* is the number of the joystick axis to configure (1...6). *speed* defines the maximum value for that axis, i. e., the resulting range of values will be $[-speed \ldots speed]$. The *threshold* defines a joystick measuring range around zero, in which the joystick will still be recognized as centered, i. e., the output value will be 0. The *threshold* can be set between 0 and 1.

log ? | start | stop | pause | forward | backward | repeat | goto <number> | clear | (keep | remove) <message> {<message>} | (load | save | saveImages [raw]) <file> | cycle | once. The command supports both recording and replaying log files. The latter is only possible if the current set of robot processes was created using the initialization command *sl* (cf. Sect. 8.5.1). The different parameters have the following meaning:

?. Prints statistics on the messages contained in the current log file.

start | stop. If replaying a log file, starts and stops the replay. Otherwise, the commands will start and stop the recording.

pause | forward | backward | repeat | goto <number>. The commands are only accepted while replaying a log file. *pause* stops the replay without rewinding to the beginning, *forward* and *backward* advance a single step in the respective direction, and repeat just resends the current message. *goto* allows jumping to a certain position in the log file.

clear | (keep | remove) <message>. *clear* removes all messages from the log file, while *keep* and *remove* only delete a selected subset based on the set of message ids specified.

(load | save | saveImages [raw]) <file>. These commands *load* and *save* the log file stored in memory. If the filename contains no path, *Config/Logs* is used as default. Otherwise, the full path is used. *.log* is the default extension of log files. It will be automatically added if no extension is given. The option *saveImages* saves only the images from the log file stored in memory to the disk. The default directory is *Config/Images*. They will be stored as BMP files containing either RGB or YCbCr images. The latter is the case if the option *raw* is specified.

cycle | once. The two commands decide whether the log file is only replayed once or continuously repeated.

mof. Recompiles all special actions and if successful, the result is sent to the robot.

mr ? [<pattern>] | modules [<pattern>] | save | <representation> (? [<pattern>] | <module> | default | off). Sends a module request. This command allows selecting the module that provides a certain representation. If a representation should not be provided anymore, it can be switched *off*. Deactivating the provision of a representation is usually only possible if no other module requires that representation. Otherwise, an error message is printed and the robot is still using its previous module configuration. Sometimes, it is desirable to be able to deactivate the provision of a representation without the requirement to deactivate the provision of all other representations that depend on it. In that case, the provider of the representation can be set to *default*. Thus no module updates the representation and it simply keeps its previous state.

A question mark after the command lists all representations. A question mark after a representation lists all modules that provide this representation. The parameter *modules* lists all modules with their requirements and provisions. All three listings can be filtered by an optional pattern. *save* saves the current module configuration to the file *modules.cfg* which it was originally loaded from. Note that this usually has not the desired effect, because the module configuration has already been changed by the start script to be compatible with the simulator. Therefore, it will not work anymore on a real robot. The only configuration in which the command makes sense is when communicating with a remote robot.

msg off | on | log <file>. Switches the output of text messages on or off, or redirects them to a text file. All processes can send text messages via their message queues to the console window. As this can disturb entering text into the console window, printing can be switched off. However, by default text messages are printed. In addition, text messages can be stored in a log file, even if their output is switched off. The file name has to be specified after *msg log*. If the file already exists, it will be replaced. If no path is given, *Config/Logs* is used as default. Otherwise, the full path is used. *.txt* is the default extension of text log files. It will be automatically added if no extension is given.

mv <x> <y> <z> [<rotx> <roty> <rotz>]. Moves the selected simulated robot to the given metric position. x, y, and z have to be specified in mm, the rotations have to be specified in degrees. Note that the origin of the Nao is about 330 mm above the ground, so z should be 330.

poll. Polls for all available debug requests and debug drawings. Debug requests and debug drawings are dynamically defined in the robot control program. Before console commands that use them can be executed, the simulator must first determine which identifiers exist in the code that currently runs. Although the acquiring of this information is usually done automatically, e. g., after the module configuration was changed, there are some situations in which a manual execution of the command *poll* is required. For instance if debug responses or debug drawings are defined inside another debug response, executing *poll* is necessary to recognize the new identifiers after the outer debug response has been activated.

