B-Human

Team Report and Code Release 2010

Thomas Röfer[1], Tim Laue[1], Judith Müller[1], Armin Burchardt[2], Erik Damrose[2], Alexander Fabisch[2], Fynn Feldpausch[2], Katharina Gillmann[2], Colin Graf[2], Thijs Jeffry de Haas[2], Alexander Härtl[2], Daniel Honsel[2], Philipp Kastner[2], Tobias Kastner[2], Benjamin Markowsky[2], Michael Mester[2], Jonas Peter[2], Ole Jan Lars Riemann[2], Martin Ring[2], Wiebke Sauerland[2], André Schreck[2], Ingo Sieverdingbeck[2], Felix Wenk[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 1, 2010

Contents

1 Introduction **7**

1.1 About us . . . 7

1.2 About the Document . . . 8

1.3 Changes Since 2009 . . . 9

2 Getting Started **11**

2.1 Unpacking . . . 11

2.2 Components and Configurations . . . 12

2.3 Compiling using Visual Studio 2008 . . . 13

2.3.1 Required Software . . . 13

2.3.2 Compiling . . . 13

2.4 Compiling on Linux . . . 13

2.4.1 Required Software . . . 13

2.4.2 Compiling . . . 14

2.5 Configuration Files . . . 14

2.6 Setting Up the Nao . . . 17

2.6.1 Requirements . . . 17

2.6.2 Creating Robot Configuration . . . 17

2.6.3 Wireless Configuration . . . 17

2.6.4 Setup . . . 18

2.7 Copying the Compiled Code . . . 18

2.7.1 Using copyfiles . . . 18

2.7.2 Using teamDeploy . . . 19

2.8 Working with the Nao . . . 20

2.9 Starting SimRobot . . . 21

3 Architecture **22**

3.1 Binding . . . 22

3.2 Processes . . . 23

3.3 Modules and Representations . . . 24

3.3.1 Blackboard . . . 24
3.3.2 Module Definition . . . 24
3.3.3 Configuring Providers . . . 26
3.3.4 Pseudo-Module *default* . . . 26
3.4 Streams . . . 26
3.4.1 Streams Available . . . 27
3.4.2 Streaming Data . . . 28
3.4.3 Making Classes Streamable . . . 29
3.5 Communication . . . 30
3.5.1 Message Queues . . . 30
3.5.2 Inter-process Communication . . . 31
3.5.3 Debug Communication . . . 32
3.5.4 Team Communication . . . 32
3.6 Debugging Support . . . 33
3.6.1 Debug Requests . . . 33
3.6.2 Debug Images . . . 33
3.6.3 Debug Drawings . . . 35
3.6.4 Plots . . . 36
3.6.5 Modify . . . 36
3.6.6 Stopwatches . . . 37

4 Cognition **38**
4.1 Perception . . . 38
4.1.1 Definition of Coordinate Systems . . . 38
4.1.1.1 Camera Matrix . . . 39
4.1.1.2 Image Coordinate System . . . 41
4.1.2 BodyContourProvider . . . 41
4.1.3 Image Processing . . . 42
4.1.3.1 Segmentation and Region-Building . . . 43
4.1.3.2 Region Classification . . . 45
4.1.3.3 Detecting Lines . . . 47
4.1.3.4 Detecting the Goal . . . 49
4.1.3.5 Detecting the Ball . . . 50
4.1.3.6 Detecting Other Robots . . . 51
4.2 Modeling . . . 52
4.2.1 Self-Localization . . . 52
4.2.2 Robot Pose Validation . . . 53
4.2.3 Ball Tracking . . . 54

4.2.4 Ground Truth . . . 55
4.2.5 Obstacle Model . . . 56
4.2.6 Robot Tracking . . . 57
4.2.7 Largest Free Part of the Opponent Goal . . . 59

5 Motion **60**
5.1 Sensing . . . 61
5.1.1 Joint Data Filtering . . . 61
5.1.2 Ground Contact Recognition . . . 61
5.1.3 Robot Model Generation . . . 62
5.1.4 Inertia Sensor Data Calibration . . . 62
5.1.5 Inertia Sensor Data Filtering . . . 63
5.1.6 Torso Matrix . . . 63
5.1.7 Detecting a Fall . . . 64
5.2 Motion Control . . . 65
5.2.1 Motion Selection . . . 66
5.2.2 Head Motions . . . 66
5.2.3 Walking . . . 66
5.2.3.1 Inverse Kinematic . . . 68
5.2.4 Special Actions . . . 71
5.2.5 Motion Combination . . . 72

6 Behavior Control **74**
6.1 XABSL . . . 75
6.2 Setting Up a New Behavior . . . 78
6.3 Behavior Used at RoboCup 2010 . . . 80
6.3.1 Button Interface . . . 80
6.3.2 Body Control . . . 81
6.3.3 Head Control . . . 81
6.3.4 Kick Pose Provider . . . 83
6.3.5 Tactics . . . 83
6.3.6 Role Selector . . . 84
6.3.7 Different Roles . . . 86
6.3.7.1 Striker . . . 86
6.3.7.2 Supporter . . . 90
6.3.7.3 Defender . . . 92
6.3.7.4 Keeper . . . 93
6.3.8 Penalty Control . . . 94

6.3.9 Display Control . . . 95
6.3.9.1 Right Eye . . . 95
6.3.9.2 Left Eye . . . 95
6.3.9.3 Torso (Chest Button) . . . 95
6.3.9.4 Feet . . . 95
6.3.9.5 Ears . . . 95

7 Challenges **97**
7.1 Dribble Challenge . . . 97
7.2 Passing Challenge . . . 98
7.3 Open Challenge . . . 99

8 SimRobot **101**
8.1 Introduction . . . 101
8.2 Scene View . . . 101
8.3 Information Views . . . 101
8.3.1 Image Views . . . 102
8.3.2 Color Space Views . . . 103
8.3.3 Field Views . . . 104
8.3.4 Xabsl View . . . 105
8.3.5 Sensor Data View . . . 106
8.3.6 Joint Data View . . . 106
8.3.7 Plot Views . . . 106
8.3.8 Timing View . . . 107
8.3.9 Module Views . . . 107
8.3.10 Kick View . . . 107
8.4 Scene Description Files . . . 110
8.5 Console Commands . . . 111
8.5.1 Initialization Commands . . . 111
8.5.2 Global Commands . . . 112
8.5.3 Robot Commands . . . 112
8.5.4 Input Selection Dialog . . . 119
8.6 Examples . . . 119
8.6.1 Recording a Log File . . . 119
8.6.2 Replaying a Log File . . . 120
8.6.3 Remote Control . . . 121

9 Acknowledgements **123**

Bibliography **125**

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 parallel to these activities, B-Human started as a part of the joint team BreDoBrothers that 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.

We repeated our successes in 2010 and won the German Open with a goal ratio of 54:2 as well

Figure 1.1: The majority of the team members at the German Open 2010 awards ceremony.

as the RoboCup with an overall goal ratio of 65:3.

The current team consists of the following persons:

Students. Alexander Fabisch, Arne Humann, Benjamin Markowsky, Carsten Könemann, Daniel Honsel, Emil Huseynli, Felix Wenk, Fynn Feldpausch, Jonas Peter, Martin Ring, Max Trocha, Michael Mester, Ole Jan Lars Riemann, Philipp Kastner, Bastian Reich, Thomas Liebschwager, Tobias Kastner, Wiebke Sauerland.

Senior Students. Alexander Härtl, Armin Burchardt, Colin Graf, Ingo Sieverdingbeck, Katharina Gillmann, Thijs Jeffry de Haas.

Researchers. Tim Laue, Judith Müller.

Senior Researcher. Thomas Röfer (team leader).

1.2 About the Document

As we wanted to revive the tradition of an annual code release two years ago, it is obligatory for us to continue with it this year. This document, which is partially based on last year's code release [23], gives a survey about the evolved system we used at RoboCup 2010. The changes made to code which was used at RoboCup 2009 are shortly enumerated in 1.3.

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

RoboCup 2010[1] 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.

1.3 Changes Since 2009

The changes made since RoboCup 2009 are described in the following sections:

4.1.3.4 Detecting the Goal
We developed a new GoalPerceptor that works on the image instead of the *RegionPercept* and thus is more efficient.

4.1.3.5 Detecting the Ball
In comparison to the BallPerceptor of the last year's code release, the new BallPerceptor is more independent of color tables, although not completely. In addition, it is more efficient and easier to maintain.

4.1.3.6 Detecting Other Robots
We are developing a robot detection via vision which we did not have last year.

4.2.2 Robot Pose Validation
We introduced a new module to refine the localization provided by the SelfLocator.

4.2.3 Ball Tracking
The new BallLocator uses multiple Kalman filters instead of a Particle filter.

4.2.5 Obstacle Model
The ObstacleModel incorporates vision information in addition to ultrasonic measurements.

4.2.6 Robot Tracking
To use the robots detected in our behavior, we implemented a Kalman filter that tracks the robots that are recognized through vision.

4.2.7 Largest Free Part of the Opponent Goal
The module is based on the visual robot detection and provides the part of the opponent goal to shoot at.

5.1.7 Detecting a Fall
To reduce the number of broken joints, we are now preparing a robot for the impact on the ground, i. e. we lower the power of the joints and turn the head.

5.2.3 Walking
This year's WalkingEngine is a further development of last year's approach with an improved method for controlling the center of mass motion and altered usage of sensor feedback, which almost doubled the walking speed.

6.3 Behavior Used at RoboCup 2010
The changes in this year's behavior are composed of new features (mainly the introduction of tactics, hand-to-hand handling, a new role assignment, a completely revised supporter, and a new way of approaching the ball) as well as using new modules replacing existing parts of the behavior, namely the KickPoseProvider and the kick engine BIKE.

[1] 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.

6.3.4 Kick Pose Provider

The best robot pose to kick the ball will be calculated by the KickPoseProvider.

7.1 Dribble Challenge

Our approach to solve the dribble challenge is mainly based on the obstacle model.

7.3 Open Challenge

We presented an approach to throw the ball back in that is fully integrated in our behavior.

8.3.10 Kick View

We developed a dynamic kick engine, the user interface of which is described in this report. The general principles of the engine are published in [20].

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, for example with `tar -xkf bhuman10_coderelease.tar.bz2`. Using Windows, it is recommended to unpack the archive using the *tar* program that is delivered with Cygwin (cf. Sect. 2.3.1). To use the B-Human software on a Nao, some libraries and headers from the Aldebaran SDK are required. These libraries can be installed automatically with the *Install/alcommonInstall.sh* script. It can be executed with Linux or Cygwin (cf. Sect. 2.3.1 and Sect. 2.4.1).

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 and is initially missing. It is created by the build system and can be removed manually by the user if desired.

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.

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/BH2010StableBehaviorControl*.

SimulatorDoc is a tool for creating the documentation of the complete simulator source code. The results will be located in *Doc/Reference/Simulator*. The generation takes a plenty of time and space due to the call graphs created with dot. Feel free to edit the configuration file in *Make/Documentation/Simulator_Documentation.cfg* if you do not require the graphs.

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*, *OptimizedWithoutAssertions*, 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. If you want to disable assertions as in *Release* but enable debugging support, *OptimizedWithoutAssertions* can be used.

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 – 1.7 with the following additional packages: make, ruby, rsync, openssh, libxml2, libxslt. Add the *...\cygwin\bin* directory to the PATH environment variable. (`http://www.cygwin.com`)
- gcc, glibc – Linux cross compiler for Cygwin, download from `http://www.b-human.de/file_download/25/bhuman-cygwin-gcc-linux-gcc-4.1.1-glibc-2.3.6-tls-boost-1.38.0-python-2.5.tar.bz2`, use a Cygwin shell to extract in order to keep symbolic links . 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 – For the extraction of the required alcommon library and compatible boost headers from the *Nao SDK release v1.6.13 linux* (*aldebaran-sdk-1.6.13-linux-i386.tar.gz*) the script *Install/alcommonInstall.sh* can be used, which is delivered with the B-Human software. The required package has to be downloaded manually and handed over to the script. It 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 click on the project to be built (usually *_Simulator* or *_Nao* or *Copdyfiles*) and choose *Build/Build Project* in the menu bar or just *Build* in the project's context menu. Select *_Simulator* as start project. You can also select *Build/Build Solution* to build everything including the documentation, but this will take very long.

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 Lucid Lynx):

- g++, make
- libqt4-dev – 4.3 or above (qt.nokia.com)

- ruby – 1.8
- 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 – For the extraction of the required alcommon library and compatible boost headers from the *Nao SDK release v1.6.13 linux* (*aldebaran-sdk-1.6.13-linux-i386.tar.gz*) the script *Install/alcommonInstall.sh* can be used, which is delivered with the B-Human software. The required package has to be downloaded manually and handed over to the script. It 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 [CONFIG=<configuration>]
```

2.5 Configuration Files

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

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

cameraCalibrator.cfg contains parameters for the CameraCalibrator (cf. Sect. 4.1.1.1).

fallDownStateDetector.cfg contains parameters for the FallDownStateDetector (cf. Sect. 5.1.7).

groundContact.cfg contains parameters for the GroundContactDetector (cf. Sect. 5.1.2).

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).

robotPerceptor.cfg contains parameters for the RobotPerceptor (cf. Sect. 4.1.3.6) and the RobotLocator (cf. Sect. 4.2.6).

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).

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>/behaviorParameters.cfg and bh2010Parameters.cfg both files contain frequently used parameters of the behavior which can be modified. The goal is an easily adjustable behavior.

Locations/<location>/behaviorTactics.cfg can be used in behavior programming. It contains three parameters to change tactics fast and easily between games or during half times.

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 that is used by the SelfLocator (cf. Sect. 4.2.1) and can also be generated by that module. It contains look-up tables for mapping perceptions to lines and line crossings.

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.

Processes contains two files that list all modules that belong to either the process *Cognition* (*CognitionModules.cfg*) or *Motion* (*MotionModules.cfg*) (cf. Sect. 3.2). Whenever a new module is added to the system, its name must be added to one of these files (cf. Sect. 3.3). The third file *connect.cfg* describes the inter-process communication connections used by our framework in a format that was originally used by the Sony AIBO. There is no need to ever change that file.

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. There also is a directory called *Default*. Whenever a file needed is not found in the directory of the current robot, the corresponding file from the directory *Default* is used.

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. 5.2.3.1).

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

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

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 v1.6.13* that is available at the internal RoboCup download area. Unpack 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-1.6.13-nao-geode.ext3* or *opennao-robocup-1.6.13-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 1.6.13.

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*.

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

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

createNewRobot.sh -t 3 -r 25 Penny

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* to *Install/file/wpa_supplicant.conf_default* and change it in the way needed for your wireless network. The name of the new configuration must be *wpa_supplicant.conf_<suffix>* in order to use this file with our install script. To use a suffix other than *default*, change the variable *wlanConfigSuf-*

fix 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 *wpa_supplicant.conf* files found 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. Notice that the script assumes that you have placed an OS image named *opennao-robocup-1.6.13-nao-geode.ext3.bz2* and an appropriate patition table with the same name but a *.parttable* extension in *Install/images* if you do not specify an alternative image file with *-i*.

Start your Nao and wait until the boot finished. Connect via SSH to the Nao and log in as root with the password *cr2010*. 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

2.7.1 Using copyfiles

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 all configurations. However, for the Nao code, building *DebugWithReleaseLibs* results in building the configuration *Optimized*. *OptimizedWithoutAssertions* is not supported in Visual Studio yet. If the code is not up-to-date in the desired configuration, it will be built. After a successful build, you will be prompted for entering the parameters described below. On the other hand you can just execute

the file *copyfiles.cmd* located in the directory *Make* at the command prompt.

Running Linux you have to execute the file *copyfiles.sh*, which is also located in the *Make* directory. Actually, also all other ways described to deploy files to the Nao finally result in executing this script.

copyfiles requires two mandatory parameters. First, the configuration the code was compiled with (*Debug*, *Optimized*, *OptimizedWithoutAssertions* 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, replacing the value in the *settings.cfg*.
-t <color>	Sets the team color to *blue* or *red*, replacing the value in the *settings.cfg*.
-p <number>	Sets the player number, replacing the value in the *settings.cfg*.
-d	Deletes the local cache and the target directory.
-m n <ip>	Copies to IP address <ip> and sets the player number to *n*.

Possible calls could be:

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

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

2.7.2 Using teamDeploy

copyfiles works quite well if you copy code to three or four robots, which are playing in one team. If you want to prepare more robots, which are playing in more teams, you can use *teamDeploy*. It reads a configuration from a single file and executes *copyfiles* for each robot correspondingly to the configuration in the file.

The tool requires the path to a team configuration file as mandatory parameter. Optionally it accepts the parameters `-c` and `-t`. If `-c` is set, it is only checked whether the file provided contains a valid team configuration. By default, code is copied to every robot of every team that is specified in the team configuration file. With `-t` you can name the team to which code should be copied to. If one or more robot names are given, code is only copied to these robots. `-t` can be used multiple times to specify several teams or robots, which should be supplied with code.

Exemplary invocations of *teamDeploy*:

