B-Human

Team Report and Code Release 2013

Thomas Röfer[1], Tim Laue[1], Judith Müller[2], Michel Bartsch[2], Malte Jonas Batram[2], Arne Böckmann[2], Martin Böschen[2], Martin Kroker[2], Florian Maaß[2], Thomas Münder[2], Marcel Steinbeck[2], Andreas Stolpmann[2], Simon Taddiken[2], Alexis Tsogias[2], Felix Wenk[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: December 29, 2013

Contents

Chapter 1

Introduction

1.1 About Us

B-Human is a joint RoboCup team of the Universität Bremen and the German Research Center for Artificial Intelligence (DFKI). The team was founded in 2006 as a team in the Humanoid League, but switched to participating in the Standard Platform League in 2009. Since then, B-Human participated in five RoboCup German Open competitions and in five RoboCups and won all official games it ever played except for the final in RoboCup 2012.

For instance, in 2013 we won the RoboCup German Open with a goal ratio of 50:2 (in five matches). We also participated in the RoboCup World Championship in Eindhoven, Netherlands. With a goal ratio of 62:4 (in eight matches), we regained the title of World Champion. In addition, we also won all technical challenges.

Short descriptions of our last year's work are given in our Team Description Paper [21] as well as in our 2013 RoboCup Winner paper [25].

The current team consists of the following persons:

Members: Michel Bartsch, Malte Jonas Batram, Arne Böckmann, Martin Böschen, Tobias Kastner, Martin Kroker, Florian Maaß, Thomas Münder, Marcel Steinbeck, Andreas Stolpmann, Simon Taddiken, Alexis Tsogias, Felix Wenk.

Leaders: Tim Laue, Judith Müller, Thomas Röfer.

1.2 About the Document

This document provides a survey of this year's code release, continuing the tradition of annual releases that was started several years ago. It is based on last year's code release and describes the evolved system used at RoboCup 2013. The changes made to the system since RoboCup 2012 are shortly enumerated in Section 1.3.

Chapter 2 gives a short introduction on how to build the code including the required software and how to run the NAO with our software. Chapter 3 describes the framework used in our system. Chapter 4 handles the cognition part of the system consisting of an overview of the perception and modeling components. Chapter 5 explains the use of our new behavior description language and gives an overview of the behavior used at RoboCup 2013. Chapter 6 gives a survey of the sensor reading and motion control parts of the system. In Chapter 8.1 the remote control environment SimRobot, the new GameController, and some other tools are described.

Figure 1.1: The majority of the team members

1.3 Changes Since 2012

The major changes made since RoboCup 2012 are described in the following sections:

2.3 Build Chain
The code is now compiled using a custom build chain built around the clang compiler. This enables the use of several new C++11 features.

2.8.4 Color Calibration
Color tables are no longer generated using kd trees. Instead they are calculated from a small set of boundaries.

3.4.4 Generating Streamable Classes
Most of B-Human's representations are now generated by a macro to be streamable automatically.

3.7 Real-Time Logging
The real-time logger now supports the logging of heavily down scaled and compressed thumbnails.

4.1.8 Goal Detection
A new goal detection algorithm has been implemented that handles images which only contain one goal post in a more sophisticated way.

4.1.9 Ball Detection
The ball detection has been improved to deal with the new jerseys.

4.1.10 Visual Obstacle Detection
The robot detection based on the perception of waistbands that are not used anymore was replaced by a significantly better visual obstacle detection.

4.1.11 Field Boundary
A new method of detecting the field boundary has been implemented to deal with the removal of visual boundaries between the fields.

4.2.1 Self-Locator
The self-locator was re-written using a set of Rao-Blackwellized particles.

4.2.3.2 Ultrasonic Obstacle Detection
The ultrasonic obstacle detection was rewritten, explicitly exploiting the overlapping measurement regions of the ultrasound sensors.

5.1 CABSL
A new behavior definition language (CABSL) was used to implement the behavior.

5.3 Behavior of 2013
The behavior was improved to be able to play with five robots on a big field. In addition we added many new moves to our arsenal like stealing the ball from an opponent by quickly dribbling it away, or catching approaching balls on the move.