qfr queue | replace | reject | collect <seconds> | save <seconds>. Sends a queue fill request. This request defines the mode how the message queue from the debug process to the PC is handled.

replace is the default mode. If the mode is set to *replace*, only the newest message of each type is preserved in the queue (with a few exceptions). On the one hand, the queue cannot overflow, on the other hand, messages are lost, e. g. it is not possible to receive 30 images per second from the robot.

queue will insert all messages received by the debug process from other processes into the queue, and send it as soon as possible to the PC. If more messages are received than can be sent to the PC, the queue will overflow and some messages will be lost.

reject will not enter any messages into the queue to the PC. Therefore, the PC will not receive any messages.

collect <seconds>. This mode collects messages for the specified number of seconds. After that period of time, the collected messages will be sent to the PC. Since the TCP stack requires a certain amount of execution time, it may impede the real-time behavior of the robot control program. Using this command, no TCP packages are sent during the recording period, guaranteeing real-time behavior. However, since the message queue of the process *Debug* has a limited size, it cannot store an arbitrary number of messages. Hence the bigger the messages, the shorter they can be collected. After the collected messages were sent, no further messages will be sent to the PC until another queue fill request is sent.

save <seconds>. This mode collects messages for the specified number of seconds, and it will afterwards store them on the memory stick as a log file under */media/userdata/Config/logfile.log*. No messages will be sent to the PC until another queue fill request is sent.

set ? [<pattern>] | <key> (? | unchanged | <data>). Changes debug data or shows its specification. This command allows changing any information that is provided in the robot code via the `MODIFY` macro. If one of the strings that are used as first parameter of the `MODIFY` macro is used as parameter of this command (the *modify key*), the related data in the robot code will be replaced by the data structure specified as second parameter. Since the parser for these data structures is rather simple, it is best to first create a valid *set* command using the *get* command (see above). Afterwards that command can be changed before it is executed. If the second parameter is the key word *unchanged*, the related `MODIFY` statement in the code does not overwrite the data anymore, i. e., it is deactivated again. A question mark directly after the command (with an optional filter pattern) will list all the modify keys that are available. A question mark after a modify key will display the type of the associated data structure rather than the data itself.

v3 ? [<pattern>] | <image> [jpeg] [<name>]. Adds a set of 3-D color space views for a certain image (cf. Sect. 8.3.2). The image can either be the camera image (simply specify *image*) or a debug image. It will be JPEG compressed if the option *jpeg* is specified. The last parameter is the name that will be given to the set of views. If the name is not given, it will be the same as the name of the image. A question mark followed by an optional filter pattern will list all available images.

vf <name>. Adds a field view (cf. Sect. 8.3.3). A field view is the means for displaying debug drawings in field coordinates. The parameter defines the *name* of the view.

vfd ? [**<pattern>**] | **<name> (?** [**<pattern>**] | **<drawing> (on** | **off)).**
(De)activates a debug drawing in a field view. The first parameter is the name of a field view that has been created using the command *vf* (see above). The second parameter is the name of a drawing that is defined in the robot control program. Such a drawing is activated when the third parameter is *on* or is missing. It is deactivated when the third parameter is *off*. The drawings will be drawn in the sequence they are added, from back to front. Adding a drawing a second time will move it to the front. A question mark directly after the command will list all field views that are available. A question after a valid field view will list all available field drawings. Both question marks have an optional filter pattern that reduces the number of answers.

vi ? [**<pattern>**] | **<image>** [**jpeg**] [**segmented**] [**<name>**]**.** Adds an image view (cf. Sect. 8.3.1). An image view is the means for displaying debug drawings in image coordinates. The image can either be the camera image (simply specify *image*), a debug image, or no image at all (*none*). It will be JPEG-compressed if the option *jpeg* is specified. If *segmented* is given, the image will be segmented using the current color table. The last parameter is the name that will be given to the set of views. If the name is not given, it will be the same as the name of the image plus the word *Segmented* if it should be segmented. A question mark followed by an optional filter pattern will list all available images.