```
teamDeploy.sh Config/teamDeploy.template.tc -c
teamDeploy.sh Config/teamDeploy.template.tc -t B-Human
teamDeploy.sh Config/teamDeploy.template.tc -t B-Human Nao1
teamDeploy.sh Config/teamDeploy.template.tc -t B-Human Nao1 -t all Nao3
```

teamDeploy uses the *SafeConfigParser* class of the python standard library to parse a simple, *ini* style configuration file such as the example file *Config/teamDeploy.template.tc*. As a result, every line with a leading character `#` is ignored. Each file can contain multiple named sections. Sections with a leading *"Team:"* define teams and sections with a leading *"Robot:"* can be used to define several robots. In addition, a "DEFAULT" can be used to define variables that can be referenced later in the file by `%(variableName)`.

Attributes available for a team:

location. This is the location a team should play with. Use one of the locations specified in *Config/Locations*.

color. This is the color of the team (*red* or *blue*).

number. This is the number of the team.

port. *(optional)* This is the UDP port that is used for the team communication. If the port is omitted it is generated from the team number with a leading *10* and a trailing *01*.

robots. This is a comma separated list of robots that should "play together" in the team. If *teamDeploy's* ability of automatic player number generation is used, the order of this list affects the player numbers assigned to the robots.

wlanConfig. *(optional)* Here you can specify an alternative wireless configuration. The configuration you choose must be defined in the file *Install/files* and has to be present on the robots of the team. If this attribute is omitted, the wireless configuration is not changed. It is recommended to change the wireless configuration only if you are connected by Ethernet.

Attributes available for a robot:

ip. This is the IP address of the robot. Only the decimal notation is supported.

build. This has to be one of the build configurations, mentioned in Section 2.7.

colorTable. *(optional)* This can be a file with an alternative color table as present in one of the location directories. If the file is not present in the chosen location it will be searched in `Default`. The extension *c64* can be omitted.

player. *(optional)* This can be a player number of the robot. If the number of one robot in a team is omitted, *teamDeploy* numbers the robots consecutively.

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.

To connect to the Nao, the directory *Make* contains the two scripts *login.cmd* and *login.sh* for Windows and Linux, respectively. The only parameter of those scripts is the IP address of the robot to login. The scripts automatically use the appropriate SSH key to login and suppress ssh's complaints about using the same host key for all robots. In addition, they write the IP address specified to the file *Config/Scenes/connect.con*. Thus a later use of the SimRobot scene *RemoteRobot.ros* will automatically connect to the same robot (cf. next section).

On the Nao, */home/root* contains scripts to start and stop *NaoQi* via SSH:

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

./naoqi.sh 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 if you have not used one of the two login scripts described in the previous section. 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 a configuration other than *Release*.

See Chapter 8 for more detailed information about SimRobot.

[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 [22], 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.

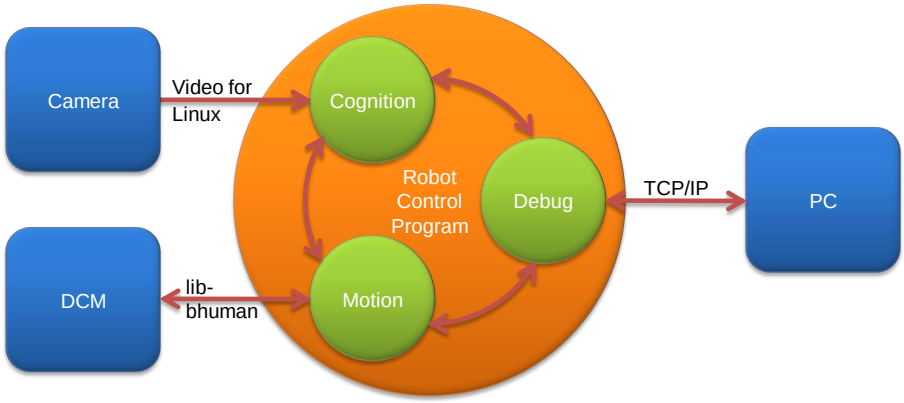


Figure 3.1: The processes used on the Nao

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 the 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 30 Hz and accepts new joint angles at 100 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 through the *libbhuman* 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 [22] 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 [7] 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.

All inter-process communication is triple-buffered. Thus, processes never block each other, because they never access the same memory blocks at the same time. In addition, a receiving process always gets the most current version of a packet sent by another 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 or Ethernet 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. In contrast to all other debugging features, the textual identifier used is not enclosed in double quotes. It must only be comprised of characters that are legal in C++ identifiers.

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);
WalkingEngine::Parameters 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. 4.9). 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.

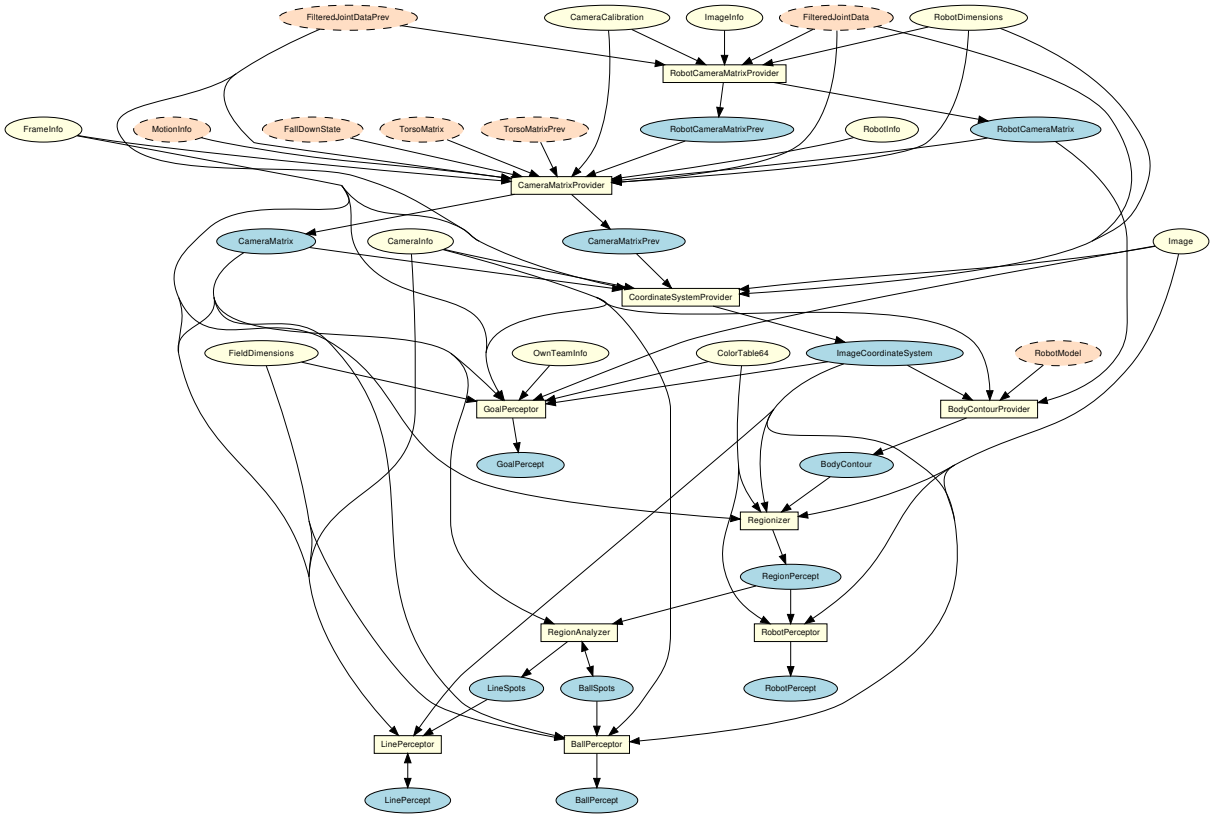


Figure 4.1: Perception module graph

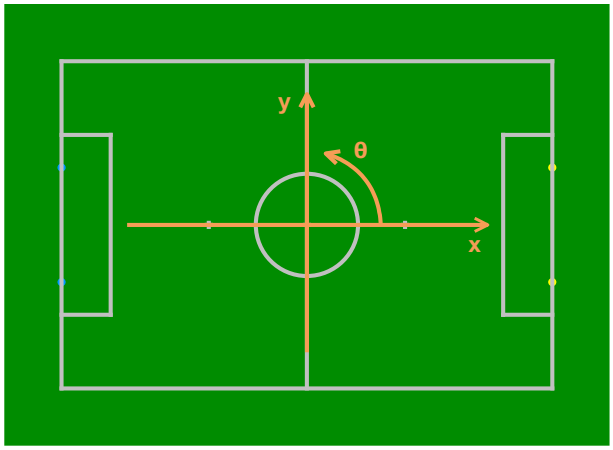


Figure 4.2: Visualization of the global coordinate system

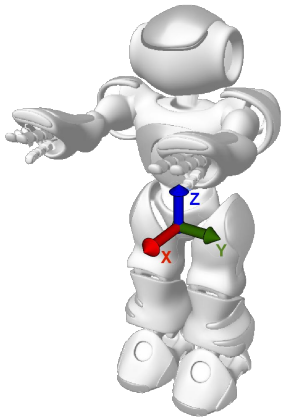


Figure 4.3: Visualization of the robot-relative coordinate system

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 *TorsoMatrix* that represents the orientation and position of a specific point within the robot's torso relative to the ground (cf. Sect. 5.1.6). Using the *RobotDimensions* and the current position of the joints the transformation of the camera matrix relative to the torso matrix is computed. This relative transformation matrix is also provided as *RobotCamera-*

Matrix, 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 in the camera's orientation 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).

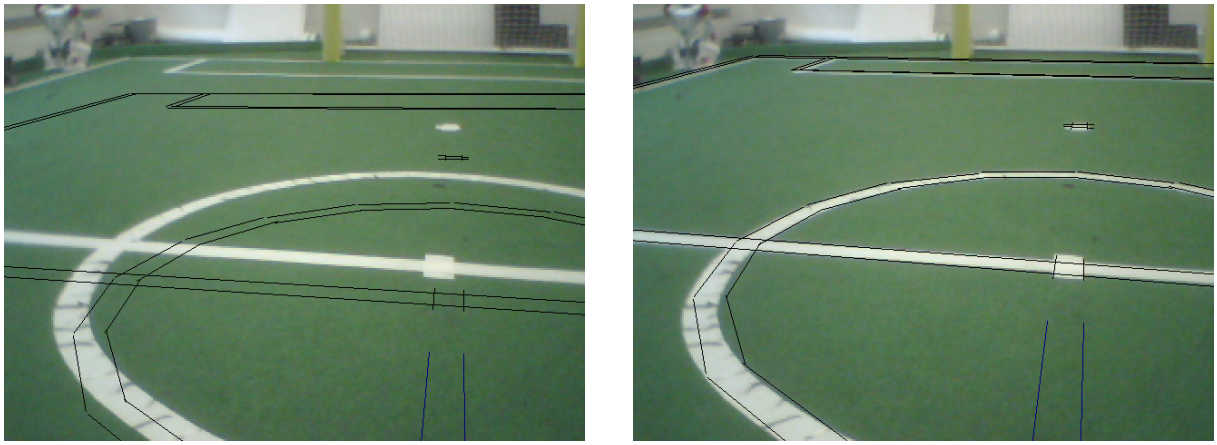


Figure 4.4: Camera calibration using *calibrationHelper*: Projected lines before (left) and after (right) the calibration procedure

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.

Another way of calibrating the robot-specific parameters mentioned above is provided by the `CameraCalibrator` module. A simple hill climbing algorithm was used to implement a semi-automatic calibration procedure. The mean error of the field corner projections is minimized, after they have been marked by the user (see 4.5). The visible field corners can be configured in the `cameraCalibrator.cfg` configuration file.

Then, the calibration procedure is as follows:

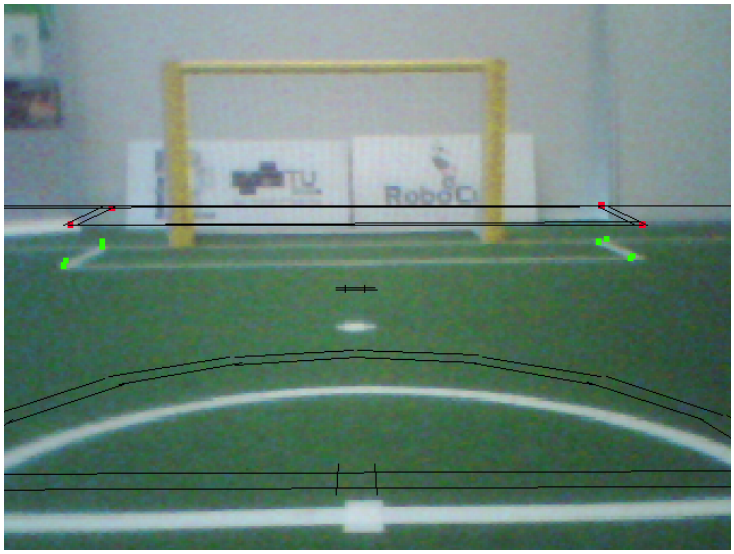


Figure 4.5: Camera calibration using CameraCalibrator: Projected lines during calibration procedure

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 *CameraCalibrator.con* (in the console type `call CameraCalibrator`).
4. Start the setup procedure using *dr module:cameraCalibrator:setup once*.
5. Run the automatic calibration process using *dr module:cameraCalibrator:start once*[1].

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 CoordinateSystemProvider. Its origin of the y-coordinate lies on the horizon within the image (even if it is not visible in the image). The x-axis points right along the horizon whereas the y-axis points downwards orthogonal to the horizon (cf. Fig. 4.6). For more information see also [25]. To get a got estimate of the current motion of the camera, the process *Motion* not only provides the current *TorsoMatrix* to process *Cognition*, but also the *TorsoMatrixPrev* that is 10 ms older. Therefrom, the *CameraMatrixPrev* is computed that enables the CoordinateSystemProvider to determine the rotation speed of the camera and thereby interpolate its orientation when recording each image row. As a result, the representation *ImageCoordinateSystem* provides a mechanism to compensate for different recording times of images and joint angles as well as for image distortion caused by the rolling shutter. For a detailed description of this method applied to the Sony AIBO see [21].

4.1.2 BodyContourProvider

If the robot sees parts of its body, it might confuse white areas with field lines. However, by using forward kinematics, the robot can actually know where its body is visible in the camera image

[1] It is advisable to switch off images during this process (in the console type *dr representation:Image off* / *dr representation:JPEGImage off*)

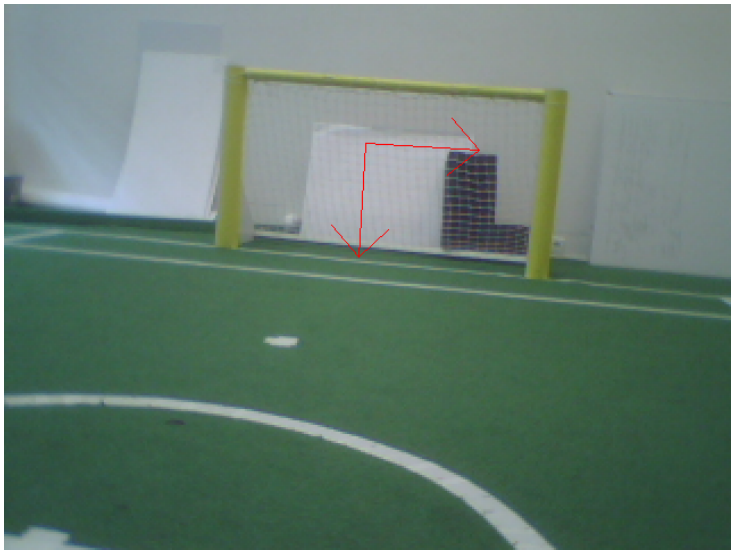


Figure 4.6: Origin of the *ImageCoordinateSystem*

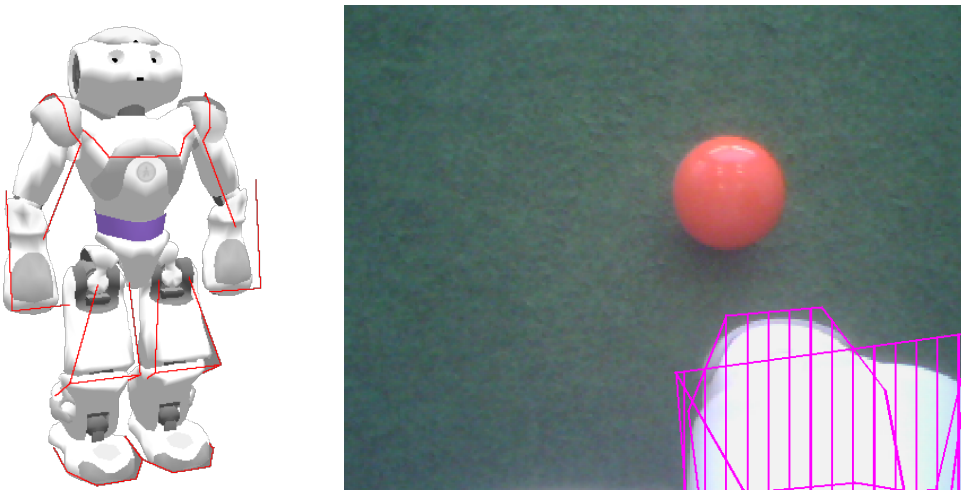


Figure 4.7: Body contour in 3-D (left) and projected to the camera image (right).

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. 4.7 left) and projecting them back to the camera image (cf. Fig. 4.7 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.

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.

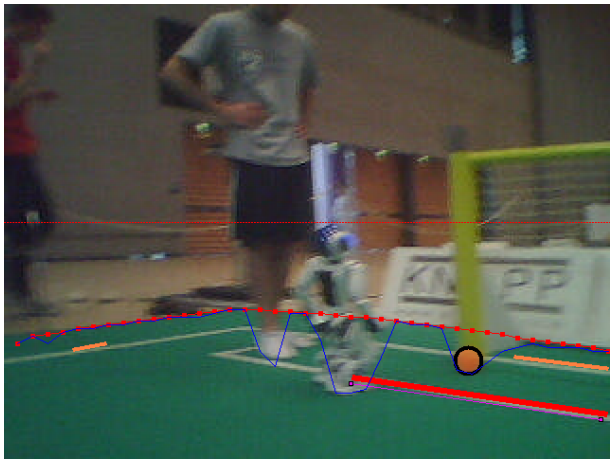


Figure 4.8: Field borders: the small red-dotted line is the horizon, the blue line connects the green points found, the red line is the upper part of the convex hull with a dot for each scan line, the ball in front of the post would be missed without the convex hull.

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. 4.8). 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. 4.9). Next to the usual scanlines there are used some scanlines between those in case of the ball was not seen for a certain time. These scanlines only recognize only orange regions. In this way we also can find balls even if the distance from the robot is that large that balls would disappear between usual scanlines.

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 ← NULL
firstInColumn ← NULL
s' ← NULL
for all s ∈ segments do
  if s.color = green then
    continue
  end if
  if column(s) ≠ column(s') then
    lastColumnPointer ← firstInColumn
    firstInColumn ← s
  end if
  lastColumnPointer ← connectToRegions(s, lastColumnPointer)
  s' ← 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)
- 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. 4.9). 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.

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 ← lastColumnPointer.next()
end while
if column(lastColumnPointer) + 1 ≠ column(s) then
  createNewRegion(s)
  return NULL
end if
if lastColumnPointer.start ≤ s.exploredEnd & lastColumnPointer.color = s.color then
  uniteRegions(lastColumnPointer, s)
end if
lastColumnPointer′ ← lastColumnPointer
while lastColumnPointer′.end ≤ s.exploredEnd do
  lastColumnPointer′ ← lastColumnPointer′.next()
  if column(lastColumnPointer′) + 1 ≠ column(s) then
    if !s.hasRegion then
      createNewRegion(s)
    end if
    return lastColumnPointer
  end if
  if lastColumnPointer′.start ≤ 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
```

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

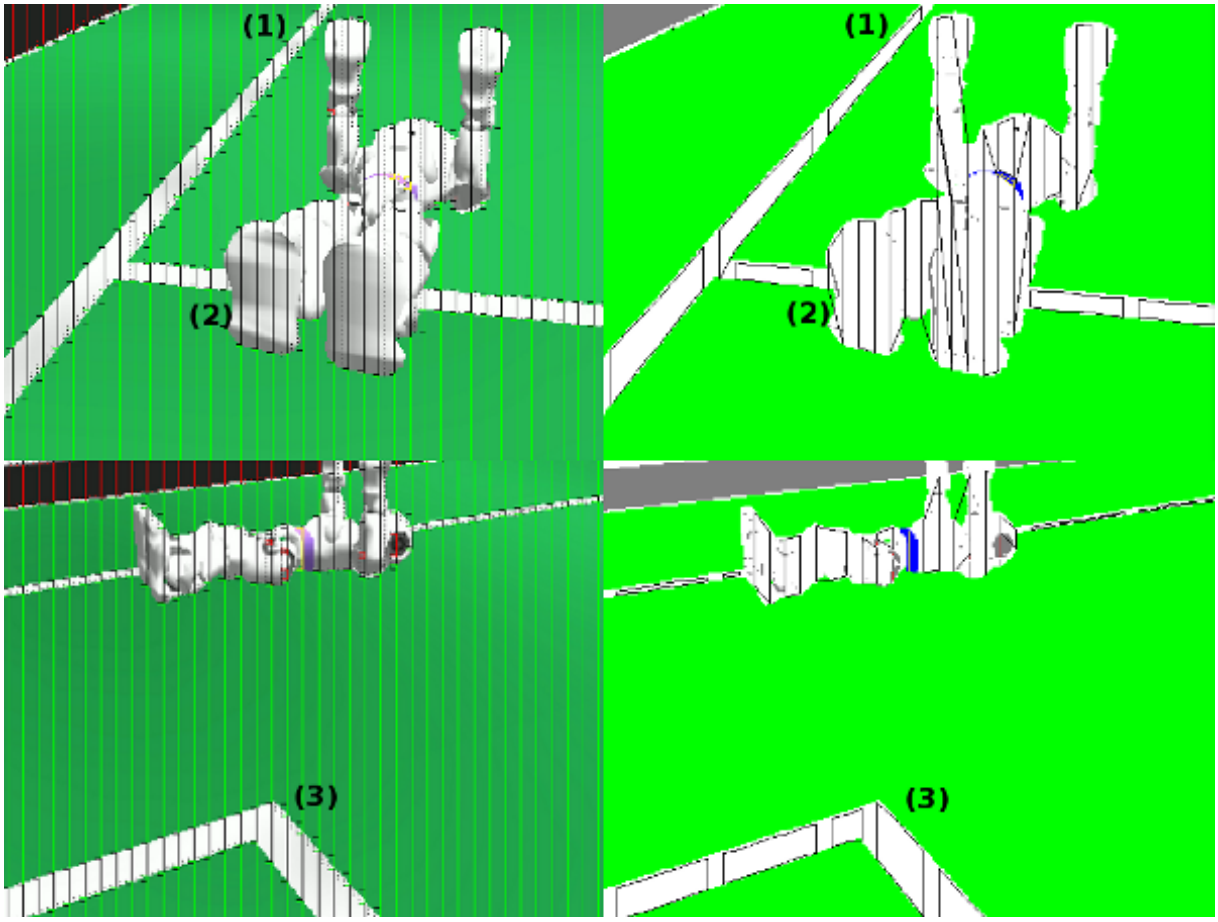


Figure 4.9: Segmentation and region-building: (1) segment is not touching the other, but the explored runs (dotted lines) are touching the other segment, (2) not connected because of length ratio, (3) not connected because of change in direction.

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?

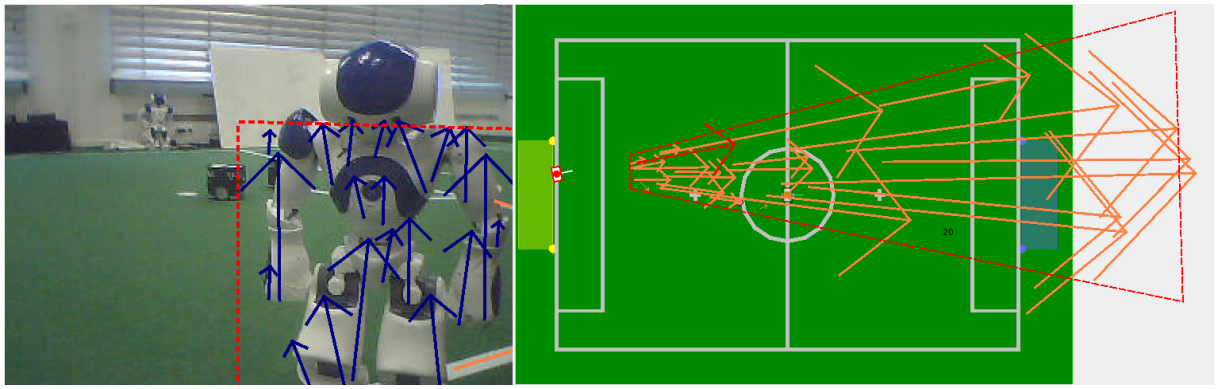


Figure 4.10: Ban sector built from *NonLineSpots*. The red arrow is a false line spot detected in the robot and discarded because of the ban sector.

- 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. 4.10). After creating the ban sectors, the LinePerceptor creates line segments from the line spots. For each line 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 [6], 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

Algorithm 3 Clustering LineSegments

```
while lineSegments ≠ ∅ do
  s ← lineSegments.pop()
  supporters ← ∅
  maxSegmentLength ← 0
  for all s' ∈ 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} ∪ supporters)
    lineSegments ← lineSegments\supporters
  end if
end while
```

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. 4.11). 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 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.

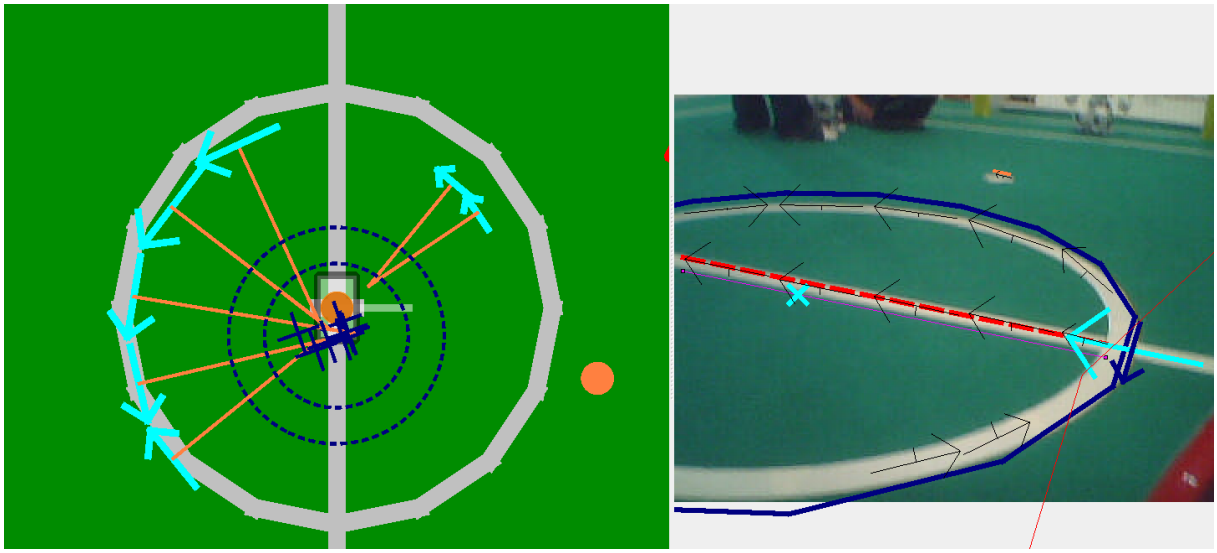
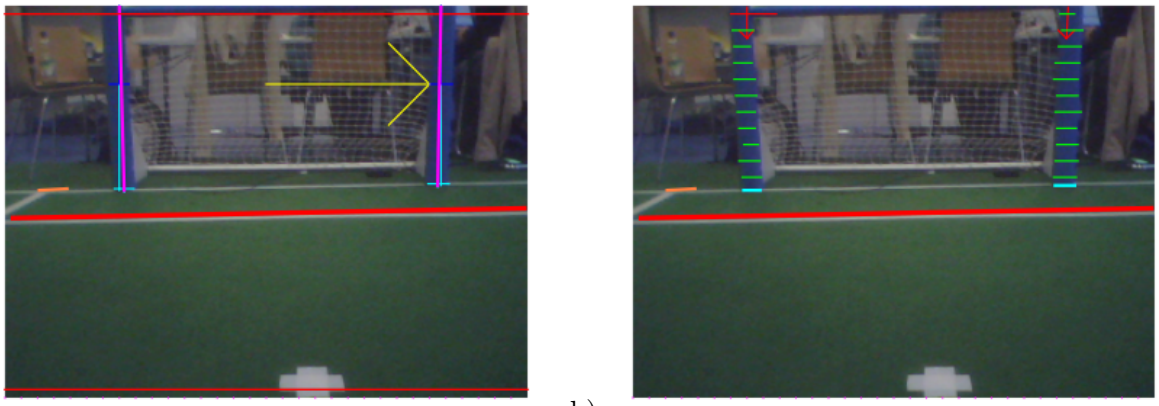


Figure 4.11: Circle detection: blue crosses: circleSpots, blue circles: threshold area, orange lines: perpendiculars of segments



a) b)

Figure 4.12: The debug drawings of the GoalPerceptor. Image a) shows the fist scans (blue and light blue) that find the foot and head of a potential post. The arrow marks the horizon. Image b) shows the runs orthogonal to the vector for determining the width of the potential post.

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 on the *Image* and provides the *GoalPercept*. All steps in this module use the *ColorTable* to segment the *Image*.

Since the foot of a post must be below the horizon and the head of a post must be above (or not visible) the post must cross the projection of the horizon to the image. Therefor it is sufficient to scan the projection of the horizon in the image for blue or yellow segments to detect points

of interest for further processing. If the horizon is above the upper image border, a line a few pixels below the upper image border will be scanned.

From these points of interest the GoalPerceptor tries to find the foot and head of a potential post. This is done by running up and down from a point of interest and continuing the run a few pixels left or right if a border was found (Fig. 4.12).

For each of these 'spots' further runs orthogonal to the vector connecting the head and foot are done to examine the width of the potential post. If the difference between this examined width and the expected width for a post at the position of the foot exceeds a certain threshold, the spot is discarded. Afterwards the GoalPerceptor tries to find some green below the foot of each spot. If there is no green below a spot, the spot will also be discarded.

All remaining spots are transformed to field coordinates. For each spot the foot is transformed to field coordinates, assuming that the foot is in the field plane. We can take another measurement of the distance of the spot using the height of the post. If the distance between these two measurements exceeds a certain threshold, the post gets discarded.

Finally the GoalPerceptor checks whether there are two many spots left or whether there are the same number of yellow and blue spots. In both cases all spots get discarded.

All remaining spots ($\#bluePost \leq 2$ and $\#yellowPost \leq 2$) make up the percept.

4.1.3.5 Detecting the Ball

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

To be considered as a ball, a ball spot must pass the following six steps:

1. Checking the ball spot: ball spots above the horizon or with a distance greater than the length of the field diagonal will not pass this test.

2. Scanning for ball points: the ball spot will be validated and new ball points will be generated based on the distance of the Cb and Cr components of the surrounding pixels.

3. Checking the ball points: the ball points will be validated. Duplicates and ball points with a high deviation of the radius will be removed.

4. Calculating center and radius of the ball.

5. Checking the environment in the current image: since the radius of the ball in the image is known, it can be checked whether there are any ball points outside the radius. Therefore some points on a circle around the ball are calculated. Their Manhattan distance of the Cb and Cr components must not exceed a certain threshold. If it does, the ball will not pass this test.

6. Calculating the relative position to the robot: the relative position of the ball on the field will be calculated either based on the size of the ball or based on the transformation from image coordinates to field coordinates.

The resulting *BallPercept* is the input for the module BallLocator (cf. Sect. 4.2.3).

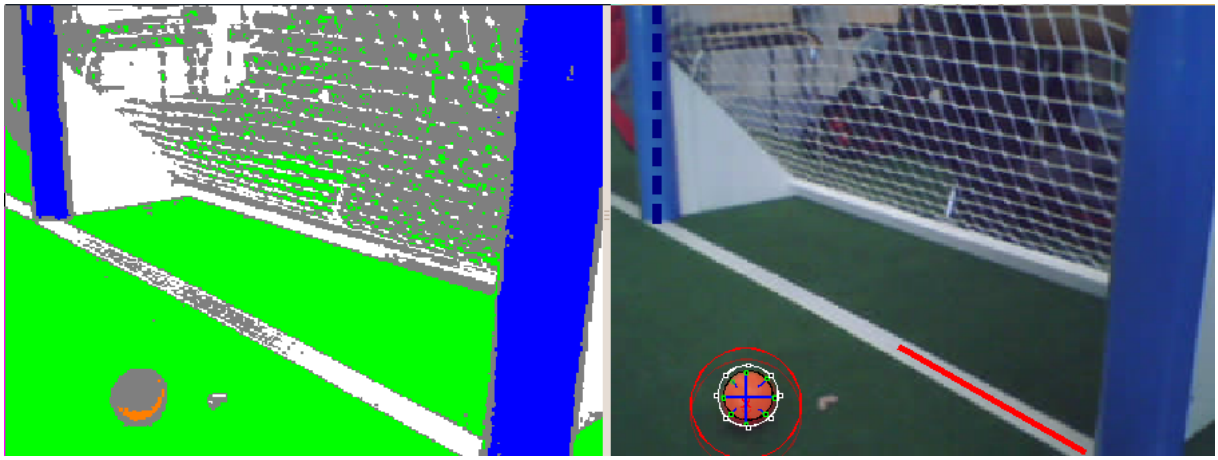


Figure 4.13: The debug drawing of the BallPerceptor. The green dots mark the end of the scan lines for ball points. The white circle should be around the ball, i. e. the color differs too much from the ball spot.