6.2.7 Arm Motions
Arm motions are now calculated dynamically by the ArmMotionEngine.

6.2.4 Ball Taking
A new ball taking motion engine has been implemented.

6.2.3 Getup Engine
The getup motion is now a dynamic motion calculated by the GetUpEngine.

7 Technical Challenges
We contributed to all three Technical Challenges.

B Custom Kernel
B-Human is now using a custom kernel with hyper-threading support and various improved drivers.

Chapter 2

Getting Started

The goal of this chapter is to give an overview of the code release package and to give instructions on how to enliven a NAO with our code. For the latter, several steps are necessary: downloading the source code, compiling the code using Visual Studio, Xcode, or the Linux shell, setting up the NAO, copying the files to the robot, and starting the software. In addition all calibration procedures are described here.

2.1 Download

The code release can be downloaded from GitHub at `https://github.com/bhuman`. Store the code release to a folder of your liking. After the download is finished, the chosen folder should contain several subdirectories which are described below.

Build is the target directory for generated binaries and for temporary files created during the compilation of the source code. It is initially missing and will be created by the build system.

Config contains configuration files used to configure the B-Human software. A brief overview of the organization of the configuration files can be found in Sect. 2.9.

Install contains all files needed to set up B-Human on a NAO.

Make Contains Makefiles, other files needed to compile the code, the *Copyfiles* tool, and a script to download log files from a NAO. After *generate* has been called this folder will also contain the project files for Xcode and Visual Studio.

Src contains the source code of the B-Human software including the B-Human User Shell (cf. Sect. 8.2).

Util contains auxiliary and third party libraries (cf. Sect. 9) as well as our tools, SimRobot and the GameController (cf. Sect. 8.1).

2.2 Components and Configurations

The B-Human software is usable on Windows, Linux, and OS X. It consists of a shared library for NAOqi running on the real robot, an additional executable for the robot, the same soft-

ware running in our simulator SimRobot (without NAOqi), as well as some libraries and tools. Therefore, the software is separated into the following components:

bush is a tool to deploy and manage multiple robots at the same time (cf. Sect. 8.2).

Controller is a static 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.

copyfiles is a tool for copying compiled code to the robot. For a more detailed explanation see Sect. 2.5. In the Xcode project, this is called *Deploy*.

libbhuman is the shared library used by the B-Human executable to interact with NAOqi.

libgamectrl is a shared NAOqi library that communicates with the GameController. Additionally it implements the official button interface and sets the LEDs as specified in the rules. More information can be found at the end of Sect. 3.1.

libqxt is a static library that provides an additional widget for Qt on Windows and Linux. On OS X, the same source files are simply part of the library *Controller*.

Nao is the B-Human executable for the NAO. It depends on *SpecialActions*, *libbhuman* and *libgamectrl*.

qtpropertybrowser is a static library that implements a property browser in Qt.

SimRobot is the simulator executable for running and controlling the B-Human robot code. It dynamically links against the components *SimRobotCore2*, *SimRobotEditor*, *SimRobotHelp*, *SimulatedNao*, and some third-party libraries. It also depends on the component *SpecialActions*, the results of which is loaded by the robot code. SimRobot is compilable in *Release*, *Develop*, and *Debug* configurations. All these configurations contain debug code, but *Release* performs some optimizations and strips debug symbols (Linux). *Develop* produces debuggable robot code while linking against non-debuggable *Release* libraries.

SimRobotCore2 is a shared library that contains the simulation engine of SimRobot.

SimRobotEditor is a shared library that contains the editor widget of the simulator.

SimRobotHelp is a shared library that contains the help widget of the Simulator.

SimulatedNao is a shared library containing the B-Human code for the simulator. It depends on *Controller*, *qtpropertybrowser* and *libqxt*, and is statically linked against them.

SpecialActions are the motion patterns (*.mof* files) that are compiled into an internal format using the *URC* (cf. Sect. 6.2.2).