vid ? [**<pattern>**] | **<name> (?** [**<pattern>**] | **<drawing> (on** | **off)).**
(De)activates a debug drawing in an image view. The first parameter is the name of an image view that has been created using the command *vi* (see above). The second parameter is the name of a drawing that is defined in the robot control program. Such a drawing is activated when the third parameter is *on* or is missing. It is deactivated when the third parameter is *off*. The drawings will be drawn in the sequence they are added, from back to front. Adding a drawing a second time will move it to the front. A question mark directly after the command will list all image views that are available. A question after a valid image view will list all available image drawings. Both question marks have an optional filter pattern that reduces the number of answers.

vp <name> <numOfValues> <minValue> <maxValue> [**<yUnit> <xUnit> <xScale>**]**.** Adds a plot view (cf. Sect. 8.3.7). A plot view is the means for plotting data that was defined by the macro `PLOT` in the robot control program. The first parameter defines the *name* of the view. The second parameter is the number of entries in the plot, i. e. the size of the x axis. The plot view stores the last *numOfValues* data points sent for each plot and displays them. *minValue* and *maxValue* define the range of the y axis. The optional parameters serve the capability to improve the appearance of the plots by adding labels to both axes and by scaling the time-axis. The label drawing can be activated by using the context menu of the plot view.

vpd ? [**<pattern>**] | **<name> (?** [**<pattern>**] | **<drawing> (?** [**<pattern>**] | **<color>** | **off)).** Plots data in a certain color in a plot view. The first parameter is the name of a plot view that has been created using the command *vp* (see above). The second parameter is the name of plot data that is defined in the robot control program. The third parameter defines the color for the plot. The plot is deactivated when the third parameter is *off*. The plots will be drawn in the sequence they were added, from back to front. Adding a plot a second time will move it to the front. A question mark directly after the command will list all plot views that are available. A question after a valid plot view will list all available plot data. Both question marks have an optional filter pattern that reduces the number of answers.

xbb ? [<**pattern**>] | **unchanged** | <**behavior**> {<**num**>}**.** Selects a Xabsl basic behavior. The command suppresses the basic behavior currently selected by the Xabsl engine and replaces it with the behavior specified by this command. Type *xbb ?* to list all available Xabsl basic behaviors. The resulting list can be shortened by specifying a search pattern after the question mark. Basic behaviors can be parameterized by a list of decimal numbers. Use *xbb unchanged* to switch back to the basic behavior currently selected by the Xabsl engine. The command *xbb* only works if the Xabsl symbols have been requested from the robot (cf. Sect. 8.3.4). Note that basic behaviors are not used anymore in the B-Human code.

xis ? [<**pattern**>] | <**inputSymbol**> **(on** | **off).** Switches the visualization of a Xabsl input symbol in the Xabsl view on or off. Type *xis ?* to list all available Xabsl input symbols. The resulting list can be shortened by specifying a search pattern after the question mark. The command *xis* only works if the Xabsl symbols have been requested from the robot (cf. Sect. 8.3.4).

xo ? [<**pattern**>] | **unchanged** | <**option**> {<**num**>}**.** Selects a Xabsl option. The command suppresses the option currently selected by the Xabsl engine and replaces it with the option specified by this command. Options can be parameterized by a list of decimal numbers. Type *xo ?* to list all available Xabsl options. The resulting list can be shortened by specifying a search pattern after the question mark. Use *xo unchanged* to switch back to the option currently selected by the Xabsl engine. The command *xo* only works if the Xabsl symbols have been requested from the robot (cf. Sect. 8.3.4).

xos ? [<**pattern**>] | <**outputSymbol**> **(on** | **off** | **?** [<**pattern**>] | **unchanged** | <**value**> **).** Shows or sets a Xabsl output symbol. The command can either switch the visualization of a Xabsl output symbol in the Xabsl view on or off, or it can suppress the state of an output symbol currently set by the Xabsl engine and replace it with the value specified by this command. Type *xos ?* to list all available Xabsl output symbols. To get the available states for a certain output symbol, type *xos <outputSymbol> ?*. In both cases, the resulting list can be shortened by specifying a search pattern after the question mark. Use *xos <outputSymbol> unchanged* to switch back to the state currently set by the Xabsl engine. The command *xos* only works if the Xabsl symbols have been requested from the robot (cf. Sect. 8.3.4).

xsb. Sends the compiled version of the current Xabsl behavior to the robot.