4.1.3.6 Detecting Other Robots

The RobotPerceptor requires the representation *RegionPercept* and extracts regions that are classified as blue (team markers of the blue team) or red (team markers of the red team) and validates the region's area and it's eccentricity to determine whether those regions are team markers. Based on the principal axis of inertia of the team marker regions, the RobotPerceptor will calculate the orientation of the team marker. An orientation of 0 or π means the robot is definitely standing, whereas an orientation around $\frac{\pi}{2}$ means the robot is not standing (cf. Fig. 4.14).

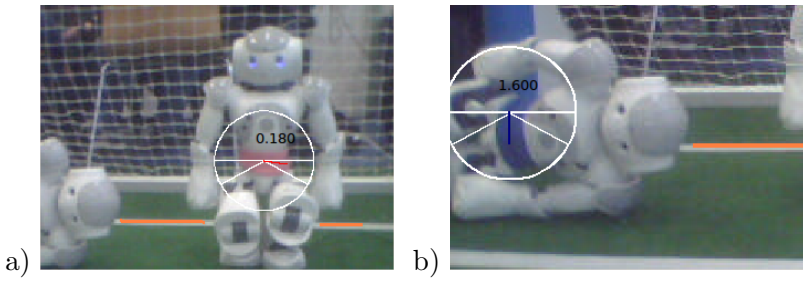


Figure 4.14: a) An upright robot. The white lines inside the circle mark the region in which the principal axis of inertia (red line) of the team marker's region has to be if the robot is upright. b) Since the principal axis of inertia (blue line) is outside the region limited by the white lines, this robot is considered as horizontal.

The principal axis of inertia is also used to define the direction of the orthogonal scan lines that verify the team markers environment. The dominating color of the environment should be white. This is verified by calculating the ratio of white pixels on the scan lines (cf. Fig. 4.15).

If the candidate robot passed all these tests, we have to find a point that allows for the localization of the robot on the field. Since it is inaccurate to determine the robot's position if we just know the position of the team marker, we search for the first green area below the robot, so that we can project a pixel on the field.

The robot detection relies on good lighting conditions and an accurate color table. White background could be a problem that results in false positives. We are already working on an improved version that will be more independent from color tables and checks more constraints.

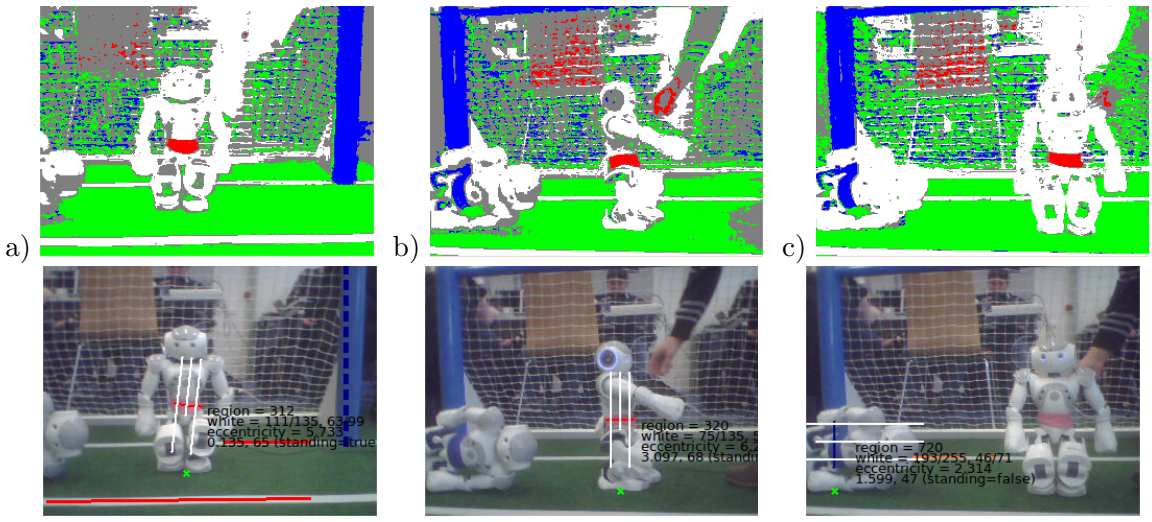


Figure 4.15: a) The segmented and raw image of a kneeling keeper. The white lines depict the scans around the team marker region. The red line marks the principal axis of inertia of the team marker region. The scan lines will only be shown if the robot is recognized. The green cross on the ground marks the green area below the robot and will be used to calculate the position of the robot. b) The robot is rotated by 90 degrees. Of course it can only be recognized if the arms do not cover the team marker from this perspective. c) A robot lying on it's side. Since the orientation of the team marker is vertical, it is recognized as horizontal. In this case the setup seems to be quite artificial but it also works for a robot lying on it's back and with it's arms behind the back, which is typical for a robot that tries to stand up.

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. 4.16.

4.2.1 Self-Localization

For self-localization, B-Human uses a particle filter based on the Monte Carlo method [1] as it is a proven approach to provide accurate results in such an environment [24]. 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 [5] has been implemented. A comprehensive description of our general state estimation implementation – applied to a Humanoid League scenario – is given in [13]. An overview of subsequently added features and the necessary perceptive components was published in [11].

The module providing the *PotentialRobotPose*, which is a simple pose in 2-D, is the SelfLocator.

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.

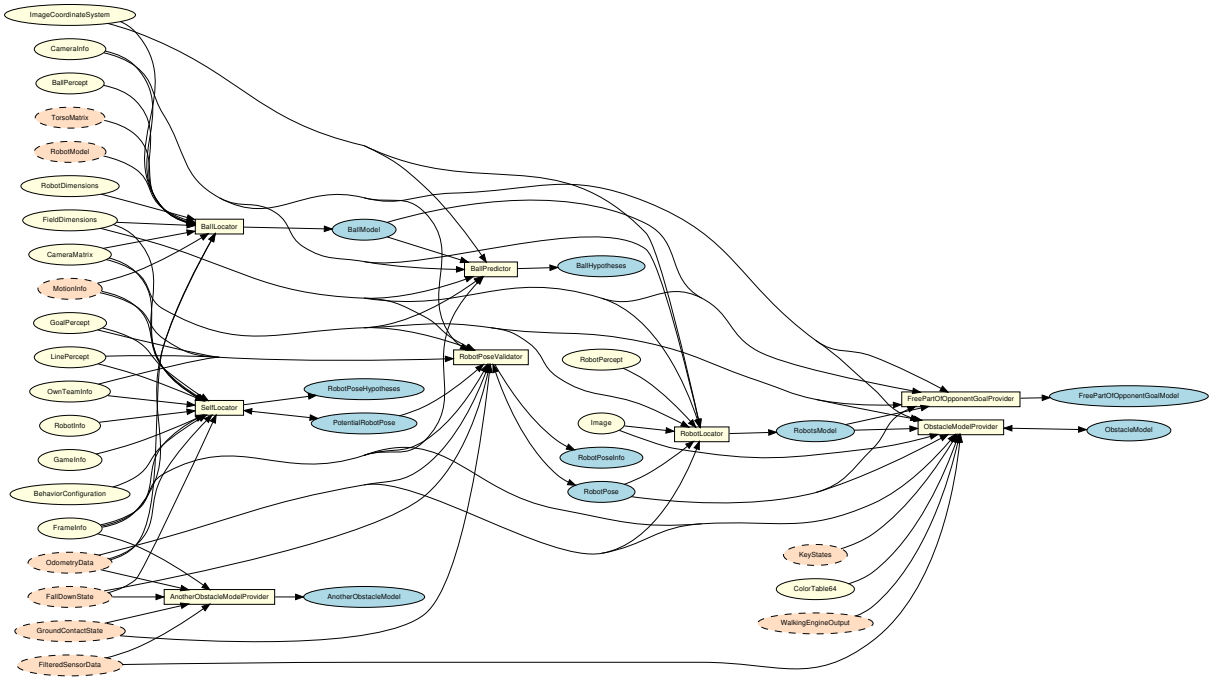


Figure 4.16: Modeling module graph

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.

Through the process of sensor resetting [17] (cf. Fig. 4.17b), which is used to overcome kidnapping problems, the probability distribution of the sample set often becomes multimodal (cf. Fig. 4.17a). To robustly and computationally efficiently track different sample clusters without any discretization drawbacks as, for instance, in the popular binning approach, the approach presented in [15] is applied.

4.2.2 Robot Pose Validation

Although the pose provided by the SelfLocator is reliable enough for tasks such as shooting the ball in goal direction or positioning between ball and goal, it has the following drawbacks:

- The provided pose might be inconsistent since the amount of used particles is restricted for performance reasons.
- The distribution of the particles expands if the robot does not perceive enough lines although it is not moving.
- It is never known whether the provided pose is close to the actual position, especially after repositioning the robot or after getting up. In these cases, the provided pose might be wrong for some time.

To address these problems the RobotPoseValidator, which provides the *RobotPose*, was introduced. It uses the *PotentialRobotPose* from the SelfLocator to relate line, goal and center circle

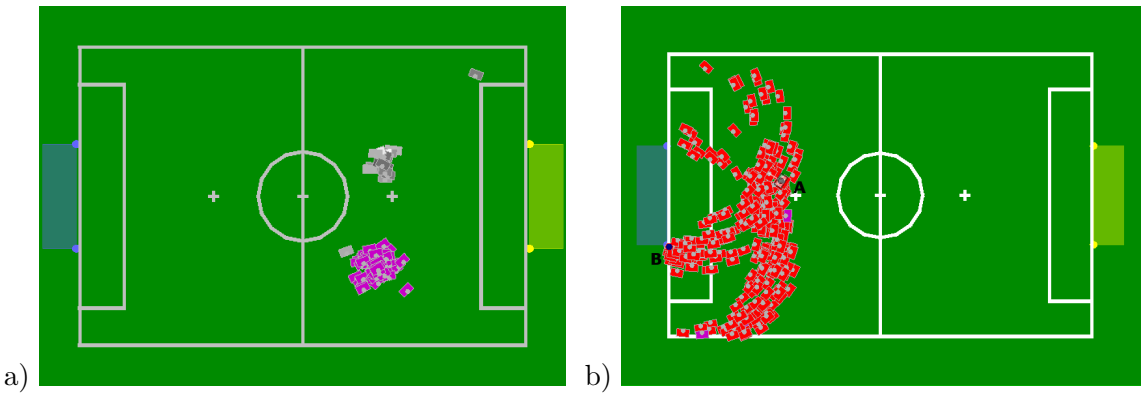


Figure 4.17: a) Multi modal probability distribution: The samples have clustered at two different positions. The cluster selected for pose computation is drawn in magenta. b) Sensor resetting: A robot standing at position A observes the non-unique goalpost at position B. The red boxes denote a sample 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.

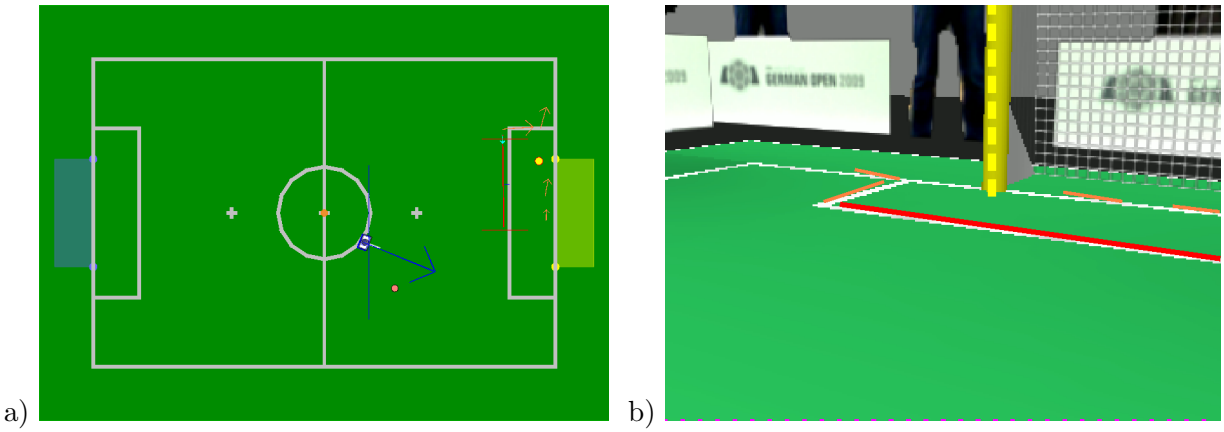


Figure 4.18: a) Robot pose components calculated from a single image (b). The vertical blue line marks a x-component calculated from line percept (large red line) and the two almost identical blue arrows mark calculated rotations using the line percept and the goal percept.

percepts to their counter parts on the field. This way each line percept can be used to calculate a translational component and the rotation of the robot pose. The results are filtered using a two-dimensional Kalman filter for the translational part and a one-dimensional Kalman filter for the rotation. The goal percepts are used to calculate additional rotations by assuming that the estimated position is correct. The center circle is used for additional translational components by assuming that the estimated rotation is correct. The resulting pose is fairly accurate, without the noise from the particle filter. In cases such as a mismatch of the goal posts – i. e. if the distance between the seen and estimated position is too large –, the provided pose is considered to be invalid. In such a situation the behavior can react by looking for field lines and goal posts to improve the localization.

4.2.3 Ball Tracking

Having used a particle filter to estimate the velocity and position of the ball [23], the new module BallLocator now uses Kalman filters to derive the actual ball motion, the *BallModel*, off the sensor data given the perception described in Sect. 4.1.3.5. Since ball motion on a RoboCup soccer

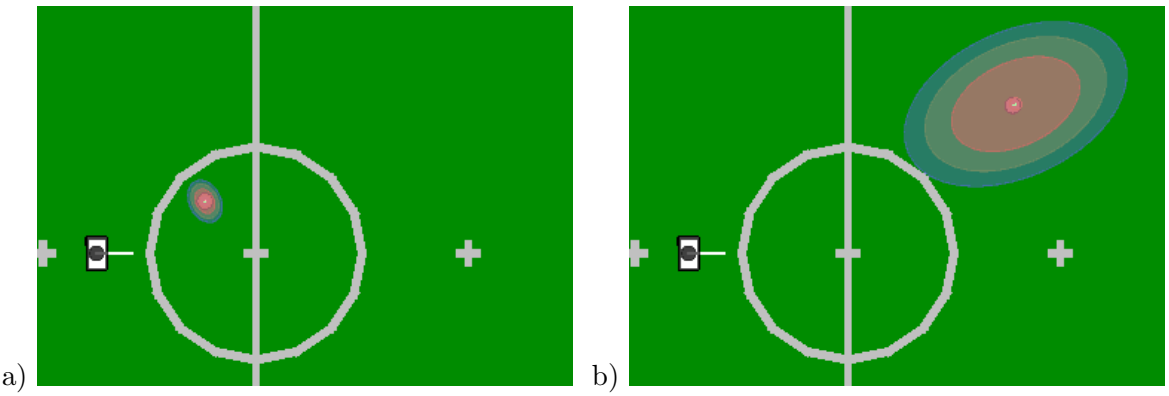


Figure 4.19: Ball model and measurement covariances for short (a) and medium (b) distances. The orange circles show the ball models computed from the probability distribution. The larger ellipses show the assumed covariances of the measurements.

field has its own peculiarities, our implementation extends the usual Kalman filter approach in several ways described below.

First of all the problem of multi-modal probability distributions, which was naturally handled by the particle filter, deserves some attention when using a Kalman filter. Instead of only one we use twelve multivariate Gaussian probability distributions to represent the belief concerning the ball. Each of these distributions is used independently for the prediction step and the correction step of the filter. Effectively, there are twelve Kalman filters running in every frame. Only one of those distributions is used to generate the actual ball model. That distribution is chosen depending on how well the current measurement, i. e. the position the ball is currently seen at, fits and how small the variance of that distribution is. That way we get a pretty accurate estimate of the ball motion while being able to quickly react on displacements of the ball, for example when the ball is moved by the referee after being kicked off the field.

To further improve the accuracy of the estimation, the twelve distributions are equally divided into two sets, one for rolling balls and one for balls that do not move. Both sets are maintained at the same time and get the same measurements for the correction steps. In each frame, the worst distribution of each set gets reset to effectively throw one filter away and replace it with a newly initialized one.

There are some situations in which a robot changes the motion of the ball. After all, we filter the ball position to finally get to the ball and kick it. The robot influences the motion of the ball either by kicking or just standing in the way of a rolling ball. To incorporate theses influences into the ball model the mean value of the best probability distribution from the last frame gets clipped against the robot's feet. In such a case the probability distribution is reset, so that the position and a velocity of the ball get overwritten with new values depending on the motion of the foot the ball is clipped against. Since the vector of position and velocity is the mean value of a probability distribution, a new covariance matrix is calculated as well.

Speaking of covariance matrices, the covariance matrix determining the process noise for the prediction step is fixed over the whole process. Contrary to that, the covariance for the correction step is derived from the actual measurement; it depends on the distance between robot and ball.

4.2.4 Ground Truth

As source for ground truth data, providing *RobotPose* and *BallModel*, a global tracking system is used. A unique marker is fixed on the robot's head (cf. Fig. 4.20) and tracked by the standard

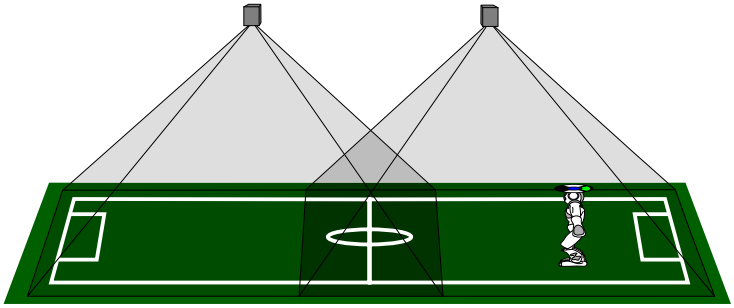
a)  b) 

Figure 4.20: a) A Nao robot equipped with a colored pattern for global tracking. b) Two cameras mounted above the field with overlapping images.

vision system of the RoboCup Small Size League [26], processing images from a camera hanging above the field. 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.

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 module SSLVision 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.

4.2.5 Obstacle Model

Since the vision system is not yet able to robustly detect close robots, the major 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. 4.21b) which rasterizes the robot's local environment. The grid currently used has a resolution of 45×45 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. 4.21a) 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

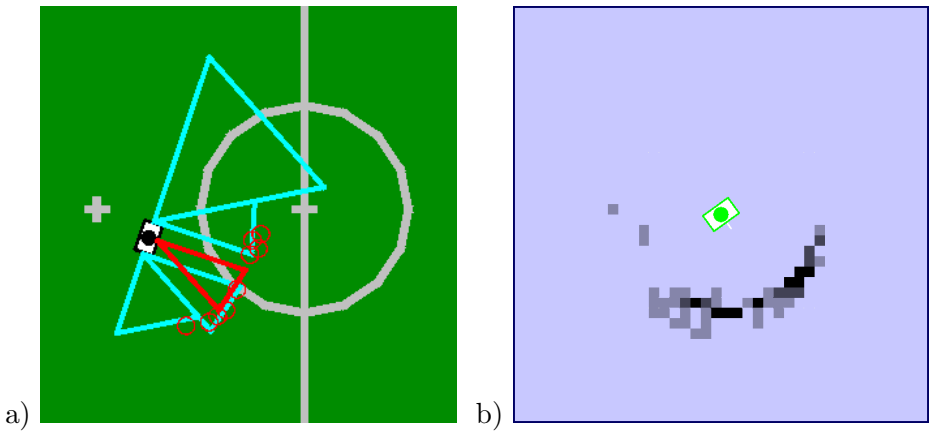


Figure 4.21: Modeling obstacle information: a) The *ObstacleModel* drawn as cyan polygons relative to the robot. The red circles show the currently occupied cells, the red triangle the current ultrasonic measurement. b) The local occupancy grid of the situation shown in a). The cells' color depends on their counter n_o.

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 3 s) 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. 4.21a. 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.

In most configurations, robots that are lying on the ground cannot be perceived by the ultrasonic sensors. To overcome this problem, results of the robot tracking module (cf. Sect. 4.2.6) are also added to the grid.

A further enhancement for the obstacle model is the verification of obstacles in the image. The obstacles registered in the grid are projected into the current image. An obstacle is deleted from the grid, if its position on the field is inside the current image and the percentual value of green is above 95%.

4.2.6 Robot Tracking

To track robots that are recognized via vision, we use a Kalman filter for each detected robot. Actually it is not a real Kalman filter, since we did not implement a motion model yet. So we are currently assuming that all recognized robots do not move but we add a significant noise in each cycle so that we can react faster to the robot's movements. Thus motion is considered as noise, which is not very inaccurate since most of the robots of the Standard Platform League usually do not move very fast yet.

The robots' environment is partially observable. The challenging part of the RobotLocator is to match recognized robots with robots in the model. The basic idea is to calculate the Mahalanobis

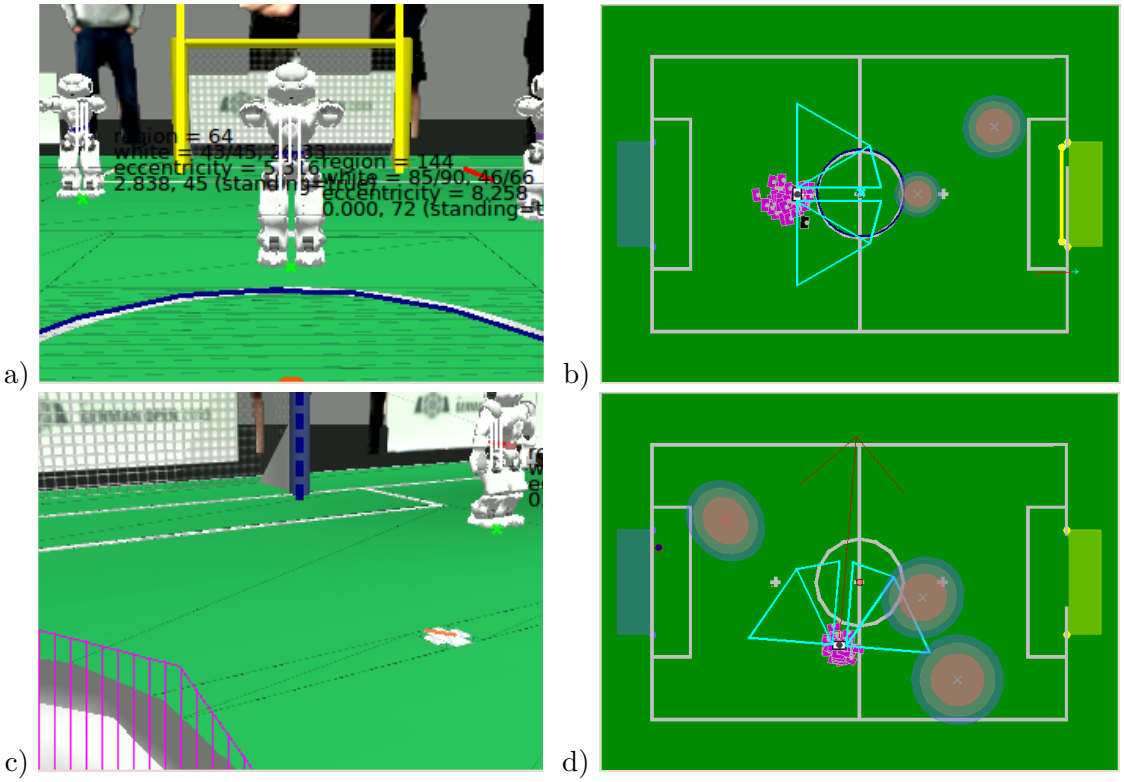


Figure 4.22: a) Simulated camera image before kickoff. The white scan lines mark recognized robots. b) The corresponding debug drawings on the field. The crosses mark the assumed position of the robots and the slight transparent circles represent the error ellipse calculated from the covariance matrix. c) A recognized robot on the left side. d) The corresponding error ellipse shows that the uncertainty of the distance is greater than the uncertainty of the angle.

distance of the detected robots to the robots in the model and always match those robots with the lowest distance. This approach would work perfectly for fully observable environments. The Mahalanobis distance as developed in [19] can be calculated as follows:

$$d(x,y) = \sqrt{(x-y)(\Sigma^{-1}(x-y))}.$$

This requires a good estimation of the measurement noise. We are usually quite sure about the angle of the recognized robot and unsure about the distance. So if the robot is in front of us, that is the angle is 0, the deviation along the x-axis is high and the deviation along the y-axis is low. If the robot is not in front of us we have to rotate the covariance of the measurement. We do this by multiplying the covariance with a rotation matrix from both sides:

$$\Sigma = \begin{pmatrix} \cos\alpha & -\sin\alpha \\ \sin\alpha & \cos\alpha \end{pmatrix} \begin{pmatrix} \sigma_x^2 & 0 \\ 0 & \sigma_y^2 \end{pmatrix} \begin{pmatrix} \cos\alpha & -\sin\alpha \\ \sin\alpha & \cos\alpha \end{pmatrix}^T,$$

where α is the angle of the measurement vector in the robot coordinate system. σ_x and σ_y are parameters that have to be defined. We use this formula to calculate the initial covariance of a robot and the measurement covariance.