URC stands for Universal Resource Compiler and is a small tool for compiling special actions (cf. Sect. 6.2.2).

All components can be built in the three configurations *Release*, *Develop*, and *Debug*. *Release* is meant for "game code" and thus enables the highest optimizations; *Debug* provides full debugging support and no optimization. *Develop* is a special case. It generates executables with some debugging support for the components *Nao* and *SimulatedNao* (see the table below for more specific information). For all other components it is identical to *Release*.

The different configurations for *Nao* and *SimulatedNao* can be looked up here:

	without assertions (NDEBUG)	debug symbols (compiler flags)	debug libs[1] (_DEBUG, compiler flags)	optimizations (compiler flags)
Release				
Nao	✓	×	×	✓
SimulatedNao	✓	×	×	✓
Develop				
Nao	×	×	×	✓
SimulatedNao	×	✓	×	×
Debug				
Nao	×	✓	✓	×
SimulatedNao	×	✓	✓	×

[1] - on Windows - `http://msdn.microsoft.com/en-us/library/0b98s6w8(v=vs.110).aspx`

2.3 Building the Code

2.3.1 Project Generation

The scripts *generate* (or *generate.cmd* on Windows) in the *Make/<OS/IDE>* directories generate the platform or IDE specific files which are needed to compile the components. The script collects all the source files, headers, and other resources if needed and packs them into a solution matching your system (i. e. Visual Studio projects and a solution file for Windows, a CodeLite project for Linux, and a Xcode project for OS X). It has to be called before any IDE can be opened or any build process can be started and it has to be called again whenever files are added or removed from the project. On Linux the *generate* script is only needed when working with CodeLite. Building the code from the command line, via the provided makefile, works without calling *generate* on Linux, as does the Netbeans project (cf. Sect. 2.3.4.2).

2.3.2 Visual Studio on Windows

2.3.2.1 Required Software

- Windows 7 or later
- Visual Studio 2012
- Cygwin ≥ 1.7.9 x86 (x86_64 is not supported!) (available at `http://www.cygwin.com`) with the following additional packages: *rsync*, and *openssh*. Let the installer add an icon to the start menu (the *Cygwin Terminal*) Add the *...\cygwin\bin* directory to the beginning of the PATH environment variable (before the Windows system directory, since there are some commands that have the same names but work differently). Make sure to start the *Cygwin Terminal* at least once, since it will create a home directory.
- alcommon – For the extraction of the required alcommon library and compatible boost headers from the *NAOqi C++ SDK 1.14.5 for Linux 32 bits* (*naoqi-sdk-1.14.5-linux32.tar.gz*) the script *Install/installAlcommon* 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 `https://community.aldebaran-robotics.com` (account required). Please note that this package is only required to compile the code for the actual NAO robot.

2.3.2.2 Compiling

Generate the Visual Studio project files using the script *Make/VS2012/generate.cmd* and open the solution *Make/VS2012/B-Human.sln* in Visual Studio. Visual Studio will then list all the components (cf. Sect. 2.2) of the software in the "Solution Explorer". Select the desired configuration (cf. Sect. 2.2, *Develop* would be a good choice for starters) and build the desired project: *SimRobot* compiles every project used by the simulator, *Nao* compiles every project used for working with a real NAO, and *Utils/bush* compiles the B-Human User Shell (cf. Sect. 8.2). You may also select *SimRobot* or *Utils/bush* as "StartUp Project".

2.3.3 Xcode on OS X

2.3.3.1 Required Software

The following components are required:

- OS X 10.9
- Xcode 5 from the Mac App Store
- Java 7 from `http://www.oracle.com/technetwork/java/javase/downloads/index.html`
- alcommon – For the extraction of the required alcommon library and compatible boost headers from the *NAOqi C++ SDK 1.14.5 for Linux 32 bits* (*naoqi-sdk-1.14.5-linux32.tar.gz*) the script *Install/installAlcommon* 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 `https://community.aldebaran-robotics.com` (account required). Please note that this package is only required to compile the code for the actual NAO robot. Also note that *installAlcommon* expects the extension *.tar.gz*. If the NAOqi archive was partially unpacked after the download, e. g., by Safari, repack it again before executing the script.