8.6 Examples

This section presents some examples of script files to automate various tasks:

8.6.1 Recording a Log File

To record a log file, the robot shall send images, joint data, sensor data, key states, odometry data, the camera matrix, and the image coordinate system. The script connects to a robot and configures it to do so. In addition, it prints several useful commands into the console window, so they can be executed by simply setting the caret in the corresponding line and pressing the *Enter* key. As these lines will be printed before the messages coming from the robot, one has to scroll to the beginning of the console window to use them. Note that the file name behind the line *log save* is missing. Therefore, a name has to be provided to successfully execute this command.

```
# connect to a robot
sc Remote 10.1.0.101

# request everything that should be recorded
dr representation:JPEGImage
dr representation:JointData
dr representation:SensorData
dr representation:KeyStates
dr representation:OdometryData
dr representation:CameraMatrix
dr representation:ImageCoordinateSystem

# print some useful commands
echo qfr queue
echo log start
echo log stop
echo log save
echo log clear
```

8.6.2 Replaying a Log File

The example script replays a log file. It instantiates a robot named *LOG* that is fed by the data stored in the log file *Config/Logs/logFile.log*. It also defines some keyboard shortcuts to allow navigating in the log file.

```
# replay a log file.
# you have to adjust the name of the log file.
sl LOG logfile

# select modules for log file replay
mr Image CognitionLogDataProvider
mr CameraInfo CognitionLogDataProvider
mr FrameInfo CognitionLogDataProvider
mr JointData MotionLogDataProvider
mr SensorData MotionLogDataProvider
mr KeyStates MotionLogDataProvider
mr FrameInfo MotionLogDataProvider
mr OdometryData MotionLogDataProvider
mr CameraMatrix CognitionLogDataProvider
mr ImageCoordinateSystem CognitionLogDataProvider

# simulation time on, otherwise log data may be skipped
```

```
st on
msg off

# all views are defined in another script
call Views

# navigate in log file using shortcuts
jc 1 log pause # Shift+Crtl+A
jc 17 log goto 1 # Shift+Crtl+Q
jc 19 log start # Shift+Crtl+S
jc 23 log repeat # Shift+Crtl+W
jc 24 log forward # Shift+Crtl+X
jc 25 log backward # Shift+Crtl+Y
```

8.6.3 Remote Control

This script demonstrates joystick remote control of the robot. The set commands have to be entered in a single line.

```
# connect to a robot
sc Remote 10.1.0.101

# all views are defined in another script
call ViewsJpeg

# request joint data and sensor data
dr representation:SensorData
dr representation:JointData

# request behavior messages
dr automatedRequests:xabsl:debugSymbols once
dr automatedRequests:xabsl:debugMessages

jc press 1 set representation:MotionRequest { motion = specialAction;
    specialActionRequest = { specialAction = kickLeftNao; mirror = false; };
    walkRequest = { speed = { rotation = 0; translation = { x = 0; y = 0; }; };
    target = { rotation = 0; translation = { x = 0; y = 0; }; }; pedantic =
    false; };  kickRequest = { kickType = none; mirror = false; dynamical =
    false; dynPoints[0] = { }; odometryOffset[0] = { }; }; }

jc press 2 set representation:MotionRequest { motion = stand;
    specialActionRequest = { specialAction = kickLeftNao; mirror = false; };
    walkRequest = { speed = { rotation = 0; translation = { x = 0; y = 0; }; };
    target = { rotation = 0; translation = { x = 0; y = 0; }; }; pedantic =
    false; }; kickRequest = { kickType = none; mirror = false; dynamical =
    false; dynPoints[0] = { }; odometryOffset[0] = { }; }; }

js 2 160 0.01 # x axis
js 4 0.5 0.01 # rotation axis
js 1 80 0.01 # y axis

jc motion set representation:MotionRequest { motion = walk; specialActionRequest
    = { specialAction = playDead; mirror = false; }; walkRequest = { speed = {
    rotation = $4; translation = { x = $2; y = $1; }; }; target = { rotation = 0;
    translation = { x = 0; y = 0; }; }; pedantic = false; }; kickRequest = {
```

```
kickType = none; mirror = false; dynamical = false; dynPoints[0] = { };
odometryOffset[0] = { }; }; }
```

Chapter 9

Acknowledgements

We gratefully acknowledge the support given by Aldebaran Robotics. We thank the Deutsche Forschungsgemeinschaft (DFG) for funding parts of our project. Since B-Human did not start its software from scratch, we also want to thank the members of the GermanTeam and of B-Smart for developing parts of the software we use.