In fact, almost never every robot is tracked at the same time. So it is difficult to find out whether a detected robot correlates with a robot in the model or is a new robot. To solve this problem we introduced the Euclidean distance as a criterion to determine whether we update an existing robot or add a new robot to our model. If the Euclidean distance of the recognized robot is

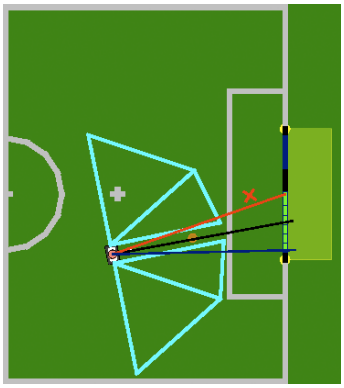


Figure 4.23: Angles from robot to sides/center of largest free part of the opponent goal. The largest free part of the opponent goal itself is visualized by green dots in each of its cells. The dark blue parts of the opponent ground line are clear, the black ones are blocked.

too high, it will not update the previously tracked robot. Another difficulty is losing track of robots we did not see for a long time. They will not be tracked any longer if the approximated probability of an area around the mean is below a certain threshold.

4.2.7 Largest Free Part of the Opponent Goal

Up to now, robots detected by our vision are primarily used to find out which section of the opponent goal is blocked and which is free to shoot at.

The *FreePartOfOpponentGoalModel* divides the opponent goal line into fourteen equally sized cells. The located robots are then projected onto the opponent goal line, with an estimated robot width of 200 mm. The value of each cell hit by the projection increases. Afterwards an aging factor is applied to the value of all cells. If the value of a cell raises above a certain threshold, we assume it is blocked by a robot. Every other cell is free, i. e. a ball kicked in its direction would not be obstructed.

Since the ball kicked by a Nao will never roll on a straight line, we want to shoot at the largest group of free cells, to maximize the chance to score a goal. We simply count the number of neighboring cells, although groups of cells on the kicking robot's side of the obstacle are preferred, because they are simply easier to hit. Also cells involved in the previously-found largest free part are favored to prevent it from oscillating between two parts of nearly the same size.

If the largest free part found falls below a minimum size the model will assume the whole goal is free, so the robot will at least shoot into its center. However a "whole-goal-blocked" flag is set. In the future we want to pass the ball to a teammate, who hopefully has a clearer shot.

Chapter 5

Motion

The process *Motion* 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 dynamic motions. Therefore, the *Sensing* part must be executed before *Motion Control*.

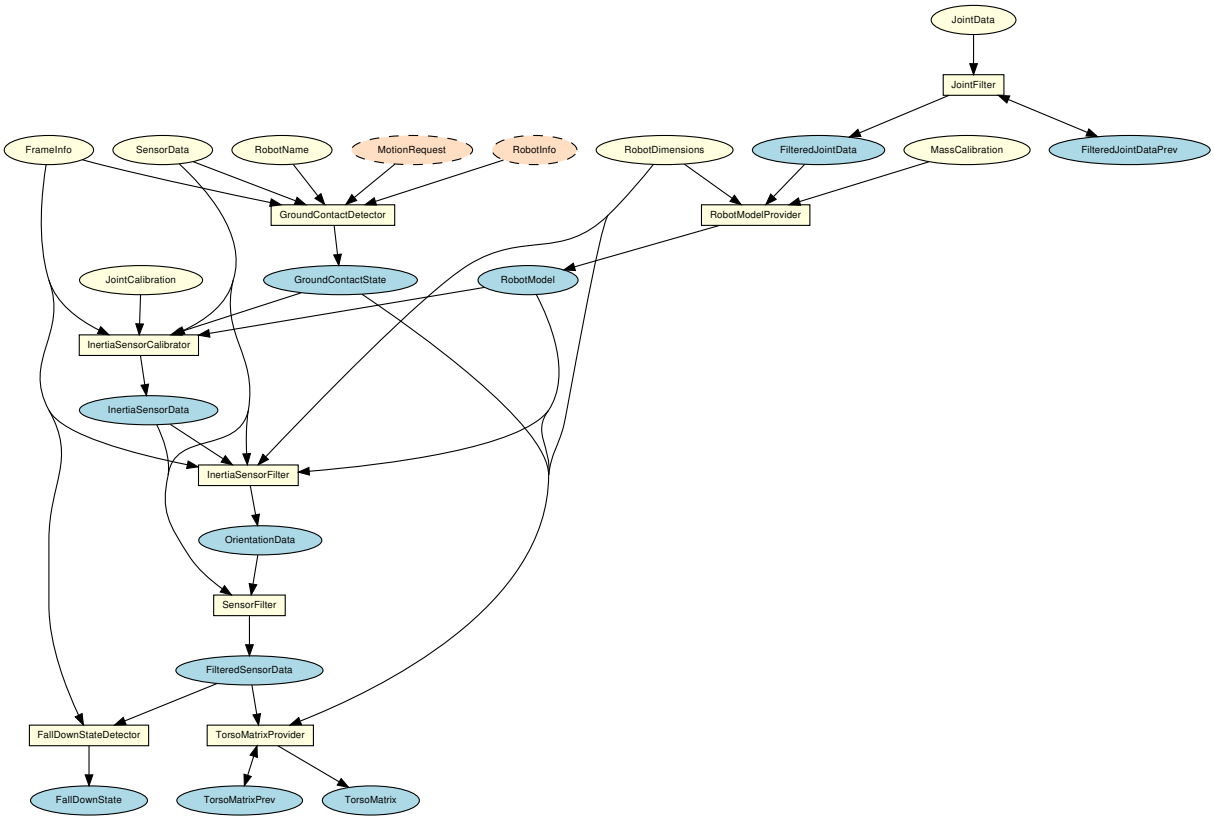


Figure 5.1: All modules and representations in the area *Sensing*

5.1 Sensing

An important task of the motion process is to retrieve the readings 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 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 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 InertiaSensorCalibrator, InertiaSensorFilter, and SensorFilter module that provides the representation *FilteredSensorData* (cf. Sect. 5.1.5). The module InertiaSensorCalibrator that determines the bias of the readings from the IMU uses 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 *FilteredSensorData* and the *RobotModel*, it is possible to calculate the *TorsoMatrix* that describes a transformation from the ground to an origin point within the thorax of the Nao (cf. Sect. 5.1.6). The *FilteredSensorData* are also used to detect a possible fall of the robot by the FallDownStateDetector which then provides the *FallDownState* (cf. Sect. 5.1.7). Figure 5.1 shows all modules in the area *Sensing* and the representations they provide.

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) 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 and the InertiaSensorCalibrator, the GroundContactDetector is the only module that uses the unfiltered *SensorData*, since the result of the GroundContactDetector is required by the InertiaSensorCalibrator. So it is not possible for the GroundContactDetector to use the *FilteredSensorData*.

To recognize the *GroundContactState* the force sensing resistors in the Nao's feet and the electric current sensors of the leg motors are used. The *GroundContactState* is considered to be true if a weighted sum of these readings exceeds a certain threshold for a certain period.

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 (in file *masses.cfg*). The configuration provided was taken from the documentation of the Nao.

5.1.4 Inertia Sensor Data Calibration

The SensorDataCalibrator module is responsible for two different tasks. It drops broken inertial sensor values and it determines a bias for the gyroscope and acceleration sensors. Therefore it uses the *SensorData* and the results are provided as *InertiaSensorData*. It uses the *GroundContactState* to avoid calibration when the robot is not standing on an approximately even ground.

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 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, 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.

A similar method is applied to the acceleration sensors. When the robot is standing it is assumed that both feed are evenly on the ground to calculate orientation of the robot's body

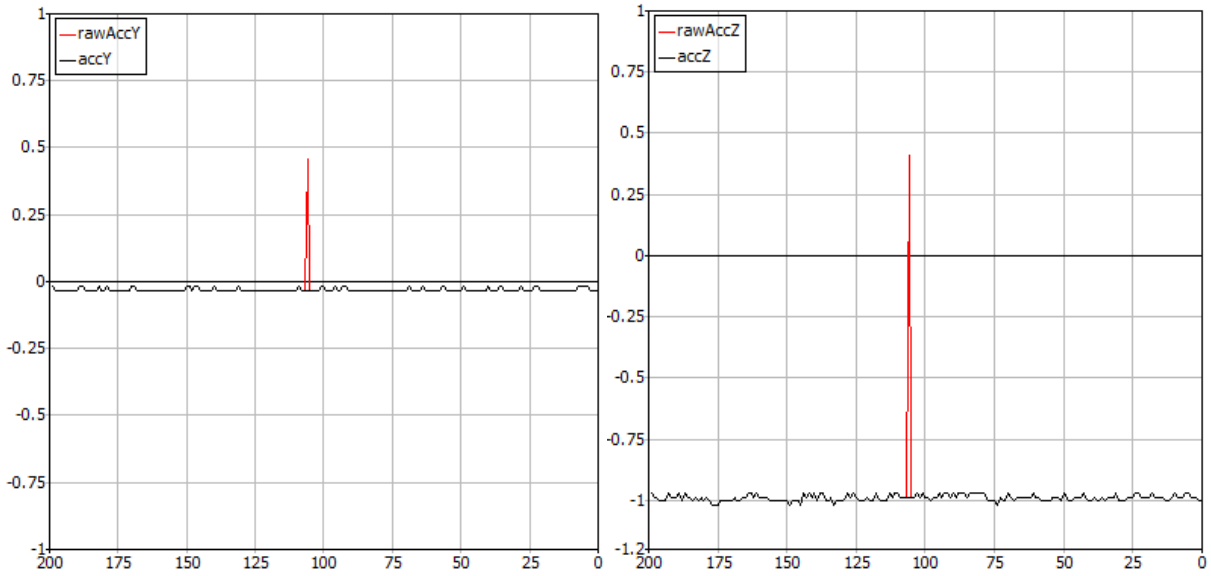


Figure 5.2: A typical broken inertia sensor data frame. The broken data was detected and replaced with its predecessor.

and expected acceleration sensor readings. These expected acceleration sensor readings are then used for determining the calibration offset.

5.1.5 Inertia Sensor Data Filtering

The InertiaSensorFilter module calculates an angle of the robot torso relative to the ground using an Unscented Kalman filter (UKF) [8]. 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. The resulting orientation is provided as *OrientationData*.

5.1.6 Torso Matrix

The *TorsoMatrix* describes the three-dimensional transformation from the projection of the middle of both feet on the ground up to the center of hip within the robot torso. Additionally, the *TorsoMatrix* contains the change of the position of the center of hip including odometry. Hence, the *TorsoMatrix* is used by the *WalkingEngine* for estimating the odometry offset. The *CameraMatrix* within the *Cognition* process is based on the *TorsoMatrix*.

To calculate the *TorsoMatrix* 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. 5.1.5), represented as rotation matrix R.

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

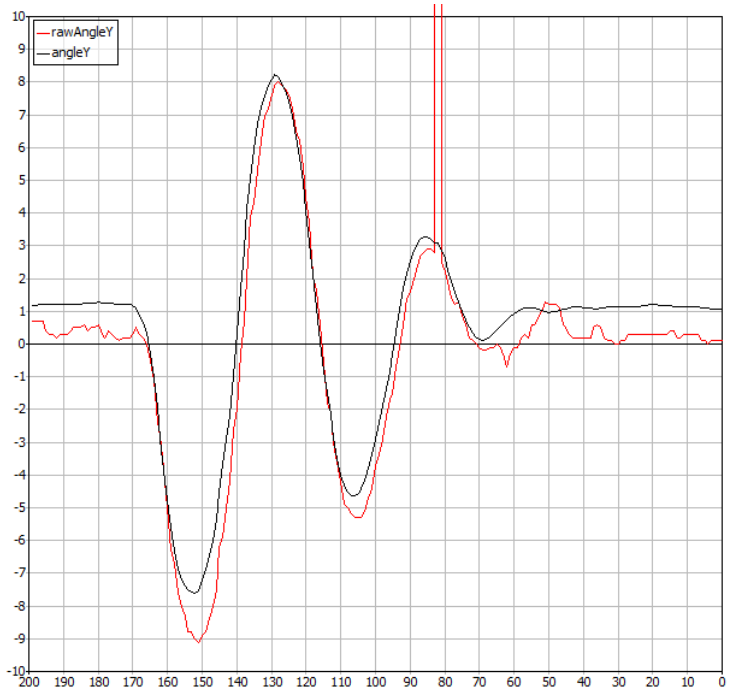


Figure 5.3: The difference between the estimated pitch angle *angleY* and the pitch angle *rawAngleY* provided by the inertia board.

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

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 torso matrix p_{im} by using the longer leg and the constant offset h that describes the position of the center of hip 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 center of hip is determined by using the inverted torso 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 torso matrix.

5.1.7 Detecting a Fall

Although we try our best at keeping the robot upright, it still happens to fall over occasionally. In such a case it is helpful to switch off the joints and bring the head into a safe position to protect the robot hardware from unnecessary damage. The task of detecting such a situation is realized by the FallDownStateDetector that provides the *FallDownState*. When trying to detect a fall we only have a very small amount of time to get the necessary actions done, as a normal fall takes about 100ms from the moment when we can be sure that the fall is inevitable. On

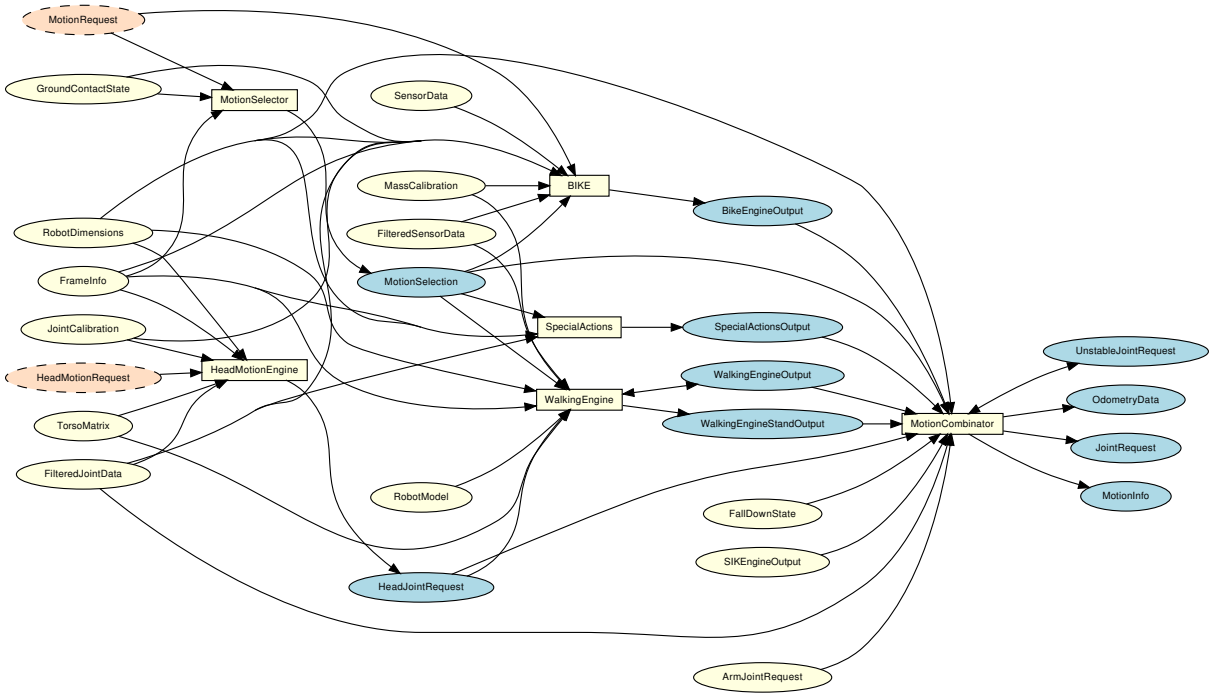


Figure 5.4: All modules and representations in the area *Motion Control*

the other hand it has fatal consequences when a fall is detected although the robot would not have fallen with the joints not being switched off. For that reason, fall detection is split up into 2 phases, the first of which is to detect that the robot is staggering. If that is the case we bring the head into a save position, which can be done without major consequences. In the second phase we detect whether the robot exceeds a second threshold in its tilt angle. If that is the case, we power down the joints to a low hardness. It has shown that this is a better approach than switching them off completely, because deactivated joints tend to gain too much momentum before hitting the floor. To determine the robot's tilt angle, the FallDownStateDetector utilizes the *FilteredSensorData*.

To start an appropriate get-up motion after a fall, the FallDownStateDetector then determines whether the robot is 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.

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*, *kicking*, and *special actions*. The walking motion and a corresponding stand are dynamically generated by the WalkingEngine (cf. Sect. 5.2.3). Some kicking motions are generated by the Bézier-spline and inverse kinematics-based kick engine (BIKE) that is described in [20]. All further motions that are created by static joint angle patterns, are provided by the module SpecialActions (cf. Sect. 5.2.4). All three modules generate joint angles. The WalkingEngine provides the *WalkingEngineOutput* and the *WalkingEngineStandOutput*. The module BIKE provides the *BikeEngineOutput* and 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, two 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*. Figure 5.4 shows all modules in the area *Motion Control* and the representations they provide.

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, BIKE, 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. This task is encapsulated within a separate module HeadMotionEngine for two reasons: on the 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

A continuous series of joint angles for walking and a set of joint angles for standing are generated by the *WalkingEngine* module. The approach used this year is a further development of last year's approach [23] [4] with an improved method for controlling the center of mass motion and altered usage of sensor feedback. The trajectory of the center of mass is based on the Three-Dimensional Linear Inverted Pendulum Mode (3D-LIPM) [9]. Hence, the position and velocity of the center of mass relative to the origin of the inverted pendulum are given by

$$x(t) = x_0 \cdot \cosh(k \cdot t) + \dot{x}_0 \cdot \frac{1}{k} \cdot \sinh(k \cdot t) \tag{5.6}$$

$$\dot{x}(t) = x_0 \cdot k \cdot \sinh(k \cdot t) + \dot{x}_0 \cdot \cosh(k \cdot t) \tag{5.7}$$

where $k = \sqrt{\frac{g}{z}}$, g is the gravitational acceleration ($\approx 9.81 \frac{\mathrm{m}}{\mathrm{s}}$), z is the height of the center of mass above the ground, x_0 is the position of the center of mass relative to the origin of the inverted pendulum at $t = 0$, and $\dot{x}_0$ is the velocity of the center of mass at $t = 0$.

In a single support phase the inverted pendulum defines the motion of the center of mass according to its position and velocity relative to the origin of the inverted pendulum. Hence at the beginning of a single support phase, the position and velocity of the center of mass should be in a state that leads to the proper position and velocity for the next single support phase (of the other leg). The origins of the inverted pendulums should thereby be placed as close as possible under the center of the feet so that the positions of the origins are defined by the steps that should be performed. Since the steps that should be performed can be chosen without severe constraints, the movement of the center of mass has to be adjusted for every step to result into origins that match to the feet positions. Most walking approaches use a short double support phase for accelerating or decelerating the center of mass to achieve such an adjustment. To maximize the possible range that can be passed within a phase, the single support phase should make up as much as possible of the whole step phase to reduce the accelerations that are necessary for shifting the foot. Hence, the approach used aims on not using a double support phase while keeping the origins of the inverted pendulums close to their optimal positions.

To proceed without a double support phase, there is a method required to manipulate the movement of the center of mass. Therefore, the point in time for altering the support leg is used to control the velocity of the center of mass in y-direction as well as shifting the origin of the inverted pendulum along the x-axis towards the elongated shape of the feet. This way the velocity of the center of mass can be manipulated in x-direction as well as in y-direction which allows to control these velocities to pass a specific distance (step size) while swinging from one leg to the other.

First of all, a definition of the point in time $t = 0$ is required to determine the point in time to alter the support leg. $t = 0$ is defined as the inflection point of the pendulum motion where the y-component of the velocity is 0 ($(\dot{x}_0)_y = 0$). The position of the center of mass at this point $(x_0)_y$ is an arbitrary parameter and has a value of greater or lower than 0 depending on the active support leg. This allows to use

$$x_y(t) = (x_0)_y \cdot \cosh(k \cdot t) \tag{5.8}$$

in the time range from t_{so} to t_s as equation that provides the y-component of the center of mass position relative to the origin of the inverted pendulum. If the non-supporting foot should be placed with a distance of $\bar{s}_y$ to the supporting foot at the end of the single support phase, the point in time to alter the support leg can be determined by finding t_s and $\bar{t}_{so}$ ($t_s > 0$, $\bar{t}_{so} < 0$) where:

$$x_y(t_s) - \bar{x}_y(\bar{t}_{so}) = \bar{s}_y \tag{5.9}$$

$$\dot{x}_y(t_s) = \dot{\bar{x}}_y(\bar{t}_{so}) \tag{5.10}$$

For the trajectory of the motion of the center of mass in x-direction the optimal position $\bar{s}_x + (\bar{x}(\bar{t}_{so}))_x$ and velocity $(\dot{\bar{x}}(\bar{t}_{so}))_x$ of the center of mass at the point in time $\bar{t}_{so}$ are calculated for the given step size $\bar{s}_x$ by using $\bar{x}(t)$ and $\dot{\bar{x}}(t)$ with $(\bar{x}_0)_x = 0$ and $(\dot{\bar{x}}_0)_x = \frac{k \cdot \bar{s}_x}{\cosh(k \cdot \bar{t}_{so}) \cdot \tanh(k \cdot t_s) - \sinh(k \cdot \bar{t}_{so})}$. With these two values and the position and velocity of the center of mass at the beginning of the single support phase a distorted pendulum motion

$$x_x(t) = c_1 \cdot \cosh(k \cdot t) + c_2 \cdot \frac{1}{k} \cdot \sinh(k \cdot t) + c_3 \cdot t + c_4 \tag{5.11}$$

with $x_x(t_s) = \bar{s}_x + (\bar{x}(\bar{t}_{so}))_x$, $\dot{x}_x(t_s) = (\dot{\bar{x}}(\bar{t}_{so}))_x$, $x_x(t_{so}) = (x(t_{so}))_x$, and $\dot{x}_x(t_{so}) = (\dot{x}(t_{so}))_x$ is

determined by solving the linear system of equations:.

$$\begin{array}{lcllcllcl} c_1 \cdot \cosh(k \cdot t_s) & + & c_2 \cdot \frac{1}{k} \cdot \sinh(k \cdot t_s) & + & c_3 \cdot t_s & + & c_4 & = & \bar{s}_x + (\bar{x}(\bar{t}_{so}))_x \\ c_1 \cdot k \cdot \sinh(k \cdot t_s) & + & c_2 \cdot \cosh(k \cdot t_s) & + & c_3 & & & = & (\dot{\bar{x}}(\bar{t}_{so}))_x \\ c_1 \cdot \cosh(k \cdot t_{so}) & + & c_2 \cdot \frac{1}{k} \cdot \sinh(k \cdot t_{so}) & + & c_3 \cdot t_{so} & + & c_4 & = & (x(t_{so}))_x \\ c_1 \cdot k \cdot \sinh(k \cdot t_{so}) & + & c_2 \cdot \cosh(k \cdot t_{so}) & + & c_3 & & & = & (\dot{x}(t_{so}))_x \end{array} \tag{5.12}$$

At the beginning of a single support phase, it is now possible to generate a new step size $\bar{s}_x$, $\bar{s}_y$ which allows to determine t_s, $\bar{t}_{so}$. The functions $x_x(t)$ and $x_y(t)$ can be used to create desired foot positions relative to the center of mass. These positions are transformed into positions relative to the robot body with a iterative procedure similar to the procedure used last year to use inverse kinematics (cf. Sect. 5.2.3.1) to generate joint angles.

5.2.3.1 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. 5.2.3.1.

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]

$$Foot2Hip = Trans_y\left(\frac{l_{dist}}{2}\right) \cdot Foot2Torso \tag{5.13}$$

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.

$$Foot2HipOrthogonal = Rot_x(\frac{\pi}{4}) \cdot Foot2Hip \tag{5.14}$$

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.

$$HipOrthogonal2Foot = Foot2HipOrthogonal^{-1} \tag{5.15}$$

[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. $Trans_{<axis>}(translation)$.