2.3.3.2 Compiling

Generate the Xcode project by executing *Make/MacOS/generate*. Open the Xcode project *Make/MacOS/B-Human.xcodeproj*. A number of schemes (selectable in the toolbar) allow building SimRobot in the configurations *Debug*, *Develop*, and *Release*, as well as the code for the NAO[1] in all three configurations (cf. Sect. 2.2). For both targets, *Develop* is a good choice. In addition, the B-Human User Shell *bush* can be built. It is advisable to delete all the schemes that are automatically created by Xcode, i. e. all non-shared ones.

When building for the NAO, a successful build will open a dialog to deploy the code to a robot (using the *copyfiles* script, cf. Sect. 2.5).[2] If the *login* script was used before to login to a NAO,

[1] Note that the cross compiler builds 32 bit code, although the scheme says "My Mac 64-bit".
[2] Before you can do that, you have to setup the NAO first (cf. Sect. 2.4).

the IP address used will be provided as default. In addition, the option `-r` is provided by default that will restart the software on the NAO after it was deployed. Both the IP address selected and the options specified are remembered for the next use of the deploy dialog. The IP address is stored in the file *Config/Scenes/connect.con* that is also written by the *login* script and used by the *RemoteRobot* simulator scene. The options are stored in *Make/MacOS/copyfiles-options.txt*. A special option is `-a`: If it is specified, the deploy dialog is not shown anymore in the future. Instead, the previous settings will be reused, i. e. building the code will automatically deploy it without any questions asked. To get the dialog back, hold down the key Shift at the time the dialog would normally appear.

2.3.4 Linux Shell

The following has been tested and works on Ubuntu 13.04. It should also work on other Linux distributions; however, different or additional packages may be needed.

2.3.4.1 Required Software

Requirements (listed by common package names) for Ubuntu 13.04:

- clang $\geq$ 3.2
- qt4-dev-tools
- libglew-dev
- libxml2-dev
- graphviz – Optional, for generating module graphs and the behavior graph.
- alcommon – For the extraction of the required alcommon library and compatible boost headers from the *NAOqi C++ SDK 1.14.5 for Linux 32 bits* (*naoqi-sdk-1.14.5-linux32.tar.gz*) the script *Install/installAlcommon* 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 `https://community.aldebaran-robotics.com` (account required). Please note that this package is only required to compile the code for the actual NAO robot.

On Ubuntu 13.04 you can execute the following command to install all requirements except for *alcommon*:

```
sudo apt-get install qt4-dev-tools libglew-dev libxml2-dev clang graphviz
```

2.3.4.2 Compiling

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

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

To clean up the whole solution, use:

```
make clean [CONFIG=<configuration>]
```

As an alternative, there is also support for the integrated development environments NetBeans and CodeLite that work similar to Visual Studio for Windows (cf. Sect. 2.3.2.2).

To use CodeLite, execute *Make/LinuxCodeLite/generate* and open the *B-Human.workspace* afterwards. Note that CodeLite 5.x is required to open the workspace generated. Older versions might crash.

To use NetBeans, extract *Make/LinuxNetBeans/B-Human.tar.gz* to *Make/LinuxNetBeans*. It contains a project folder that can be opened using NetBeans.

2.4 Setting Up the NAO

2.4.1 Requirements

First of all, download the current version of the *NAO System Image 1.14.5 for NAO 4 (Intel Atom)* and the *Flash USB Stick 1.12.5* from the download area of `https://community.aldebaran-robotics.com` (account required). In order to flash the robot, you furthermore need a USB flash drive having at least 2 GB space and a network cable.

To use the scripts in the directory *Install*, the following tools are required[3]:

sed, *scp*, *sshpass*.

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

The commands in this chapter are shell commands. They should be executed inside a Unix shell. On Windows, you *must* use the *Cygwin Terminal* to execute the commands. All shell commands should be executed from the *Install* directory.