In addition, we want to thank the authors of the following software that is used in our code:

AT&T graphviz. For compiling the behavior documentation and for the module view of the simulator.

DotML 1.1. For generating option graphs for the behavior documentation (http://www.martin-loetzsch.de/DOTML).

doxygen. For generating the Simulator documentation (http://www.stack.nl/˜dimitri/doxygen).

flite. For speaking the IP address of the Nao without NaoQi (http://www.speech.cs.cmu.edu/flite/).

RoboCup GameController. For remotely sending game state information to the robot (http://www.tzi.de/spl).

libjpeg. Used to compress and decompress images from the robot's camera (http://www.ijg.org).

XABSL. For implementing the robot's behavior (http://www.informatik.hu-berlin.de/ki/XABSL).

OpenGL Extension Wrangler Library. For determining which OpenGL extensions are supported by the platform (http://glew.sourceforge.net).

GNU Scientific Library. Used by the simulator. (http://david.geldreich.free.fr/dev.html).

libxml2. For reading simulator's scene description files (http://xmlsoft.org).

ODE. For providing physics in the simulator (http://www.ode.org).

protobuf. Used for the SSL Vision network packet-format (http://code.google.com/p/protobuf).

QHull. Calculates the convex hull of simulated objects (http://www.qhull.org).

Qt. The GUI framework of the simulator (http://qt.nokia.com).

zbuildgen. Creates and updates the makefiles and Visual Studio project files.

Bibliography

[1] Jared Bunting, Stephan Chalup, Michaela Freeston, Will McMahan, Rick Middleton, Craig Murch, Michael Quinlan, Christopher Seysener, and Graham Shanks. Return of the nubots! - the 2003 nubots team report. *Technical Report*, 2003.

[2] RoboCup Technical Committee. RoboCup Standard Platform League (Nao) Rule Book. http://www.tzi.de/spl/pub/Website/Downloads/Rules2009.pdf as of May 23, 2009.

[3] Russell C. Eberhart and James Kennedy. A new optimizer using particles swarm theory. In *Sixth International Symposium on Micro Machine and Human Science*, pages 39–43, 1995.

[4] D. Fox, W. Burgard, F. Dellaert, and S. Thrun. Monte Carlo Localization: Efficient Position Estimation for Mobile Robots. In *Proc. of the National Conference on Artificial Intelligence*, 1999.

[5] Emden R. Gansner and Stephen C. North. An open graph visualization system and its applications to software engineering. *Software Practice and Experience*, 30, 2000.

[6] Keyan Ghazi-Zahedi, Tim Laue, Thomas Röfer, Peter Schöll, Kai Spiess, Arndt Twickel, and Steffen Wischmann. Rosiml - robot simulation markup language, 2005. http://www.informatik.uni-bremen.de/spprobocup/RoSiML.html.

[7] J.-S. Gutmann and D. Fox. An experimental comparison of localization methods continued. *Proceedings of the 2002 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2002.

[8] L.J. Heyer, S. Kruglyak, and S. Yooseph. Exploring expression data: identification and analysis of coexpressed genes, 1999.

[9] V. Jagannathan, R. Dodhiawala, and L. Baum. *Blackboard Architectures and Applications.* Academic Press, Inc., 1989.

[10] S. J. Julier, J. K. Uhlmann, and H. F. Durrant-Whyte. A new approach for filtering nonlinear systems. In *American Control Conference, 1995. Proceedings of the*, volume 3, pages 1628–1632, 1995.

[11] J. Kulk and J. Welsh. A low power walk for the nao robot. In *Proceedings of ACRA*, 2008.

[12] T. Laue and T. Röfer. Getting upright: Migrating concepts and software from four-legged to humanoid soccer robots. In E. Pagello, C. Zhou, and E. Menegatti, editors, *Proceedings of the Workshop on Humanoid Soccer Robots in conjunction with the 2006 IEEE International Conference on Humanoid Robots*, 2006.