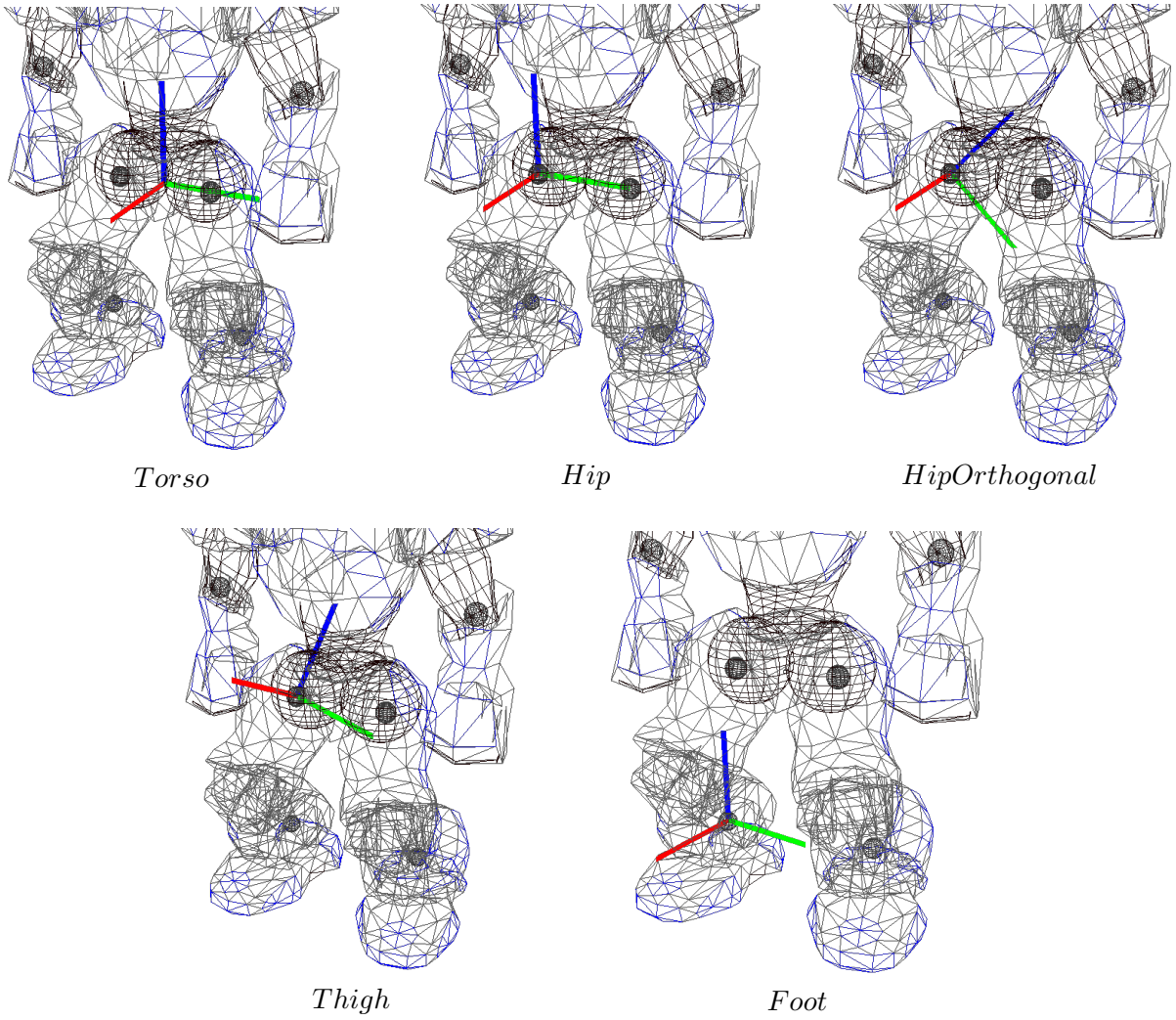


Figure 5.5: Visualization of coordinate frames used in the inverse kinematic. Red = x-axis, green = y-axis, blue = z-axis.

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 (5.16). Knowing that the angle enclosed by the limbs corresponds to the knee joint, that joint angle is computed by equation (5.17).

$$c^2 = a^2 + b^2 - 2 \cdot a \cdot b \cdot \cos\gamma \tag{5.16}$$

$$\gamma = \arccos \frac{{l_{upperLeg}}^2 + {l_{lowerLeg}}^2 - {l_{trans}}^2}{2 \cdot l_{upperLeg} \cdot l_{lowerLeg}} \tag{5.17}$$

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

$$\delta_{knee} = \pi - \gamma \tag{5.18}$$

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

$$\delta_{footPitch1} = \arccos \frac{{l_{lowerLeg}}^2 + {l_{trans}}^2 - {l_{upperLeg}}^2}{2 \cdot l_{lowerLeg} \cdot l_{trans}} \tag{5.19}$$

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]

$$\delta_{footPitch2} = \text{atan2}(x, \sqrt{y^2 + z^2}) \tag{5.20}$$

$$\delta_{footRoll} = \text{atan2}(y, z) \tag{5.21}$$

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.

$$\delta_{footPitch} = \delta_{footPitch1} + \delta_{footPitch2} \tag{5.22}$$

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

$$HipOrthogonal2Thigh = Thigh2Foot^{-1} \cdot HipOrthogonal2Foot \tag{5.23}$$

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

$$Thigh2Foot = Rot_x(\delta_{footRoll}) \cdot Rot_y(\delta_{footPitch}) \cdot Trans_z(l_{lowerLeg}) \cdot Rot_y(\delta_{knee}) \cdot Trans_z(l_{upperLeg}) \tag{5.24}$$

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 $\cos \delta_x$).

$$Rot_{Hip} = Rot_z(\delta_z) \cdot Rot_x(\delta_x) \cdot Rot_y(\delta_y) = \begin{pmatrix} c_y c_z - s_x s_y s_z & -c_x s_z & c_z s_y + c_y s_x s_z \\ c_z s_x s_y + c_y s_z & c_x c_z & -c_y c_z s_x + s_y s_z \\ -c_x s_y & s_x & c_x c_y \end{pmatrix} \tag{5.25}$$

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:

$$\frac{-r_{01}}{r_{11}} = \frac{\cos \delta_x \cdot \sin \delta_z}{\cos \delta_x \cdot \cos \delta_z} = \frac{\sin \delta_z}{\cos \delta_z} = \tan \delta_z \tag{5.26}$$

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

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

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

$$\delta_z = \delta_{hipYaw} = \text{atan2}(-r_{01}, r_{11}) \tag{5.27}$$

$$\delta_y = \delta_{hipPitch} = \text{atan2}(-r_{20}, r_{22}) \tag{5.28}$$

At last the rotation by 45 degrees (cf. eq. 5.14) has to be compensated in joint space.

$$\delta_{hipRoll} = \delta_x - \frac{\pi}{4} \tag{5.29}$$

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 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

[5]Ignored

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.

5.2.5 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

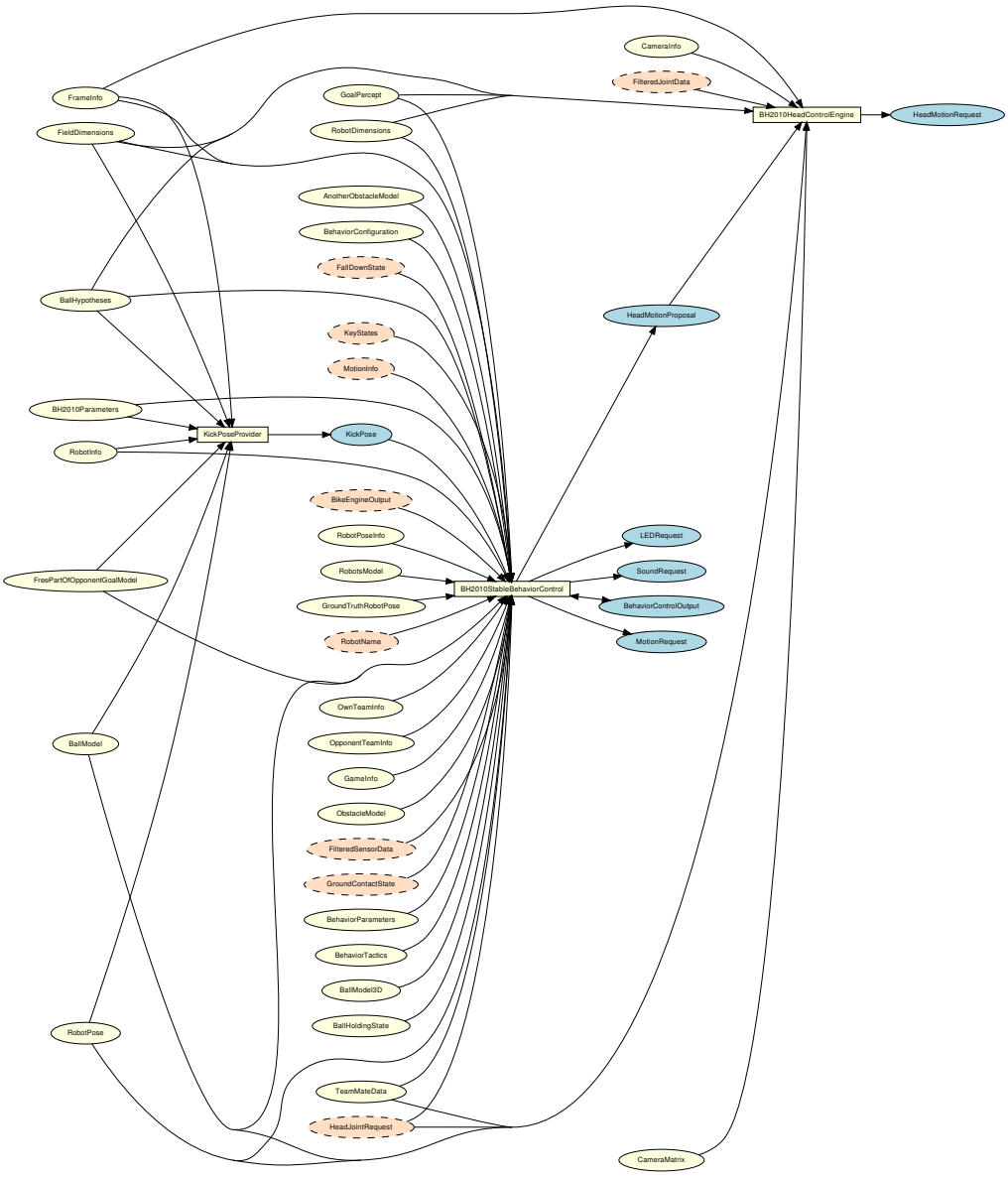


Figure 6.1: Behavior Control module graph

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 2010 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 *BH2010StableBehaviorControl* 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 file *CognitionModules.cfg* that can be found in the directory *Config/Processes*.

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*. In the method `handleMessage` the own behavior has to be added with a logical OR operator.

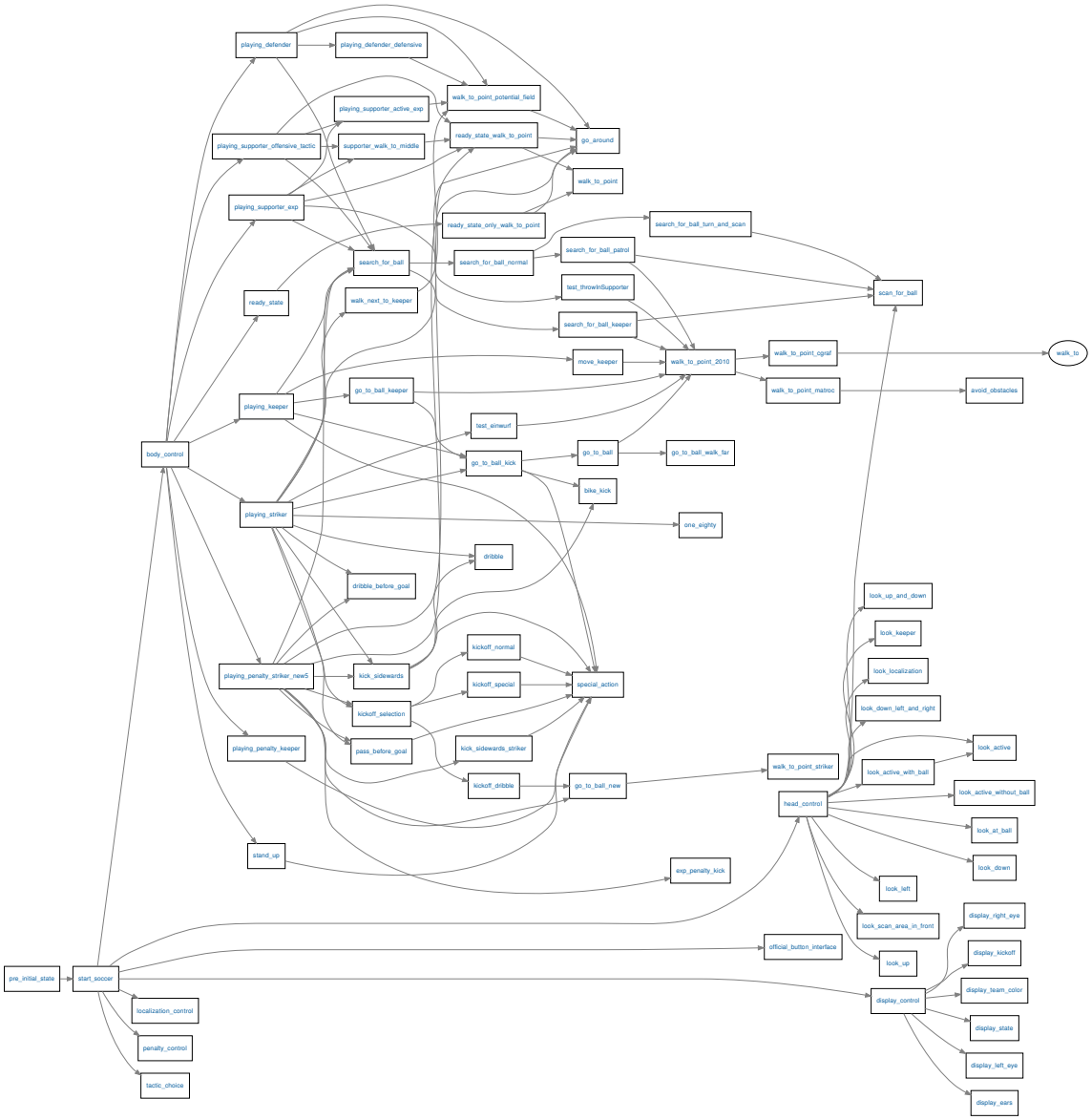


Figure 6.2: Option graph of behavior used at RoboCup 2010

6.3 Behavior Used at RoboCup 2010

In the following the behavior used at RoboCup 2010 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*, *official_button_interface*, *penalty_control*, *localization_control* and *tactic_choice* in parallel. Moreover a role selector is running in parallel, which was transferred from a XABSL-option to the role symbols.

The *official_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 role dynamically corresponding to the current game situation. The *tactic_choice* selects the tactic, which influences the role selector in its behavior. 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, the strategies used, the role selector, and 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). The 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 kick-off team. These can be changed by pressing the left or the right foot button, respectively. It is also possible to switch the game state to penalty shootout. Therefore the robots don't need any special software or to be rebooted to get ready. This is also done by pressing the right foot button which doesn't only cycle through own and opponent kick-off but also through penalty shootout. 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 2010.

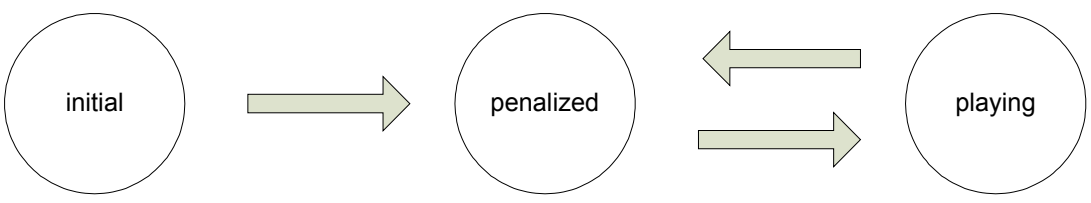


Figure 6.3: States and transitions of the button interface

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 kick-off 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 position, when the other team has kick-off. In case of kick-off, 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 homogeneously, 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. 6.3.7.1. 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 controls the angles of the two head joints, and therefore the direction in which the robot looks. In our behavior the head control is an independent option that runs in parallel to the body control option. In general, it takes requests from body control to either execute a predefined motion pattern or to activate the module BH2010HeadControlEngine to dynamically control the head motions instead. 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 of designing and choosing head control modes is that the information provided by our vision system is required from many software modules with different tasks. For example the GoalPerceptor, BallPerceptor, LinePerceptor, and the RobotPerceptor provide input for the modeling modules (Sect. 4.2) that provide localization, the ball position, and positions of other robots. 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 modes for a variety of needs. In the design process we are able to use three different ways of setting the position of the head. We can specify the absolute angels of the head joints (*panTilt*), or a position on the field (*targetOnGround*). The head control modes used in our behavior are described below.

look_active. In this mode the robot looks to the most interesting point which is calculated by the BH2010HeadControlEngine. Our solution for an effective head motion are points of interest. During the game there are different important points on the field such as the goals, field lines, and the ball position. But the actual interest in a certain point and the unimportance of another one changes from situation to situation. So at a particular time, the robot needs to localize itself and it is unnecessary to look at the ball. In another case, the robot wants to walk to the ball which it does not see, when it is advisable to look at the estimated ball position, the robot got from its teammates. So we collect all points that seem to be important. Besides from the position of these points, we collect the time and presence of eye-contact. To calculate the actual point of interest we use different factors, e. g. the location of the point, the possibility to focus it just by moving the head, the previous point of interest, and the so called *ballfactor*. The *ballfactor* expresses the importance of the ball. The higher the factor, the more the robot looks at the ball.

look_at_ball. Looks at the estimated ball position. This mode is used after finding the ball in an earlier search to confirm the ball position. It is also used when the Nao is near the ball and prepares a kick or starts to dribble with the ball. We then collect information about the ball position over a longer period of time. When the robot has lost the ball, it starts to search for it with *look_active* with the highest possible *ballfactor*.

look_down_left_and_right. Looks to the area in front of the Nao until it notices the ball. This option is used for searching the ball in front of the robot.

look_keeper. Moves the head in a half circle to scan the important area in front of the goal. This mode is used by the penalty keeper, when it has lost the ball, to avoid goal shots of the opponent.

look_localization. The Nao moves the head from left to right until it recognizes the goal. After looking at the goal for a certain time, the robot starts moving the head, stopping when it sees the goal again. With this motion, it has a bigger overview of the field and can see more field lines, resulting in better self-localization, the main purpose of this mode.

look_scan_area_in_front. The head moves in a circle with a left and right motion until the robot notices the ball. Then the Nao looks at the ball for a certain time before starting to scan again. This mode is used as the first head control mode after the penalty state and by the keeper to protect the area in front of the goal.

look_up_and_down. Moves the head up and down in a continuous motion until the Nao notices the ball. Combined with moving around the body, the robot has a good overview about its environment. This mode is used in different situations, e. g. when searching for the ball.

scan_for_ball. Moves the head in a circle until the robot sees the ball. With this mode, the robot has a large field of vision without moving its body. This mode is used for searching the ball in different situations of the game.

6.3.4 Kick Pose Provider

To calculate the optimal position for a kick a module called *KickPoseProvider* is used. It stores for every possible kick (forward, sidewards, backwards) how long it takes, what offset the robot has to have to the ball, and which direction the kick has. With this information it calculates for each kick a *KickPose*, i. e. at which point the robot has to stand and which direction it has to face to execute the kick. Afterwards, each pose is evaluated to find the best one of the possible poses.

The evaluation of the different poses is based on several criteria. The most important one is simply how long it will take the robot to execute the specified kick. This is broken down into how long it takes for the robot to reach the pose and how long it takes to perform the actual kick. Other things that are taken into account are whether the kick is strong enough for the ball to reach the opponent goal and the time since the ball was last seen. Some other constant properties of the kick influence the evaluation via the execution time of the kick. If, for example, a kick is rather unstable and should only be used if the robot is already standing at an almost perfect position for executing the kick, the probability of the kick being chosen can be reduced by increasing its stored execution time.

If the robot is not the keeper and the kick pose is within the own penalty area, the time to reach the pose is set to the maximum value as no player except the keeper is allowed to be within the penalty area. An example for where this situation may apply is when the ball is lying on the own penalty area line and going behind the ball to kick it would result in a penalty, however performing a backwards kick would still be a valid option. If no valid pose is found, i. e. the ball is within the own penalty area, the module returns a pose close to the ball outside the penalty area trying to keep opponent players from reaching the ball giving the keeper more time to kick it out of the penalty area.

The kick is usually directed towards the center of the opponent goal, however there are two exceptions. If the goal is not free, i. e. a robot is blocking the goal, the kick is directed towards the center of the largest free part of the opponent goal. The other exception is when the ball is lying too close to the opponent ground line and not directly in front of the goal. In that case the striker kicks the ball in front of the opponent goal where either the supporter is waiting or where it has a better angle for kicking the ball itself.

6.3.5 Tactics

In contrast to last year's behavior, where only a single formation was used for the robots, tactics were added to the behavior this year. There are three tactics namely *offensive*, *normal*, and *defensive* that are described in detail in this section (cf. Fig. 6.4).

The *normal tactic* is the tactic that was used already in last year's competition. Within this tactic the roles *striker*, *supporter*, and *keeper* are distributed among the field players, where the striker always walks towards the ball when possible (cf. Sect. 6.3.7.1), the supporter always

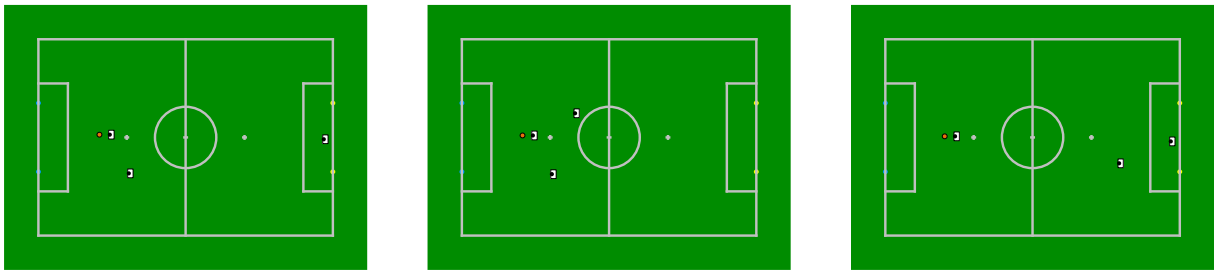


Figure 6.4: Typical situations for each of the tactics. From left to right: normal, offensive, and defensive tactics.

walks next to the striker with a defined offset, and the keeper stays inside its penalty area.

The *offensive tactic* also contains a striker and a supporter, but the keeper does not stay within the own penalty area. Instead of protecting the goal, the keeper becomes a second supporter and walks next to the other side of the striker, so that there is a striker, an offensive supporter and a defensive supporter on the field. The defensive supporter also has a backward offset to the striker that is a bit higher than the one of the offensive supporter. By this arrangement, a triangle is built, so that the goal is not completely unprotected, but an aggressive play can be done.

The opposite of the offensive tactic is the *defensive tactic*. In this case the roles *striker*, *defender*, and *keeper* are used, where the striker is the only player walking towards the ball and into the opponent field half at all. The keeper stays within its own penalty area and the defender supports the keeper by also defending the goal, but in front of the own penalty area.

Generally there are several possibilities for deciding which tactic shall be chosen. In the RoboCup 2010 competition, the tactics were simply chosen solely based on the current goal ratio. If the number of goals shot is at least five higher than the number of goals received, the offensive tactic is chosen. In the opposite case, that is a goal difference of at least five against us, the defensive tactic is chosen. In any other case (a goal difference less than five) the normal tactic is chosen.

In general the tactic is chosen in the option *tactic_choice* mentioned above. There is also a special handling for the ready state. If the ready state is active always the normal tactic is chosen to have a normal positioning of the robots (e. g. the only legal position for the keeper is within the penalty area). Moreover the offensive and defensive tactic only become active if three robots are on field. In any case where less than three robots are on the field, the normal tactic is used (because it is the only tactic that considers this case).

6.3.6 Role Selector

The role selector is used to change the roles of the robots on the field dynamically. The (normally) available roles are *striker*, *supporter*, *defender*, and *keeper* (and *undefined* in the case that a robot is penalized). Furthermore the roles *offensive* and *defensive supporter* exist for the offensive tactic. In general, the role selection is dependent on the number of field players, their current roles, the chosen tactic 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. The role selection will be described first for the normal tactic as it is the most important and normally used the majority of game time. The differences in the role selection for the other tactics will be described afterwards.