2.4.2 Creating a Robot Configuration

Before you start to set up the NAO, you need to create configuration files for each robot you want to set up. To create a robot configuration, run *createRobot*. 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 IPv4 address of the robot on both interfaces LAN and WLAN. 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 host name in the NAO operating system and is saved in the chestboard of the NAO as *BodyNickname*.

Before creating your first robot configuration, check whether the network configuration template files *wireless* and *wired* in *Install/Network* and *default* in *Install/Network/Profiles* match the requirements of your local network configuration.

Here is an example for creating a new set of configuration files for a robot named Penny in team three with IP xxx.xxx.3.25:

```
./createRobot -t 3 -r 25 Penny
```

Help for *createRobot* is available using the option *-h*.

[3] In the unlikely case that they are missing in a Linux distribution, execute *sudo apt-get install sed scp sshpass*. On Windows and OS X, they are already installed at this point.

Running *createRobot* creates all needed files to install the robot. This script also creates a robot directory in *Config/Robots*.

2.4.3 Managing Wireless Configurations

All wireless configurations are stored in *Install/Network/Profiles*. Additional configurations must be placed here and will be installed alongside the *default* configuration. After the setup will be completed, the NAO will always load the *default* configuration, when booting the operating system.

You can later switch between different configurations by calling the script *setprofile* on the NAO, which overwrites the *default* configuration.

```
setprofile SPL_A
setprofile Home
```

2.4.4 Setup

After the robot specific configuration files were created (cf. Sect. 2.4.2 and Sect. 2.4.3) plug in your USB flash drive and start the *NAO flasher tool*[4]. Select the *opennao-atom-system-image-1.14.5.opn* and your USB flash drive. Enable "Factory reset" and click on the write button.

After the USB flash drive has been flashed, plug it into the NAO and press the chest button for about 5 seconds. Afterwards the NAO will automatically install NAO OS and reboot. While installing the basic operating system, connect your computer to the robot using the network cable and configure your IP address with 169.254.220.1/24[5]. Once the reboot is finished, the NAO will do its usual Aldebaran wake up procedure.

Unfortunately, copying files to the NAO using *scp* freezes the robot's operating system from time to time. Therefore, it is necessary to use the USB flash drive to handle the installation of the B-Human basic system. For more information have a look at Appendix B.

Unplug the USB flash drive from the NAO. Format this USB flash drive or another one using the FAT32 filesystem. Afterwards mount the USB flash drive and use the *installRobot.usb* script. The script takes the three parameters *name*, *address*, and *path*. *Name* is the name of the robot you have just configured in Section 2.4.2. *Address* is the IP address of the robot you obtain when pressing the chest button of the NAO. *Path* is the root path of your mounted USB flash drive. On Windows, use */cygdrive/<driveletter>* to specify the drive.

For example run:

```
./installRobot.usb Penny 169.254.220.18 /media/usb
```

When running *installRobot.usb*, it will check whether your computer is connected to the robot properly. If it is not, make sure that the network cable is plugged in correctly and restart the NAO by holding the chest button for a few seconds.

Follow the instructions inside the shell. After the installation has finished, the NAO will be restarted automatically. You should now be able to ping the robot using the IP address you configured in Section 2.4.2.

After the first installation of the B-Human basic system using *installRobot.usb*, the robot's operating system should not freeze anymore while copying files via *scp*. Therefore, following

[4] On Linux and OS X you have to start the flasher with root permissions. Usually you can do this with sudo ./flasher

[5] This is not necessary on OS X if using DHCP.

installations[6] can be handled using the *installRobot* script, which works quite similar to the *installRobot.usb* script, but without the necessity of an USB flash drive. **Note:** Installing the B-Human basic system the first time after flashing the robot should always be executed using the *installRobot.usb* script.

For example, to update an already installed NAO, run:

```
./installRobot Penny 169.254.220.18
```

2.5 Copying the Compiled Code

The tool *copyfiles* is used to copy compiled code and configuration files to the NAO. Although *copyfiles* allows specifying the team number, it is usually better to configure the team number and the UDP port used for team communication permanently in the file *Config/settings.cfg*.