[13] T. Laue and T. Röfer. Particle filter-based state estimation in a competitive and uncertain environment. In *Proceedings of the 6th International Workshop on Embedded Systems.* VAMK, University of Applied Sciences; Vaasa, Finland, 2007.

[14] T. Laue and T. Röfer. Simrobot - development and applications. In H. B. Amor, J. Boedecker, and O. Obst, editors, *The Universe of RoboCup Simulators - Implementations, Challenges and Strategies for Collaboration. Workshop Proceedings of the International Conference on Simulation, Modeling and Programming for Autonomous Robots (SIMPAR 2008)*, Lecture Notes in Artificial Intelligence. Springer, 2008.

[15] T. Laue and T. Röfer. Pose extraction from sample sets in robot self-localization - a comparison and a novel approach. In *Proc. of the 4th European Conference on Mobile Robots (ECMR 2009)*, Mlini/Dubrovnik, Croatia, 2009.

[16] T. Laue, K. Spiess, and T. Röfer. SimRobot - A General Physical Robot Simulator and Its Application in RoboCup. In A. Bredenfeld, A. Jacoff, I. Noda, and Y. Takahashi, editors, *RoboCup 2005: Robot Soccer World Cup IX*, number 4020 in Lecture Notes in Artificial Intelligence, pages 173–183. Springer, 2006.

[17] S. Lenser and M. Veloso. Sensor resetting localization for poorly modeled mobile robots. In *Proc. of the IEEE International Conference on Robotics and Automation (ICRA)*, 2002.

[18] Martin Loetzsch, Max Risler, and Matthias Jüngel. XABSL - A pragmatic approach to behavior engineering. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2006)*, pages 5124–5129, Beijing, October 2006.

[19] Thomas Röfer, Tim Laue, Armin Burchardt, Erik Damrose, Katharina Gillmann, Colin Graf, Thijs Jeffry de Haas, Alexander Härtl, Andrik Rieskamp, André Schreck, and Jan-Hendrik Worch. B-human team report and code release 2008, 2008. Only available online: http://www.b-human.de/download.php?file=coderelease08_doc.

[20] T. Röfer. Region-based segmentation with ambiguous color classes and 2-D motion compensation. In U. Visser, F. Ribeiro, T. Ohashi, and F. Dellaert, editors, *RoboCup 2007: Robot Soccer World Cup XI*, Lecture Notes in Artificial Intelligence. Springer.

[21] T. Röfer, J. Brose, D. Göhring, M. Jüngel, T. Laue, and M. Risler. Germanteam 2007. In U. Visser, F. Ribeiro, T. Ohashi, and F. Dellaert, editors, *RoboCup 2007: Robot Soccer World Cup XI Preproceedings*. RoboCup Federation, 2007.

[22] T. Röfer, T. Laue, and D. Thomas. Particle-filter-based self-localization using landmarks and directed lines. In A. Bredenfeld, A. Jacoff, I. Noda, and Y. Takahashi, editors, *RoboCup 2005: Robot Soccer World Cup IX*, number 4020 in Lecture Notes in Artificial Intelligence, pages 608–615. Springer, 2006.

[23] T. Röfer, T. Laue, M. Weber, H.-D. Burkhard, M. Jüngel, D. Göhring, J. Hoffmann, B. Altmeyer, T. Krause, M. Spranger, O. v. Stryk, R. Brunn, M. Dassler, M. Kunz, T. Oberlies, M. Risler, and etc. Germanteam robocup 2005, 2005. Only available online: http://www.germanteam.org/GT2005.pdf.

[24] D. v. Heesch. Doxygen – source code documentation generator tool. http://www.stack.nl/~dimitri/doxygen/, viewed Oct 28, 2008.

[25] Stefan Zickler, Tim Laue, Oliver Birbach, Mahisorn Wongphati, and Manuela Veloso. SSL-vision: The shared vision system for the RoboCup Small Size League. In Jacky Baltes, Michail G. Lagoudakis, Tadashi Naruse, and Saeed Shiry, editors, *RoboCup 2009: Robot Soccer World Cup XIII*, Lecture Notes in Artificial Intelligence. Springer, to appear in 2010.