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 role that scores goals whereas the other roles are only sensible in conjunction with the striker.

If two robots are on the field, generally a more defensive tactic is chosen. One of the players 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. The selection of which robot becomes striker depends on an assessment that is identical to the one used to distribute *striker* and *supporter* and is described in detail below.

The last possible case is that all three robots are on the field, which get the roles *striker*, *supporter*, and *keeper* in a normal tactic. Naturally the robot with player number 1 is the 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 the characteristics *distance and angle to the ball*, *angle to the line between ball and opponent goal*, and *time since ball last seen*. It estimates which player is the best to reach the ball as fast as possible. These characteristics are computed for both robots and the robot with the better assessment becomes striker, the other robot becomes supporter.

To suppress an oscillation of those two roles for the case that both robots have almost the same assessment, a hysteresis is introduced, which is realized by adding a bonus to the assessment for the current striker. If the difference of the assessments of both robots *is* mostly the same, the roles would not change.

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 kick-off position. If only one field player[1] 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.

For the offensive and defensive tactic the role selection is a bit different. In the ready state there is a change in the roles for the defensive tactic with three players on field. Instead of a supporter, the defenders role is used. The other two roles remain unchanged.

For the offensive tactic no changes will be made in the ready state because the keeper has a defined kick-off position by the rules. In this case always the normal roles (striker, supporter and keeper) are used. In the playing state the roles selection differs for the offensive tactic as follows: In general the role selection is similar to the one used for the normal tactic. The only difference is that the assessment computation is also done for the player with player number 1, because this player is used as third field player in this tactic. In this tactic also the robot with

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

the best assessment (among all players) becomes striker. The distribution of roles for the other two robots additionally depends on their player number. If the player with the player number 1 is not chosen as striker, it always becomes the role of the defending supporter. This is done in the assumption that the keeper should have the shortest way back to the goal. Corresponding to the selection the third robot becomes an offensive supporter. Otherwise, if the player with player number 1 is chosen to be the striker, the roles of the other two robots are assigned based on their distance to the defined positions next to the striker (the total distance of both robots is minimized, where crossing ways are treated as large distances). This algorithm is also executed for the case that the keeper becomes the defensive supporter to decide the side of the striker where the supporters should walk to.

There is also a special case in which the offensive tactic is set back to the normal tactic during the play. This is done in the case that the ball is placed inside the own penalty area. In this situation the defensive supporter becomes keeper and walks back towards the own goal, because it is the only player that is allowed to enter the own penalty area.

For the defensive tactic there are less differences to the normal tactics. The keeper always stays keeper and remains in its own penalty area. The other two roles are distributed according to the assessment mentioned above. The only difference is that the robot not becoming striker doesn't get the supporter's role but the defender's role.

6.3.7 Different Roles

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

6.3.7.1 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 turning the first time, the head control executes a full scan of the area in front of the robot, to rule out the possibility that the ball is there. Afterwards the robot directly starts turning around to search for the ball, where the turning direction is determined based on the last estimation of the ball. After the robot turned until the area scanned before is not in the field of view anymore, the robot stops and again executes a full scan of the area in front of it. The turn and scan process is repeated until the robot has scanned its complete environment. This turn sequence is done by all field players except the keeper, who only turns 45° to one side (assuming that the robot was faced directly towards the opponent goal before), does a full scan of the area in front of it, turns 45° to the other direction and scans again. This sequence will be repeated infinitely until the keeper (or another field player) finds the ball. For one of the field players the complete turn is repeated until the ball was found. The other field player starts patrolling over the field. Which field player repeats the turn process and which player starts patrolling is determined by the number of playing robots and their current position on the field, so that always a single field player or the field player which is nearer to the own goal will patrol to another position on the field to be able to see the ball (cf. Fig. 6.5). After the specified position is reached, the robot scans again the area in front of it and begins to turn again.

There is a special handling for the case that an opponent robot stood right in front of the striker when it loses the ball. In this case the robot does not start immediately with the

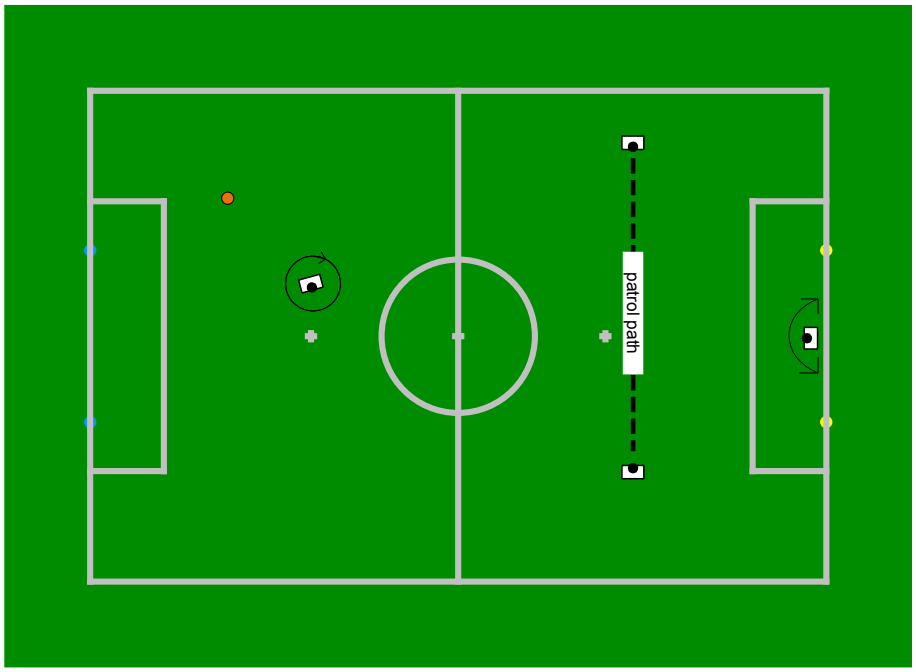


Figure 6.5: Visualization of the search-for-ball procedure. Robots at the end of the line depict stop positions, arrows indicate rotations of the robots.

turn process. Instead the robot walks some steps backwards and looks left and right with the assumption that the ball rolled a bit behind it. If the robot could not find the ball by walking backwards, it starts the scan-and-turn sequence.

kick-off. Another special situation is the kick-off. When the own team has kick-off, it is not allowed scoring 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. This year different kick-off variants were used that were selected randomly in general. These are a dribble kick-off, a single kick-off and two multi-player kick-offs. Although the kick-off used was selected randomly as described above, there are special constraint concerning the number of playing robots and the chosen tactic. That means that in case of less than three players on field or a defensive tactic the selection takes place between the single kick-off and the dribble kick-off. Otherwise the selection takes place between all possible kick-off variants. In the following, all kick-off variants are described in detail. Thereby all kick-off variants have in common that the robot executing the kick-off has to walk behind the ball before kicking it in any of the chosen directions.

The single kick-off is used if only one or two robots are on the field. Then 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 that could receive a pass (only keeper or defender, or no other robot is on the field).

The dribble kick-off is used for dribbling the ball out of the center circle instead of kicking it. In this variant the robot walks towards the ball until it is near enough for dribbling. Afterwards the robot dribbles the ball outside the circle by using the foot in front of which the ball is lying. Since it is nearly impossible to lead the robot behind the ball in a way that the ball is precisely in front of a foot and because of a randomness in the foot used for dribbling, an uncertainty about the dribbling direction is included by this variant. This is

an advantage because it is difficult for the opponents to react on a random ball movement, and furthermore it is no disadvantage for us, because the only goal in this situation is to get the ball out of the circle *anyhow*.

The normal kick-off is used if a supporting robot is available and no defensive tactic is chosen. In this case a sideways kick is executed by the striker, where 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 sidewards towards the supporting robot until it leaves the center circle.

The special kick-off is also used when a supporting robot is available and no defensive tactic is chosen. In general it is similar to the normal kick-off except of the direction of the kick. After the kick-off selection took place and this kind of kick-off was selected, the striker sends a message to the supporting robot. The robot receiving the message starts to walk forward towards the opponent goal until a defined position is reached. Meanwhile the striker walks to the ball and kicks it diagonally towards the supporter, which can take on the ball and kick it into the opponent goal ideally. The idea behind this kind of kick-off was to get the ball forward towards the opponent goal and to get the ball away from (and with some luck even behind) the opponent striker.

hand_to_hand_situation. When the striker walks from behind to the ball, but an opponent robot is detected between the ball and goal so that no forward kick is possible, a hand-to-hand-situation appears. In this case the distance to the opponent robot decides about what action will be executed. If the distance to the opponent is high enough, a normal sidewards kick is executed in case a supporter is available, or a diagonal one otherwise. In the case that the distance to the opponent player is much smaller, it is randomly chosen whether the robot dribbles the ball or executes a very fast kick which is implemented directly within the walking engine (cf. Sect. 5.2.3). Both possibilities are used to get the ball away or around the opponent robot as fast as possible. This is not possible with other kicks at the moment, because they take too much time to be executed.

In the case that the robot is close to the sideline, the direction of the diagonal kick and the walking engine kick are selected by the robot position to always kick towards the middle.

dribble_in_front_of_goal. 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 situations in which the striker dribbles are in a hand-to-hand situation and when the ball is lying on (or almost on) the opponent goal line. In the second case the robot aligns behind the ball and dribbles it into the goal by walking against the ball.

pass_in_front_of_goal. If 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 (which is the case when the ball is lying in a corner next to the opponent goal), 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.

walk_next_to_keeper. In some cases the keeper walks towards the ball. These situations are on the one hand if the ball is inside the own penalty area, and on the other hand if the ball is near the own goal (outside the own penalty area) but the keeper is much nearer than any other field player. In this case the keeper walks towards the ball, the striker aligns relative to the keeper, positioning on the opposite field side than where the ball is located.

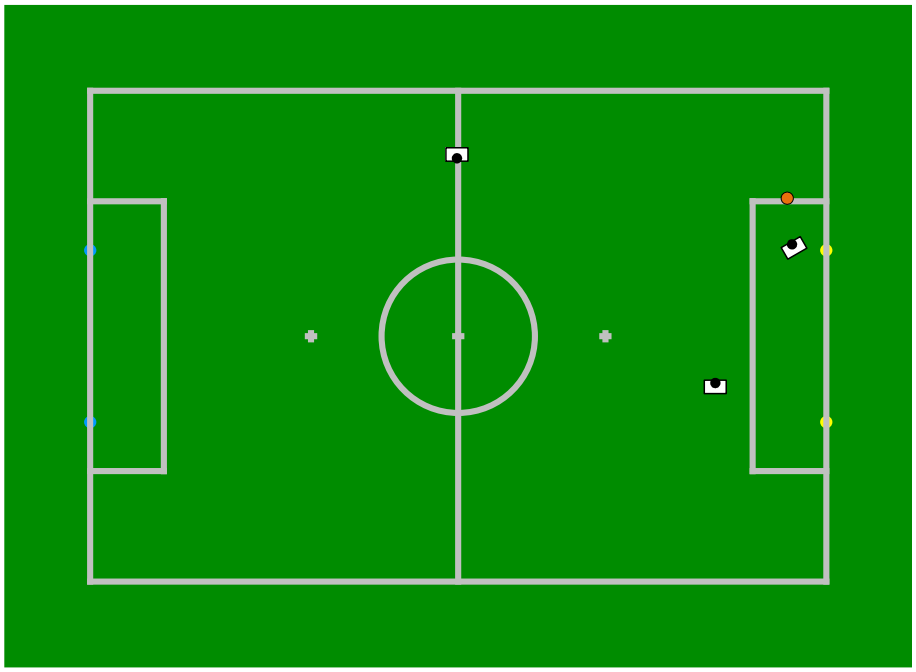


Figure 6.6: Positioning of the field players when the keeper decides to kick the ball away. One field player (striker) stays near the ball in front of the penalty area. The other field player (supporter) waits for the ball at the middle line.

For the positioning of the striker a similar potential field as for the supporter is used (see below). The idea behind that is to still have a field player near the ball that does not disturb the keeper in its action (cf. Fig. 6.6).

dribble_instead_sidewards_kick. This dribbling is used if an opponent robot is near the ball facing towards the own goal and the own robot walks to the side of the ball to execute a sidewards kick. In this situation the robot doesn't execute the sidewards kick but dribbles the ball forwards. This is done to prevent that the opponent robot is able to kick the ball towards the own goal while the own robot begins to execute a sideways kick. This dribbling moves the ball a little bit to the side to which the own robot can walk very quickly for kicking it afterwards while the opponent robot either ends its kick or has to search for the ball.

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. 4.2.5). 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.

To more easily adjust the distance, which the striker holds to the obstacle, the entire parameters of this option are saved in a configuration file.

go_to_ball. When no special situation is active (such as kick-off, obstacle avoidance, or behavior near the goal) and the ball is not directly in front of the striker's feet, it walks towards the ball.

To reach a good position near the ball the *KickPose* (cf. Sect. 6.3.4) is used and the striker walks to the provided pose. Therefore the coordinates are passed to the walking engine (cf. Sect. 5.2.3) as a target. To reach the ball in an acceptable time the striker first turns and then walks straight towards the ball when the distance from ball to it is quite high. In case that the striker's position is between ball and goal or on a level with the ball and the pose is behind the ball, it has to walk around the ball until the pose is reached. Whether the robot turns left or right around the ball depends on the angle to the ball and to the goal.

After reaching the pose, the striker stops in front of the ball and is ready for the decision which appoints the next option. The next possible options are a kick, a pass, or a hand to hand situation. This depends on the situation in which the opponent is. The possibilities are that the opponent goal is free, an opponent robot is blocking its goal, or it is standing too close so that the striker is not able to perform a pass.

kick. Which kick will be executed is selected by the *KickPoseProvider* (cf. Sect. 6.3.4). The following kicks or kick directions were used at RoboCup 2010: forward (with variable strength), backward, sideways, and diagonally. After reaching the *KickPose* the striker executes the chosen kick immediately.

6.3.7.2 Supporter

The supporter's role is used only when there are three robots on the field. Its task is to support the striker. As already mentioned in Sect. 6.3.5, there are two different supporters (offensive/normal and defensive), in which the offensive supporter is similar to the normal one. Because of that the normal supporter is described in detail first and afterwards only the differences of the defensive supporter are mentioned.

The main purpose of the supporter is, as the name indicates, to support the striker. This is done by positioning the robot next to the striker with a defined offset. Besides positioning several special operations exist. The supporter is, for example, involved in two of the three kick-off variants, where the striker passes towards the supporter, and hence the supporter has to go to defined positions. Moreover the search-for-ball-behavior mentioned above is used for the supporter as well as the obstacle avoidance, and a special positioning, when the keeper decides to walk towards the ball. In the latter case the supporter positions on the middle line facing towards the center circle where the field side depends on the position of the ball seen by the teammates (cf. Fig. 6.6). Another special situation is the case that the ball is lying next to the opponent goal and the striker wants to execute a pass. In this case the supporter leaves its regular position and walks to a position in front of the goal to receive the pass. The last special handling is used directly after the playing state begins. If the opponent team has kick-off, the supporter waits five seconds until it follows the striker. This shall prevent dangerous situations if the striker is not able to reach the ball before the other team does.

In any other case the supporter positions next to the striker, where the side is kept until the target position comes too close to the field border. This way most of the field is covered and an oscillation of positions is prevented.

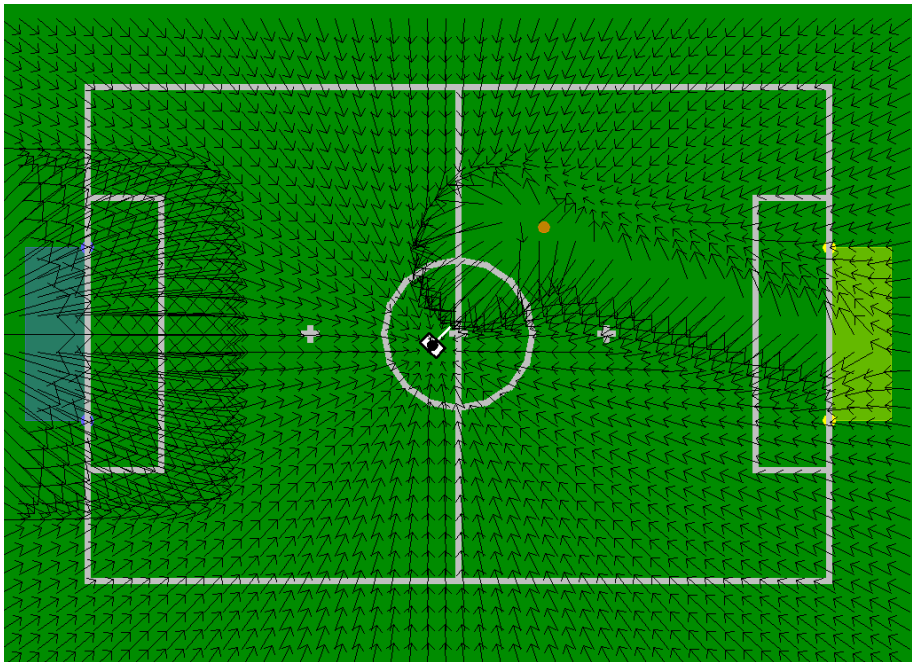


Figure 6.7: Visualization of the potential field for positioning the supporter. The components that are combined can be clearly seen: the rejection of the own penalty area, the attraction of the target position, the circular rejection of the striker with a bias to the own goal, and the rejection of the line between ball and opponent goal.

The positioning of the supporter is realized by a potential field [10] (cf. Fig. 6.7). The decision to use a potential field instead of just letting the robot walk to the desired position was taken because the supporter is supposed not to disturb the striker. The potential field is a combination of several basic potential fields that are described below. The output of the potential field is the desired walking direction of the robot, where the supporter should always be turned towards the ball, except when the target position is far away.

Attraction of target position. Since the primary goal of the supporter is to reach the desired target position, it is attracted by this position. The intensity of this potential field is proportional to the distance to the target, up to a certain maximum. This way the walking speed is reduced when the robot comes near the target position.

Rejection of the striker. The supporter is supposed to not disturb the striker. That is why it is rejected by the position of the striker. This way it can be ensured that the supporter does not come too close to the striker, e. g. when the striker blocks the direct way to the target position.

Rejection of the ball-goal-line. The striker is the player that is supposed to score goals. To ensure that the supporter does not obstruct the striker in this purpose, it is rejected from the ball-goal-line. This ensures that the striker does not accidentally hit the supporter when trying to score a goal.

Attraction towards the back of the striker. The combination of the previous two potential fields can lead to a local minimum in front of the striker, which is a bad position for

the supporter. To prevent this situation an additional potential field is introduced that moves the supporter behind the striker.

Rejection of the own penalty area. Since all players except the goal keeper are disallowed to enter the own penalty area (this would be an "'illegal defender"' according to the rules), the supporter is rejected from the own penalty area. This behavior is used for the case that the ball, and hence the striker and also the supporter, get close to the own penalty area.

The main difference between the several supporters (normal, offensive, defensive) is the offset relative to the striker. The normal and the offensive supporter are positioned approx. 1.1 m to the side and 0.3 m to the back relative to the striker, whereas the defensive supporter is positioned further backwards, approx. 1 m relative to the striker. Another difference is that the offensive and defensive supporters (that currently only appear simultaneously in the offensive tactic) do not change the side when the target position is near the field border, because in the offensive tactic there is one supporter on each side of the striker, so that the supporters would disturb each other if they would change sides or share one side of the striker.

6.3.7.3 Defender

The defender's task is, in general, to defend the own goal without entering the own penalty area. The defender only appears if the keeper is out of the field, or if the defensive tactic is chosen. Similar to the behavior of the keeper, the defender tries to position between the ball and the own goal. It has a fixed x-position on the field and only moves to the left and right, i. e. it only moves on an imaginary line located in front of the own penalty area (cf. Fig. 6.8). The positioning of the robot is realized with the potential field that is also used for positioning the supporter (cf. Sect. 6.3.7.2). Also the obstacle avoidance is completely the same as for the striker.

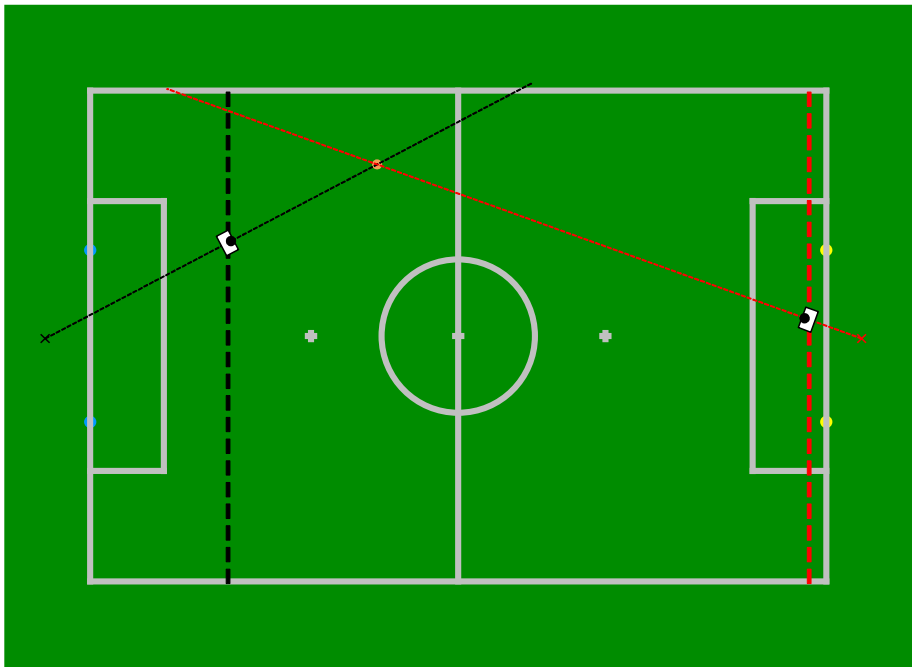


Figure 6.8: Imaginary lines for positions of defender and keeper

In the case that the keeper is off the field, this role is necessary to still have any player on the field that defends the own goal, and in the case that the defensive tactic is chosen, this role is used to increase the defense capabilities of the own team. The positioning of the defender differs for the case that the defensive tactic is chosen in that the robot is displaced approx. 50 cm to the side to let the keeper have the ball in sight all the time. In contrast to last year, the defender is not equipped with the capability to kick the ball, because the role selection switches to the striker's role if a robot is in a good position to kick the ball.

6.3.7.4 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.

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. 6.3.7.3), but in contrast the keeper sits down when it has reached a good position (cf. Fig. 6.9). 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 and self-localization.

If the ball is rolling towards the goal, the goalie has to select from a given set of actions: staying crouched, spreading the legs, or diving. The decision is made by considering the velocity and the distance of the ball. The estimates are used to calculate both the remaining time until the ball will intersect the lateral axis of the goalie and the position of intersection. If the ball rolls straight towards the goalkeeper, or it is expected to clearly miss the goal, the keeper remains crouched. In case of a close intersection, the goalie changes to a wide defensive posture to increase its range (for approximately four seconds, conforming to the rules of 2010). The diving is initiated when the ball intersects the goalkeeper's lateral axis in a position farther away from the farthest possible point of the defensive posture.

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. 6.3.7.1). 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.

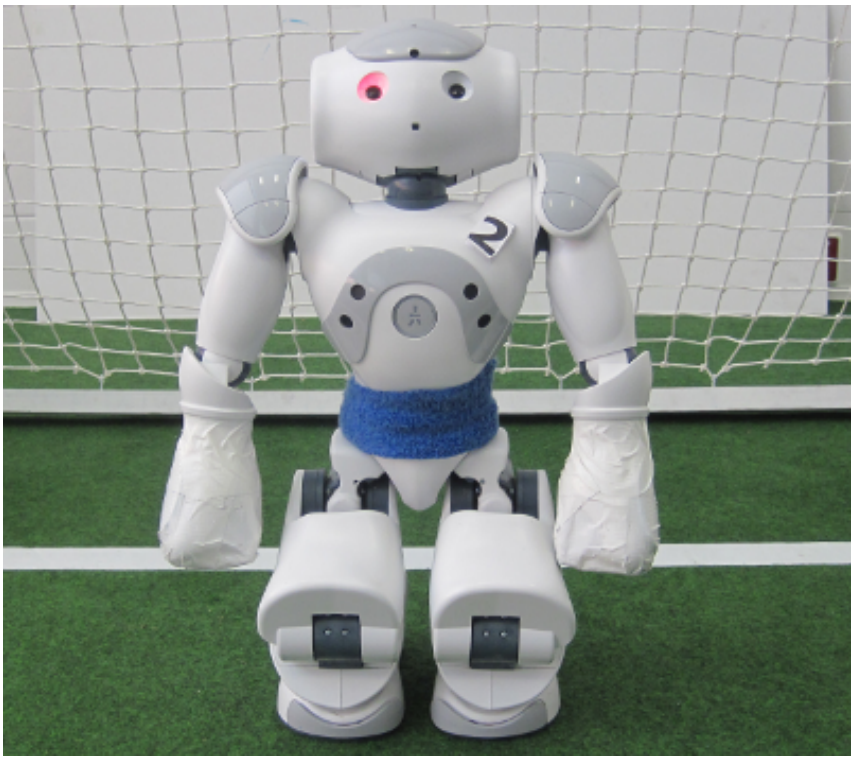


Figure 6.9: Sitting position of the keeper

6.3.8 Penalty Control

Penalty control is responsible to stop the robot's movement while being 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 implement the 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 it 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.