On Windows as well as on OS X you can use your IDE to use *copyfiles*. In Visual Studio you can run the script by "building" the tool *copyfiles*, which can be built in all configurations. If the code is not up-to-date in the desired configuration, it will be built. After a successful build, you will be prompted to enter the parameters described below. On the Mac, a successful build for the NAO always ends with a dialog asking for *copyfiles*' command line options.

You can also execute the script at the command prompt, which is the only option for Linux users. The script is located in the folder *Make/<OS/IDE>*.

copyfiles requires two mandatory parameters. First, the configuration the code was compiled with (*Debug*, *Develop* or *Release*)[7], 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*.
-n <number>	Sets team number, replacing the value in the *settings.cfg*.
-r	Restarts *bhuman* and if necessary *naoqi*.
-rr	Restarts *bhuman* and *naoqi*.
-m n <ip>	Copies to IP address <ip> and sets the player number to *n*. This option can be specified more than ones to deploy to multiple robots.
-wc	Compiles also under Windows if the binaries are outdated.
-nc	Never compiles, even if binaries are outdated. Default on Windows/OS X.
-d	Removes all log files from the robot's /home/nao/logs directory before copying files.
-h | --help	Prints the help.

Possible calls could be:

```
./copyfiles Develop 134.102.204.229 -n 5 -t blue -p 3 -r
./copyfiles Release -m 1 10.0.0.1 -m 3 10.0.0.2
```

The destination directory on the robot is */home/nao/Config*. Alternatively the B-Human User Shell (cf. Sect. 8.2) can be used to copy the compiled code to several robots at once.

[6] Re-installation might be useful when updating scripts etc.

[7] This parameter is automatically passed to the script when using IDE-based deployment.

2.6 Working with the NAO

After pressing the chest button, it takes about 40 seconds until NAOqi is started. Currently the B-Human software consists of two shared libraries (*libbhuman.so* and *libgamectrl.so*) that are loaded by NAOqi at startup, and an executable (*bhuman*) also loaded at startup.

To connect to the NAO, the subdirectories of *Make* contain a *login* script for each supported platform. The only parameter of that script is the IP address of the robot to login. It automatically uses the appropriate SSH key to login. In addition, the IP address specified is written to the file *Config/Scenes/connect.con*. Thus a later use of the SimRobot scene *RemoteRobot.ros2* will automatically connect to the same robot. On OS X, the IP address is also the default address for deployment in Xcode.

Additionally the script *Make/Linux/ssh-config* can be used to output a valid ssh *config* file containing all robots currently present in the robots folder. Using this configuration file one can connect to a robot using its name instead of the IP address.

There are several scripts to start and stop NAOqi and *bhuman* via SSH. Those scripts are copied to the NAO upon installing the B-Human software.

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

nao start|stop|restart starts, stops or restarts NAOqi. After updating *libbhuman* with *copyfiles* NAOqi needs a restart. *Copyfiles*' option *-r* accomplishes this automatically.

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. If *copyfiles* is started with option *-r*, it will restart *bhuman*.

status shows the status of NAOqi and *bhuman*.

stop stops running instances of NAOqi and *bhuman*.

halt shuts down the NAO. If NAOqi is running, this can also be done by pressing the chest button longer than three seconds.

reboot reboots the NAO. If NAOqi is running, this can also be done by pressing the chest button longer than three seconds while also pressing a foot bumper.

2.7 Starting SimRobot

On Windows and OS X, SimRobot can either be started from the development environment or by starting a scene description file in *Config/Scenes*[8]. 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/SimRobot/Linux/<configuration>/SimRobot*, either from the shell or from your favorite file browser, and load a scene description file afterwards. When a simulation is opened

[8] On Windows, the first time starting such a file the *SimRobot.exe* must be manually chosen to open these files. Note that both on Windows and OS X, starting a scene description file bears the risk of executing a different version of SimRobot than the one that was just compiled.

for the first time, only the scene graph is displayed. The simulation is already running, which can be noted from the increasing number of simulation steps shown in the status bar. A scene view showing the soccer field can be opened by double-clicking *RoboCup*. The view can be adjusted by using the context menu of the window or the toolbar. Double-clicking *Console* will open a window that shows the output of the robot code and that allows entering commands. All windows can be docked in the main window.