6.3.9 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.

6.3.9.1 Right Eye

Color	Role	Additional information
Blue	All	No ground contact
Blue	Keeper	Flashes if the ball comes up to the keeper
White	Defender	
Green	Supporter	Flashes if the supporter is active
Red	Striker	
Orange	Offensive Supporter	
Magenta	Defensive Supporter	

6.3.9.2 Left Eye

Color	Information
White	Ball was seen
Green	Goal was seen
Red	Ball and goal were seen
Blue	No ground contact

6.3.9.3 Torso (Chest Button)

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

6.3.9.4 Feet

- The left foot shows the team color. If the team is currently the red team the color of the LED is red, otherwise blue.
- The right foot shows whether the team has kick-off or not. If the team has kick-off, the color of the LED is white, otherwise it is switched off. In case of a penalty shootout, the color of the LED is green.

6.3.9.5 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 all LEDs are off, no player is connected. If half of the LEDs are on, one player is connected, and if all LEDs are on, two players are connected to the current robot.

Chapter 7

Challenges

For the RoboCup 2010 Technical Challenge, B-Human has developed approaches for all three sub-challenges. Unfortunately we did not score in the Dribble Challenge but we placed second at the open challenge and third in the passing challenge. This section describes our approaches.

7.1 Dribble Challenge

The aim of the dribble challenge was to shoot a goal, but neither the robot, nor the ball were allowed to touch any obstacles on the field. Three robots in their crouching position were used as obstacles, which were placed on the field in such a way to prevent direct kicks towards the goal. The dribbling robot had three minutes to score a goal from its starting position, which was the front line of the penalty box of the blue goal. The ball was placed on the penalty mark in front of the robot.

For the dribble challenge a special behavior was implemented, which used some modified skills of the regular soccer behavior and some new skills for dribbling and obstacle avoidance were added. The whole dribble challenge behavior can be divided up into a more or less high level part and a low level part. The high level behavior tells the robot to dribble on a predefined path, whereas the low level behavior does the actual dribbling.

To dribble along the predefined path, the robot first randomly chooses a side. Then it dribbles the ball around the center circle to the side it initially had chosen. After that the robot dribbles to a spot between the opponent's penalty mark and penalty line. If the distance to the goal is less than one meter, the robot tries to kick a goal. Therefore the regular kick option of the soccer behavior is used. Unfortunately our vision could not detect the sitting robots reliably, so we implemented this hardcoded behavior. We chose exactly that path because we expected that an obstacle had to be somewhere in the center so that, according to the rules, the direct path to the goal was blocked. If there is an obstacle along the way of the robot, it should detect it with its sonar sensors and then dribbles around it.

The dribbling behavior works as follows: the robot does not shoot the ball forward, but runs slowly against it with one foot, so that the ball rolls a bit ahead. Depending on the shortest distance to the ball, the robot chooses the right or the left foot to reduce the time needed to align behind the ball. We did not use kicks at all, because at such a slow pace the ball did not always roll straight ahead and it seems to be very risky to kick without knowledge of the obstacle positions. If the sonar detects an obstacle anywhere up to 90 cm in front of it, the robot will try to dribble around it by walking to the ball and turning around it until the measured distance is greater than one meter. Otherwise, if there are obstacles on the left or right side in a range less

than 20 cm, the robot dribbles forward until the distance to these obstacles is greater than 90 cm. If there are no more dangerous obstacles around, the robot continues to dribble straight to the goal.

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 third 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 is played, the challenge is finished successfully. Since it is very difficult to satisfy all these restrictions in just one attempt, the last rule indicates that any number of attempts can be done within the count of three minutes. The rule also implies that all three valid passes must be achieved in one attempt. If more than one participant completes one successful attempt, the time over all tries counts.

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 of 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 and searching for the ball if the ball was not seen by it or its teammate 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 than 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 a 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

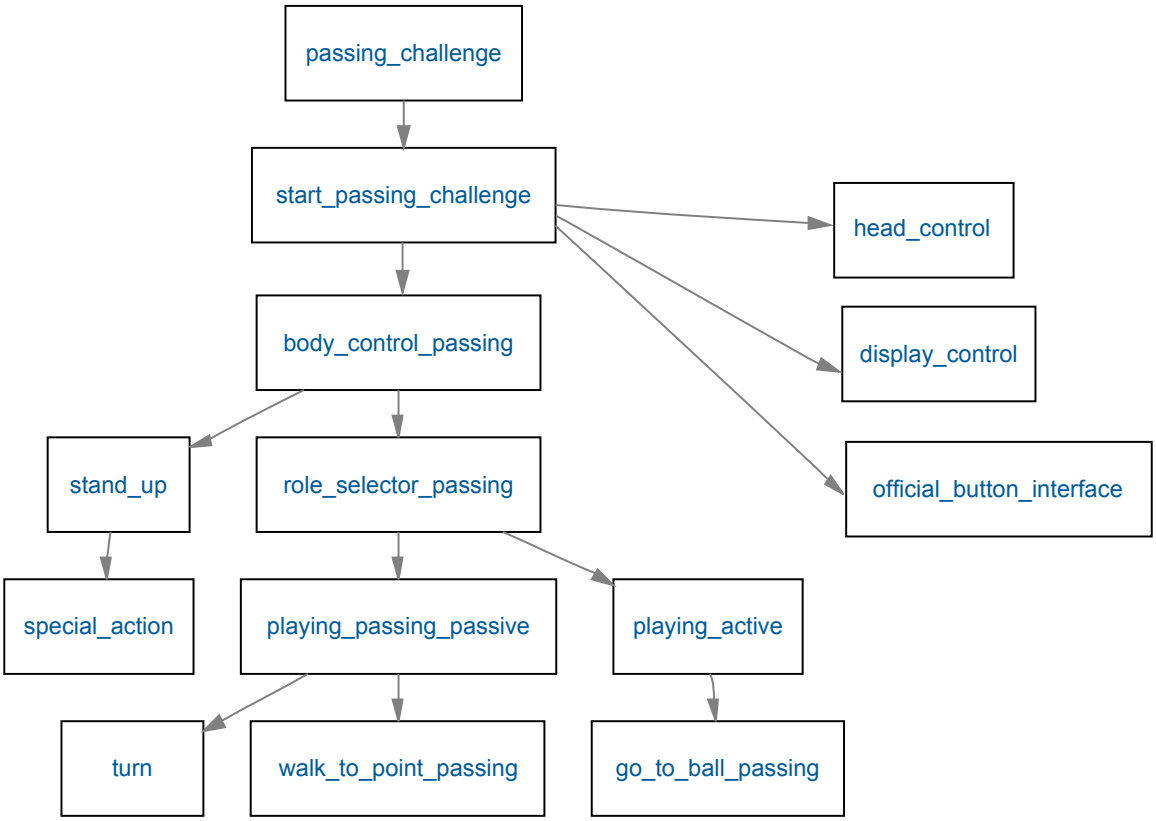


Figure 7.1: A brief overview of the passing challenge behavior option graph.

leaves the field, which is one of the conditions to fail the challenge attempt. If the pass is too short and the ball stays in the middle area – that no robot is allowed to enter – the challenge attempt 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 adapted version of the dynamic kick of our soccer behavior was used. In this context the kick was enhanced with the possibility to set the strike out position of the foot dynamically in order to the target position to obtain a variable trajectory of the ball.

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. If the active robot discovers the ball in the other half of the field or if the ball was not seen by the active robot for more than ten seconds, his role also changes back to passive. 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 (shown in figure 7.1) is based on the default soccer behavior to obtain the official button control, the LED assignment and the reaction to the GameController.

7.3 Open Challenge: Throw-in in the Standard Platform League

Motivation. Since 2005, the robots of the Standard Platform League are able to kick the ball out. Since then, the procedure has always been the same. An assistant referee picks the ball up

and puts it back inside the field at a position that is disadvantageous for the team that touched the ball last before it went out. Given that there are enough field players, each team could have a robot waiting at the position the ball would most likely be placed back, i.e. the penalty for kicking out the ball was often quite low. The throw-in rule never gave the team that did not kick the ball out actual control over the ball. So it is time to change that by introducing a new rule that lets a Nao actually throw-in the ball.

Throwing-in the Ball. To do so, we presented a behavior in which a robot walks to the ball, picks it up with both hands and throws it towards a teammate. The critical part is grasping the ball. We model the ball's relative position in 3-D using a Kalman filter to ensure precision. Then, the grasp points are determined in Cartesian space as well as trajectories for both hands from those positions to positions over the head. Inverse kinematics is used to transform these trajectories to joint angle space, taking joint angle limits into account. Actually, the target 6-D positions of the hands are always slightly inside the ball to exert enough force to keep the ball between the hands. A final motion will actually throw the ball. While the arms are controlled dynamically, the movements of the other parts of the body are currently fixed joint angles sequences. The underlying approach is not limited to picking up a ball from the ground plane but is aiming to do actual pick and place operations with the Nao Academic Edition, i. e. with a Nao with actuated hands. However, in the presentation, the RoboCup Edition was used.

Integration into the Game. It seems unrealistic to perform a throw-in without any intervention of the referees, because often the ball would leave the carpet if it had not been stopped by a referee. Therefore, when the ball goes out, an assistant referee puts it back on the sideline where it left the field. The GameController sends the number of the team that kicked the ball out together with a counter that reports the time since that event (both is already the case). For a limited amount of time, only a single robot of the other team is allowed to walk to the ball and throw it in. The other robots have to keep a minimum distance of 50 cm to the ball. The game continues when the time is up or the ball was thrown.

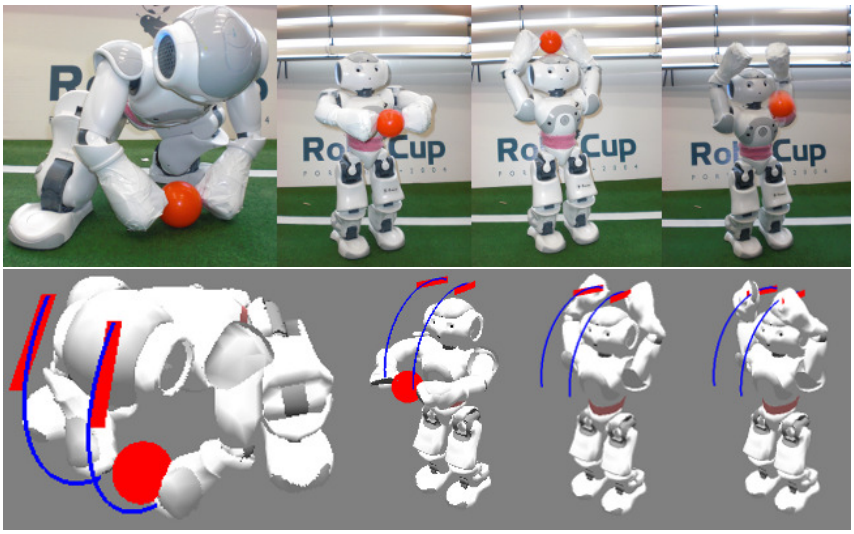


Figure 7.2: Nao throwing the ball. The lower image sequence shows the trajectories planned for the arms (blue: picking up, red: throwing).

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. 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, e. g., by double-clicking on the root element. 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 ten 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*, *module views*, and the *kick view*. Field, image, and plot views display debug drawings or plots received from the robot, whereas the other views visualize certain color channels, the current color table, specific information

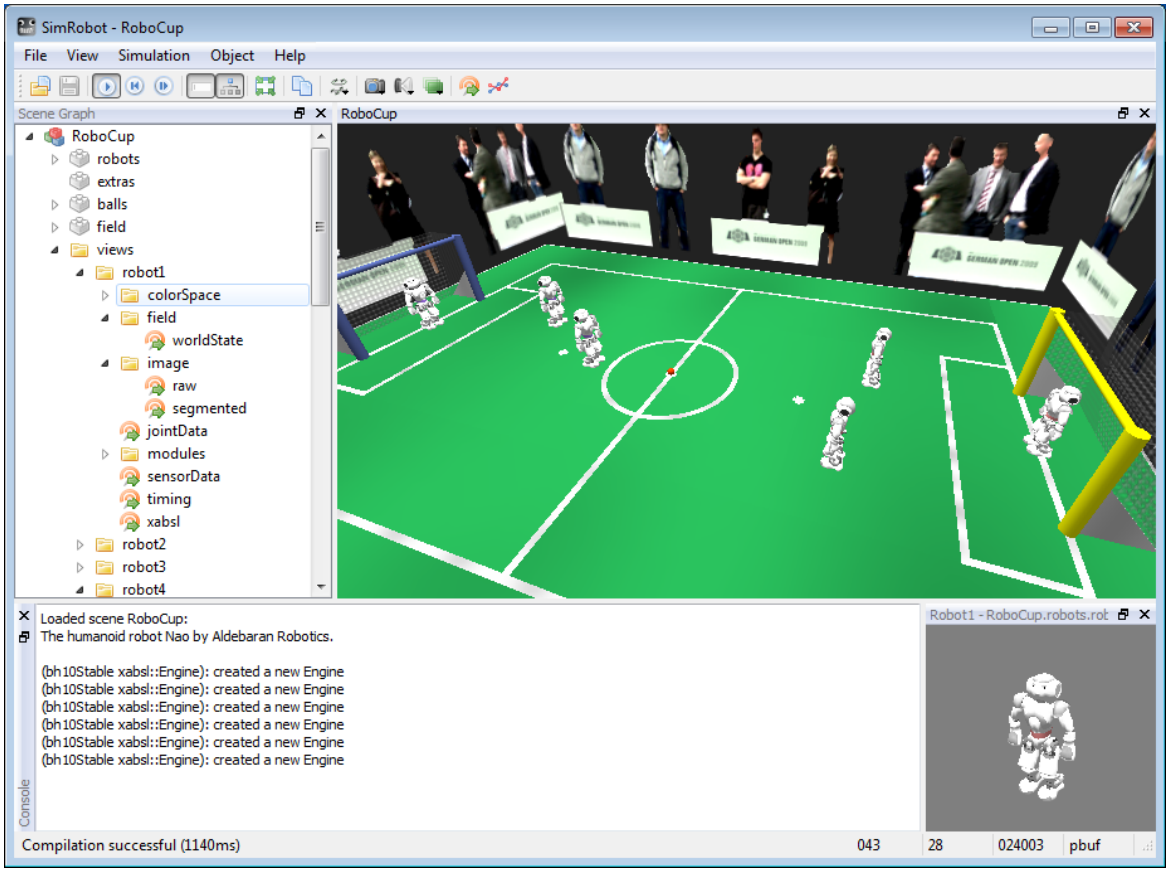


Figure 8.1: SimRobot with the scene graph on the left and two scene views on the right. The console window is shown at the bottom.

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 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*

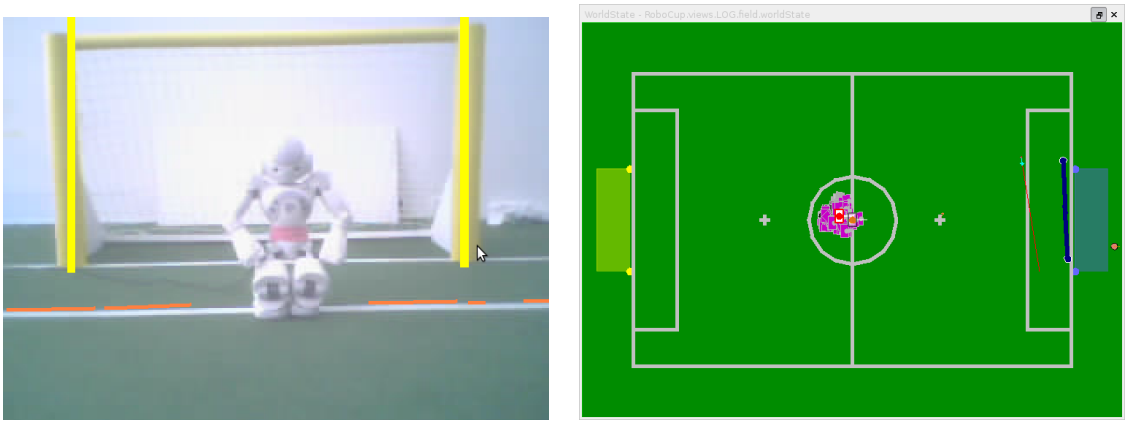


Figure 8.2: Image view and field view with several debug drawings

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.

Mouse Wheel / Page Up, Page Down. Zooms the image in or out.

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).

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

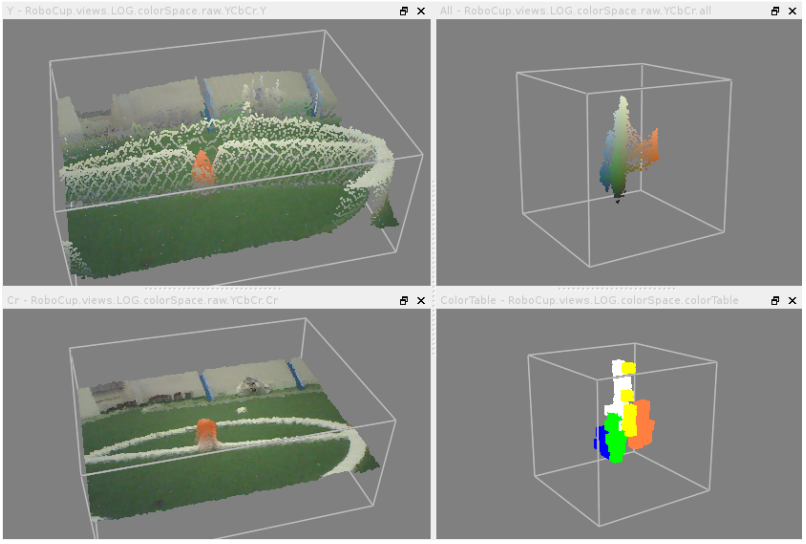


Figure 8.3: Color channel views, image color space view, and color table view

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

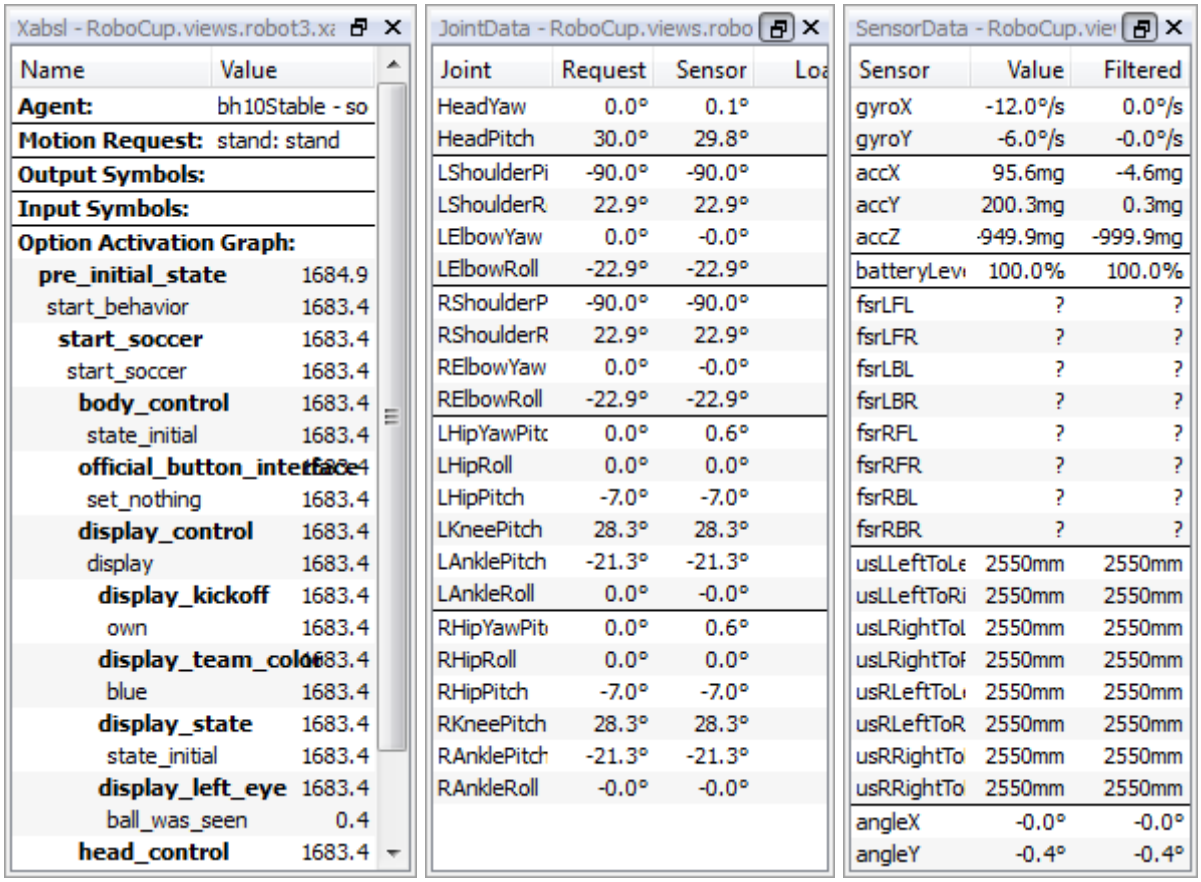


Figure 8.4: Xabsl view, joint data view and sensor data view

```
vfd worldState representation:RobotsModel:robots

# 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.

The field can be zoomed in or out by using the *mouse wheel* or the *page up/down* buttons. It can also be dragged around with the left mouse button.

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. 8.4). 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
```

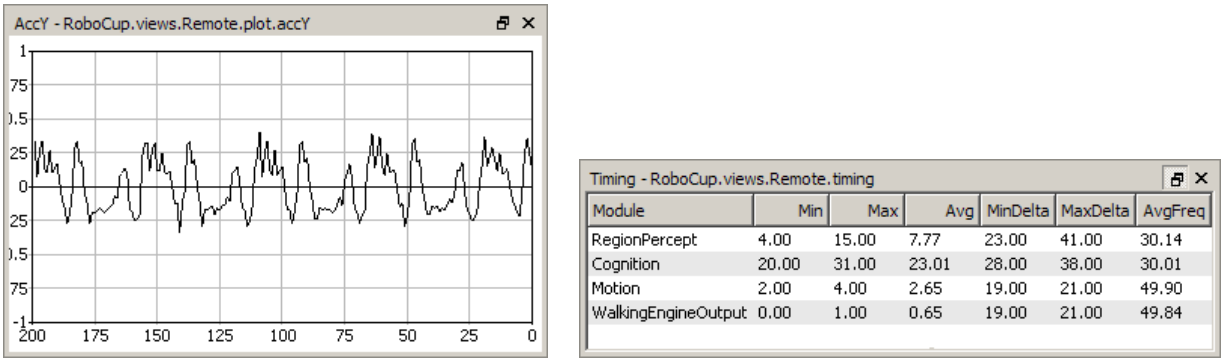


Figure 8.5: The plot view and the timing view

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).

The font size used can be changed by using the *page up/down* buttons.

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:

```
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 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 [2] 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*.

The module graph can be zoomed in or out by using the *mouse wheel* or the *page up/down* buttons. It can also be dragged around with the left mouse button.

8.3.10 Kick View

The basic idea of the kick view shown in Fig. 8.7 is to visualize and edit basic configurations of motions for the dynamic motion engine described in [20]. In doing so the central element of this view is a 3-D robot model. Regardless if the controller is connected to a simulated or real robot, this model represents the current robot posture.

Since the dynamic motion engine organizes motions as a set of Bézier curves, the movement of the limbs can easily be visualized. Thereby the sets of curves of each limb are also represented by combined cubic Bézier curves. Those curves are attached to the 3-D robot model with the robot center as origin. They are painted into the 3-dimensional space. Each curve is defined by equation 8.1:

$$c(t) = \sum_{i=0}^{n} B_{i,n}(t) P_i \tag{8.1}$$

To represent as many aspects as possible, the kick view has several sub views:

3-D View. In this view each combined curve of each limb is directly attached to the robot model and therefore painted into the 3-dimensional space. Since it is useful to observe the

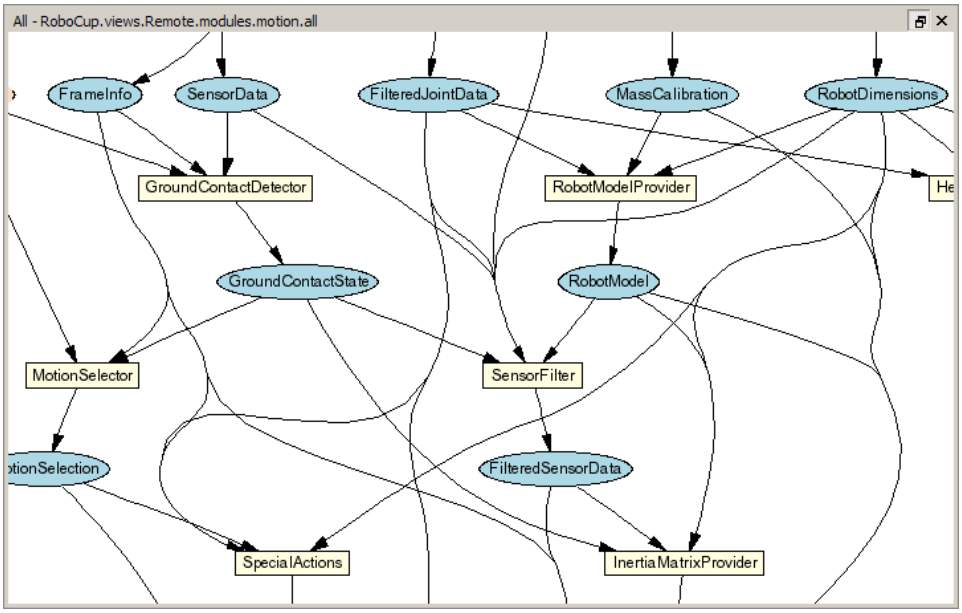


Figure 8.6: The module view showing part of the modules in the process *Motion*

motion curves from different angles, the view angle can be rotated by clicking with the *left mouse button* into the free space and dragging it in one direction. In order to inspect more or less details of a motion, the view can also be zoomed in or out by using the *mouse wheel* or the *page up/down* buttons.

A motion configuration is not only visualized by this view, it is also editable. Thereby the user can click on one of the visualized control points (cf. Fig. 8.7 at E) and drag it to the desired position. In order to visualize the current dragging plane a green area (cf. Fig. 8.7 at B) is displayed during the dragging process. This area displays the mapping between the screen and the model coordinates and can be adjusted by using the *right mouse button* and choosing the desired axis configuration.

Another feature of this view is the ability to display unreachable parts of motion curves. Since a motion curve defines the movement of a single limb, an unreachable part is a set of points that cannot be traversed by the limb due to mechanic limitations. The unreachable parts will be clipped automatically and marked with orange color (cf. Fig. 8.7 at C).

1-D/2-D View. In some cases a movement only happens in the 2-dimensional or 1-dimensional space (for example: Raise a leg is a movement along the z-axis only). For that reason more detailed sub views are required. Those views can be displayed as an overlay to the 3-D view by using the buttons *Show 1D* and *Show 2D* on the left (cf. Fig. 8.7 at D).

Because clarity is important, only a single curve of a single phase of a single limb can be displayed at the same time. If a curve should be displayed in the detailed views, it has to be activated. This can be done by clicking on one of the attached control points.

The 2-D view (cf. Fig. 8.8) is divided into three sub views. Each of these sub views represents only two dimensions of the activated curve. The curve displayed in the sub views is defined by equation 8.1 with $P_i = \binom{c_{x_i}}{c_{y_i}}$, $P_i = \binom{c_{x_i}}{c_{z_i}}$ or $P_i = \binom{c_{y_i}}{c_{z_i}}$.
The 1-D sub views (cf. Fig. 8.8) are basically structured as the 2-D sub views. The difference is that each single sub view displays the relation between one dimension of the activated curve and the time t. That means that in equation 8.1 P_i is defined as: $P_i = c_{x_i}$, $P_i = c_{y_i}$, or $P_i = c_{z_i}$.

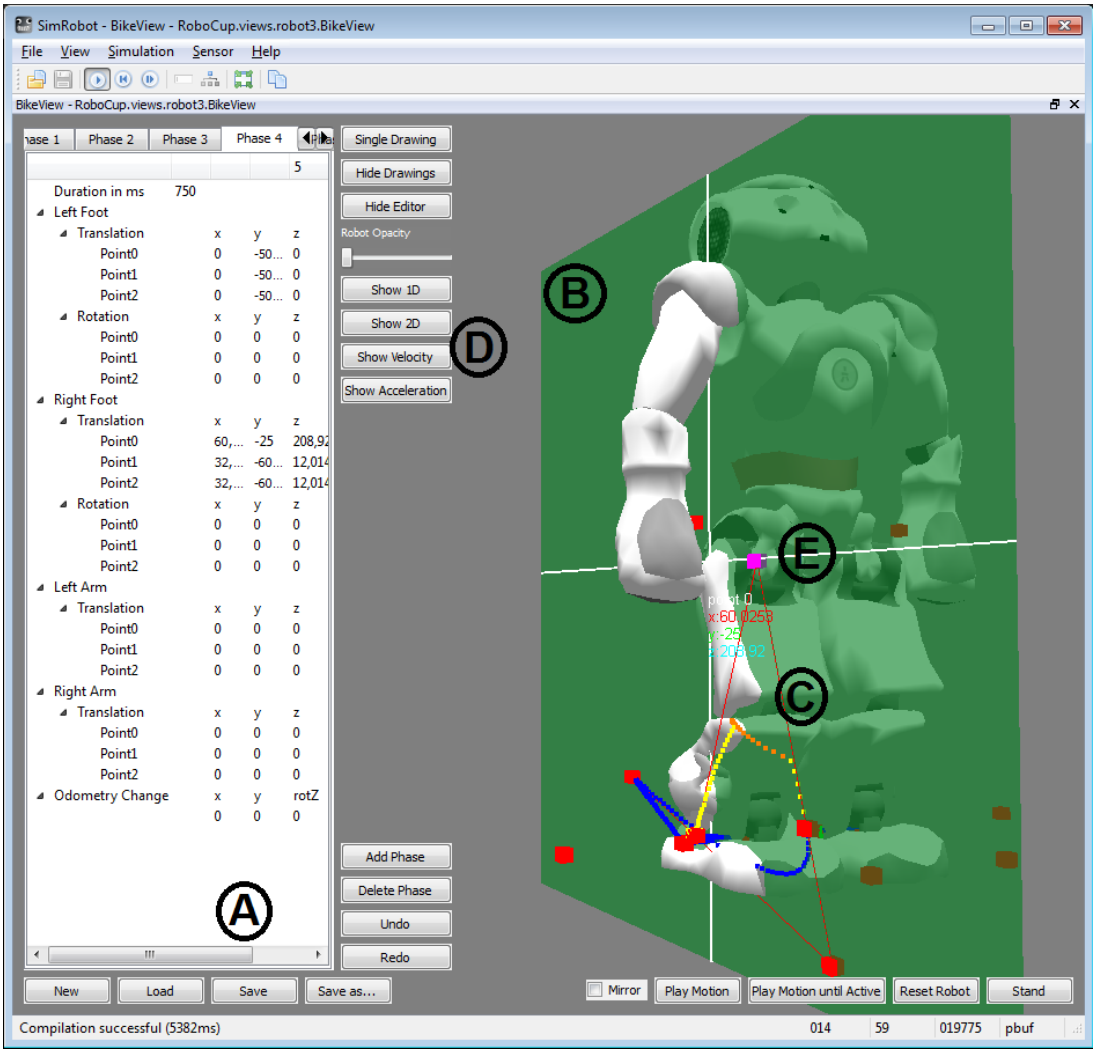


Figure 8.7: The kick view. *A* marks the editor view, *B* denotes the drag and drop plane, *C* points at a clipped curve, *D* tags the buttons that activate more detailed overlays, and *E* labels one of the control points.

As in the 3-D view the user has the possibility to edit a displayed curve directly in any sub view by drag and drop.

Editor Sub View. The purpose of this view is to constitute the connection between the real structure of the configuration files and the graphical interface. For that reason this view is responsible for all file operations (for example open, close, and save). It represents loaded data in a tabbed view, where each phase is represented in a tab and the common parameters in another one.

By means of this view the user is able to change certain values directly without using drag and drop. Values directly changed will trigger a repainting of the 3-D view, and therefore, changes will be visualized immediately. This view also allows phases to be reordered by drag and drop, to add new phases, or to delete phases.

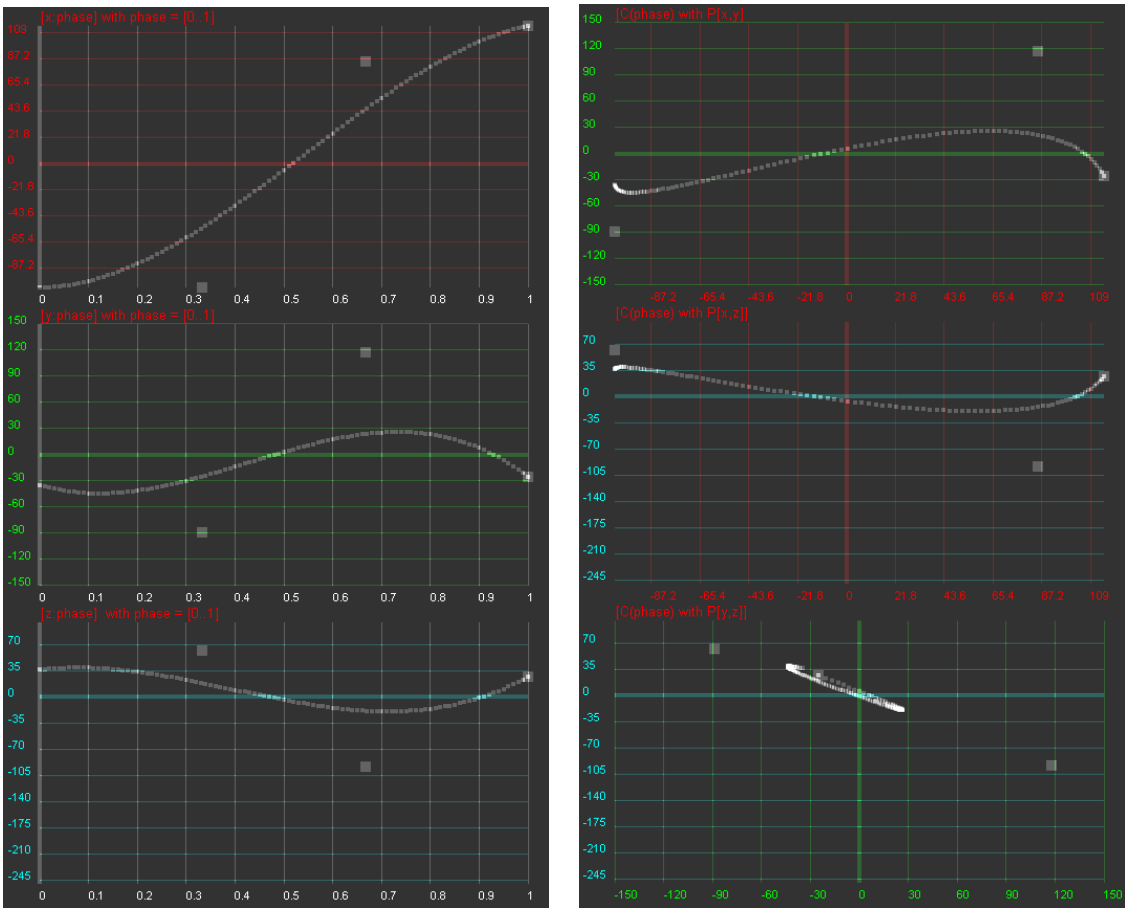


Figure 8.8: Left: 1-D sub views. Right: 2-D sub views

8.4 Scene Description Files

The language of scene description files is RoSiML [3]. In the main *.ros* files, such as *BH2010.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.

scwp <name> <a.b.c.d>. Starts a wireless connection to a real robot and attaches a 3-D robot model with it. The first parameter *name* defines the identifier that will be used for the robot and its model. The second parameter specifies the IP address of the real robot. Similar to the command *sc* this command will add a new robot to the list of available robots, but will also add a robot model to the list of available *puppets* using *name*. The connection established between the model and the real robot will take care that all data measured by the robot – such as the joint angles – are applied to the model. The connection will be terminated, when the simulation is reset or the simulator is exited.

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.

bike. Adds a kick view to the scene graph.

cameraCalibrator <view> (on | off) This command activates or deactivates the CameraCalibrator module for the given view. By default you may want to use the "raw" view. For a detailed description of the CameraCalibrator see Sect. 4.1.1.1.

ci off | on [<fps>]. Switches the calculation of images on or off. With the optional parameter *fps*, a customized image frame rate can be set. The default value is 30. 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 (1** | **2)** <**command**> | **(press** | **release)** <**button**> <**command**>**.** Sets a joystick command. If the first parameter is *press* or *release*, the number following 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 that will be executed in every frame while the corresponding button is pressed. The prefixes *press* or *release* 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*, an analog joystick command is defined. There are two slots for such commands, number 1 and 2, e. g., to independently control the robot's walking direction and its head. 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.

jm <**axis**> **(off** | <**button**> <**button**> **).** Maps two buttons on an axis. Pressing the first button emulates pushing the axis to its positive maximum speed. Pressing the second button results in the negative maximum speed. The command is useful when more axes are required to control a robot than the joystick used actually has.

js <**axis**> <**speed**> <**threshold**> [<**center**>]**.** Set axis maximum speed and ignore threshold for command *jc motion* <*num*>. *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*... *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. An optional parameter allows for shifting the center the center itself, e. g., to compensate for the bad calibration of a joystick.

log ? | **start** | **stop** | **pause** | **forward** [**image**] | **backward** [**image**] | **repeat** | **goto** <**number**> | **clear** | **(keep** | **remove)** <**message**> {<**message**>} | **(load** | **save** | **saveImages** [**raw**]) <**file**> | **cycle** | **once** | **full** | **jpeg.** 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 [**image**] | **backward** [**image**] | **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. With the optional parameter *image*, it is possible to step from image to image. *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.

full | jpeg. These two commands decide whether uncompressed images received from the robot will also be written to the log file as full images, or JPEG-compressed. When the robot is connected by cable, sending uncompressed images is usually a lot faster than compressing them on the robot. By executing *log jpeg* they can still be saved in JPEG format, saving a log memory space during recording as well as disk space later. Note that running image processing routines on JPEG images does not always give realistic results, because JPEG is not a lossless compression method, and it is optimized for human viewers, not for machine vision.

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.

mvb <x> <y> <z>. Moves the ball to the given metric position. x, y, and z have to be specified in mm. Note that the origin of the ball is about 32.5 mm above the ground.

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.

8.5.4 Input Selection Dialog

In scripting files it is sometimes necessary to let the user choose between some values. We therefore implemented a small input dialog. After the user has selected a value it will then be passed as input parameter to the scripting command. To use the input dialog you just have to type the following expression:

```
${<label>,<value1>,<value2>,...}
```

Inside the brackets there has to be a list of comma-separated values. The first value will be interpreted as a label for the input box. All following values will then be selectable via the dropdown list. The following example shows how we used it to make the IP address selectable. The user's choice will be passed to the command *sc* that establishes a remote connection to the robot with the selected IP address. Figure 8.9 shows the corresponding dialog.

```
sc Remote ${IP address:,10.0.1.1,192.168.1.1}
```

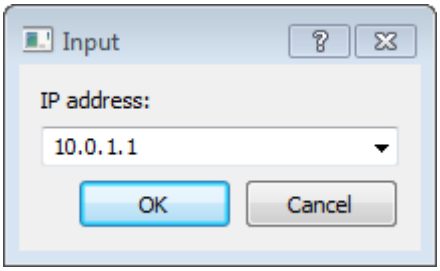


Figure 8.9: A dialog for selecting an IP address

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 = { mode = percentageSpeed; speed = { rotation = 0;
    translation = { x = 0; y = 0; }; }; target = { rotation = 0; translation
    = { x = 0; y = 0; }; }; pedantic = false; kickType = none; kickTarget =
    { rotation = 0; translation = { x = 0; y = 0; }; }; }; bikeRequest = {
    bMotionType = none; mirror = false; dynamical = false; dynPoints[0] = {
    }; }; sikeRequest = { motion = 5; mirror = false; }; }

jc press 2 set representation:MotionRequest { motion = stand;
    specialActionRequest = { specialAction = playDead; mirror = false; };
    walkRequest = { mode = percentageSpeed; speed = { rotation = $5;
    translation = { x = $2; y = $1; }; }; target = { rotation = 0; translation
    = { x = 0; y = 0; }; }; pedantic = false; kickType = none; kickTarget =
    { rotation = 0; translation = { x = 0; y = 0; }; }; }; bikeRequest = {
    bMotionType = none; mirror = false; dynamical = false; dynPoints[0] = {
    }; }; sikeRequest = { motion = 5; mirror = false; }; }

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

jc press 1 jc motion 2 set representation:MotionRequest { motion = walk;
    specialActionRequest = { specialAction = playDead; mirror = false; };
    walkRequest = { mode = percentageSpeed; speed = { rotation = $5;
    translation = { x = $2; y = $1; }; }; target = { rotation = 0; translation
    = { x = 0; y = 0; }; }; pedantic = false; kickType = none; kickTarget = {
```

```
rotation = 0; translation = { x = 0; y = 0; }; }; bikeRequest = {
bMotionType = none; mirror = false; dynamical = false; dynPoints[0] = {
}; }; sikeRequest = { motion = 5; mirror = false; }; }
```

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.
(`http://www.graphviz.org`)

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.xabsl.de`)

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.
(`http://host07.ath.cx/zbuildgen`)

Bibliography

[1] Dieter Fox, Wolfram Burgard, Frank Dellaert, and Sebastian Thrun. Monte-Carlo Localization: Efficient Position Estimation for Mobile Robots. In *Proceedings of the Sixteenth National Conference on Artificial Intelligence*, pages 343 – 349, Orlando, FL, USA, 1999.

[2] Emden R. Gansner and Stephen C. North. An Open Graph Visualization System and Its Applications to Software Engineering. *Software – Practice and Experience*, 30(11):1203–1233, 2000.

[3] 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`.

[4] Colin Graf, Alexander Härtl, Thomas Röfer, and Tim Laue. A Robust Closed-Loop Gait for the Standard Platform League Humanoid. In Changjiu Zhou, Enrico Pagello, Emanuele Menegatti, Sven Behnke, and Thomas Röfer, editors, *Proceedings of the Fourth Workshop on Humanoid Soccer Robots in conjunction with the 2009 IEEE-RAS International Conference on Humanoid Robots*, pages 30 – 37, Paris, France, 2009.

[5] Jens-Steffen Gutmann and Dieter Fox. An Experimental Comparison of Localization Methods Continued. In *Proceedings of the 2002 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2002)*, volume 1, pages 454–459, Lausanne, Switzerland, 2002.

[6] Laurie J. Heyer, Semyon Kruglyak, and Shibu Yooseph. Exploring Expression Data: Identification and Analysis of Coexpressed Genes. *Genome Research*, 9(11):1106–1115, 1999.

[7] V. Jagannathan, Rajendra Dodhiawala, and Lawrence S. Baum, editors. *Blackboard Architectures and Applications*. Academic Press, Boston, 1989.

[8] Simon J. Julier, Jeffrey K. Uhlmann, and Hugh F. Durrant-Whyte. A New Approach for Filtering Nonlinear Systems. In *Proceedings of the American Control Conference*, volume 3, pages 1628–1632, 1995.

[9] Shuuji Kajita, Fumio Kanehiro, Kenji Kaneko, Kiyoshi Fujiwara, Kazuhito Yokoi, and Hirohisa Hirukawa. A Realtime Pattern Generator for Biped Walking. In *Proceedings of the 2002 IEEE International Conference on Robotics and Automation (ICRA 2002)*, pages 31–37, Washington, D.C., USA, 2002.

[10] Oussama Khatib. Real-time Obstacle Avoidance for Manipulators and Mobile Robots. *The International Journal of Robotics Research*, 5(1):90–98, 1986.

[11] Tim Laue, Thijs Jeffry de Haas, Armin Burchardt, Colin Graf, Thomas Röfer, Alexander Härtl, and Andrik Rieskamp. Efficient and reliable sensor models for humanoid soccer robot

self-localization. In Changjiu Zhou, Enrico Pagello, Emanuele Menegatti, Sven Behnke, and Thomas Röfer, editors, *Proceedings of the Fourth Workshop on Humanoid Soccer Robots in conjunction with the 2009 IEEE-RAS International Conference on Humanoid Robots*, pages 22 – 29, Paris, France, 2009.

[12] Tim Laue and Thomas Röfer. Getting Upright: Migrating Concepts and Software from Four-Legged to Humanoid Soccer Robots. In Enrico Pagello, Changjiu Zhou, and Emanuele Menegatti, editors, *Proceedings of the Workshop on Humanoid Soccer Robots in conjunction with the 2006 IEEE International Conference on Humanoid Robots*, Genoa, Italy, 2006.

[13] Tim Laue and Thomas Röfer. Particle Filter-based State Estimation in a Competitive and Uncertain Environment. In *Proceedings of the 6th International Workshop on Embedded Systems*. Vaasa, Finland, 2007.

[14] Tim Laue and Thomas Röfer. SimRobot - Development and Applications. In Heni Ben Amor, Joschka Boedecker, and Oliver 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)*, Venice, Italy, 2008.

[15] Tim Laue and Thomas Röfer. Pose Extraction from Sample Sets in Robot Self-Localization - A Comparison and a Novel Approach. In Ivan Petrović and Achim J. Lilienthal, editors, *Proceedings of the 4th European Conference on Mobile Robots - ECMR'09*, pages 283–288, Mlini/Dubrovnik, Croatia, 2009.

[16] Tim Laue, Kai Spiess, and Thomas Röfer. SimRobot - A General Physical Robot Simulator and Its Application in RoboCup. In Ansgar Bredenfeld, Adam Jacoff, Itsuki Noda, and Yasutake Takahashi, editors, *RoboCup 2005: Robot Soccer World Cup IX*, volume 4020 of *Lecture Notes in Artificial Intelligence*, pages 173–183. Springer, 2006.

[17] Scott Lenser and Manuela Veloso. Sensor Resetting Localization for Poorly Modelled Mobile Robots. In *Proceedings of the 2000 IEEE International Conference on Robotics and Automation (ICRA 2000)*, volume 2, pages 1225–1232, San Francisco, CA, USA, 2000.

[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, China, 2006.

[19] Prasanta Chandra Mahalanobis. On the Generalised Distance in Statistics. In *Proceedings of the National Institute of Sciences of India*, volume 2, pages 49–55, 1936.

[20] Judith Müller, Tim Laue, and Thomas Röfer. Kicking a Ball – Modeling Complex Dynamic Motions for Humanoid Robots. In Eric Chown, Akihiro Matsumoto, Paul Plöger, and Javier Ruiz del Solar, editors, *RoboCup 2010: Robot Soccer World Cup XIV*, Lecture Notes in Artificial Intelligence. Springer, to appear in 2011.

[21] Thomas Röfer. Region-Based Segmentation with Ambiguous Color Classes and 2-D Motion Compensation. In Ubbo Visser, Fernando Ribeiro, Takeshi Ohashi, and Frank Dellaert, editors, *RoboCup 2007: Robot Soccer World Cup XI*, volume 5001 of *Lecture Notes in Artificial Intelligence*, pages 369–376. Springer, 2008.

[22] Thomas Röfer, Jörg Brose, Daniel Göhring, Matthias Jüngel, Tim Laue, and Max Risler. GermanTeam 2007. In Ubbo Visser, Fernando Ribeiro, Takeshi Ohashi, and Frank Dellaert, editors, *RoboCup 2007: Robot Soccer World Cup XI Preproceedings*, Atlanta, GA, USA, 2007. RoboCup Federation.

[23] Thomas Röfer, Tim Laue, Judith Müller, Oliver Bösche, Armin Burchardt, Erik Damrose, Katharina Gillmann, Colin Graf, Thijs Jeffry de Haas, Alexander Härtl, Andrik Rieskamp, André Schreck, Ingo Sieverdingbeck, and Jan-Hendrik Worch. B-Human Team Report and Code Release 2009, 2009. Only available online: `http://www.b-human.de/file_download/26/bhuman09_coderelease.pdf`.

[24] Thomas Röfer, Tim Laue, and Dirk Thomas. Particle-filter-based Self-localization using Landmarks and Directed Lines. In Ansgar Bredenfeld, Adam Jacoff, Itsuki Noda, and Yasutake Takahashi, editors, *RoboCup 2005: Robot Soccer World Cup IX*, volume 4020 of *Lecture Notes in Artificial Intelligence*, pages 608–615. Springer, 2006.

[25] Thomas Röfer, Tim Laue, Michael Weber, Hans-Dieter Burkhard, Matthias Jüngel, Daniel Göhring, Jan Hoffmann, Benjamin Altmeyer, Thomas Krause, Michael Spranger, Oskar von Stryk, Ronnie Brunn, Marc Dassler, Michael Kunz, Tobias Oberlies, Max Risler, Uwe Schwiegelshohn, Matthias Hebbel, Walter Nisticó, Stefan Czarnetzki, Thorsten Kerkhof, Matthias Meyer, Carsten Rohde, Bastian Schmitz, Michael Wachter, Tobias Wegner, and Christine Zarges. GermanTeam RoboCup 2005, 2005. Only available online: `http://www.germanteam.org/GT2005.pdf`.

[26] 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*, volume 5949 of *Lecture Notes in Artificial Intelligence*, pages 425–436. Springer, 2010.