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 main scene *RoboCup*, the objects in the group *RoboCup/robots*, and all other views.

To connect to a real NAO, open the RemoteRobot scene *Config/Scenes/RemoteRobot.ros2*. You will be prompted to enter the NAO's IP address.[9] In a remote connection, the simulation scene is usually empty. Therefore, it is not necessary to open a scene view.

2.8 Calibrating the Robots

Correctly calibrated robots are very important since the software requires all parts of the NAO to be at the expected locations. Otherwise the NAO will not be able to walk stable and projections from image-space to wold-space (and vice versa) will be wrong. In general a lot of calculations will be unreliable. Two physical components of the NAO can be calibrated via SimRobot; the joints (cf. Sect. 2.8.2) and the cameras (cf. Sect. 2.8.3). Checking those calibrations from time to time is important, especially for the joints. New robots come with calibrated joints and are theoretically ready to play out of the box. However, over time and usage, the joints wear out. This is especially noticeable with the hip joint.

In addition to that the B-Human software uses seven color classes (cf. Sect. 4.1.4) which have to be calibrated, too (cf. Sect. 2.8.4). Changing locations or light conditions might require them to be adjusted.

2.8.1 Overall physical calibration

The physical calibration process can be split into three steps with the overall goal of an upright and straight standing robot, and a correctly calibrated camera. The first step is to get both feet in a planar position. This does not mean that the robot has to stand straight. It is done by lifting the robot up so that the bottom of the feet can be seen. The joint offsets of feet and legs are then changed until both feet are planar and the legs are parallel to one another. The distance between the two legs can be measured at the gray parts of the legs. They should be 10 cm apart from center to center.

The second step is the camera calibration (cf. Sect. 2.8.3). This step also measures the tilt of the body with respect to the feet. This measurement can then be used in the third step to improve the joint calibration and straighten the robot up (cf. Sect. 2.8.2). In some cases, it may be necessary to repeat these steps.

[9] The script might instead automatically connect to the IP address that was last used for login or deployment.

2.8.2 Joint Calibration

The software supports two methods for calibrating the joints; either by manually adjusting offsets for each joint, or by using the JointCalibrator module which uses an inverse kinematic to do the same (cf. Sect. 6.2.1.2). The third step of the overall calibration process (cf. Sect. 2.8.1) can only be done via the JointCalibrator. When switching between those two methods, it is necessary to save the *JointCalibration*, redeploy the NAO and restart bhuman. Otherwise the changes done previously will not be used.

Before changing joint offsets, the robot has to be set in a standing position with fixed joint angles. Otherwise the balancing mechanism of the motion engine might move the legs, messing up the joint calibrations. This can be done with

```
get representation:MotionRequest
```

and then set *motion* = *stand* in the returned statement. Followed by a

```
get representation:JointRequest
```

and a set of the returned *JointRequest*.

When the calibration is finished it should be saved

```
save representation:JointCalibration
```

and

```
set representation:JointRequest unchanged
```

should be used so the head can be moved when calibrating the camera.

Manually Adjusting Joint Offsets

There are two ways to adjust the joint offsets. Either by requesting the *JointCalibration* representation with a *get* call:

```
get representation:JointCalibration
```

modifying the calibration returned and then setting it. Or by using a Data View (cf. Sect. 8.1.4.5)

```
vd representation:JointCalibration
```

which is more comfortable. The units of the offsets are in degrees.

The *JointCalibration* also contains other information for each joint that should not be changed!

Using the JointCalibrator

First set the JointCalibrator to provide the*JointCalibration*:

```
mr JointCalibration JointCalibrator
```

When a completely new calibration is wanted, the *JointCalibration* can be reset:

```
dr module:JointCalibrator:reset
```

Afterwards the translation and rotation of the feet can be modified. Again either with

```
get module:JointCalibrator:offsets
```

or with:

```
vd module:JointCalibrator:offsets
```

The units of the translations are in millimeters and the rotations are in radian.

Straightening Up the NAO

Note: after the camera calibration it is not necessary to set the *JointRequest* for this step.

The camera calibration also calculates a rotation for the body tilt and role. Those values can be passed to the JointCalibrator that will then set the NAO in an upright position. Call:

```
get representation:CameraCalibration
mr JointCalibration JointCalibrator
get module:JointCalibrator:offsets
```

Copy the value of *bodyTiltCorrection* (representation *CameraCalibration*) into *bodyRotation.y* (representation *JointCalibration*) and *bodyRollCorrection* (representation *CameraCalibration*) into *bodyRotation.x* (representation *JointCalibration*). Afterwards, set *bodyTiltCorrection* and *bodyRollCorrection* (representation *CameraCalibration*) to zero.

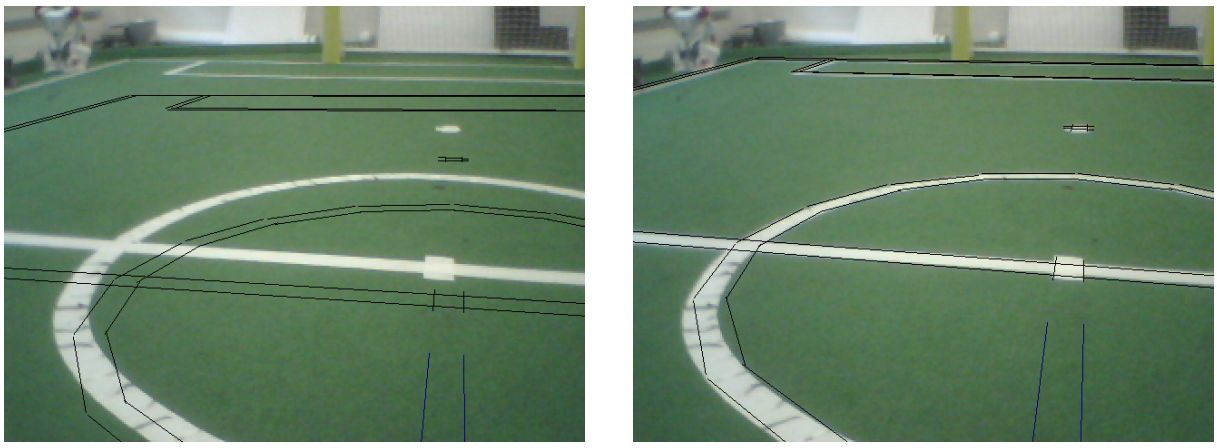
The last step is to adjust the translation of both feet at the same time (and most times in the same direction) so they are perpendicular positioned below the torso. A plummet or line laser is very useful for that task.

When all is done save the representations by executing

```
save representation:JointCalibration
save representation:CameraCalibration
```

Then, redeploy the NAO, and restart bhuman.

2.8.3 Camera Calibration



(a) (b)

Figure 2.1: Projected lines before (a) and after (b) the calibration procedure

For calibrating the cameras (cf. Sect. 4.1.2.1) using the module CameraCalibratorV4, follow the steps below:

1. Connect the simulator to a robot on the field and place it on a defined spot (e. g. the penalty mark).

2. Run the SimRobot configuration file *CameraCalibratorV4.con* (in the console type *call CameraCalibratorV4*). This will initialize the calibration process and furthermore print commands to the simulator console that will be needed later on.

3. Announce the robot's position on the field (cf. Sect. 4.1.2) using the `CameraCalibratorV4` module (e. g. for setting the robot's position to the penalty mark of a field, type *set module:CameraCalibratorV4:robotPose rotation = 0; translation = {x = -2700; y = 0;};* in the console).

4. Start collecting points using *dr module:cameraCalibratorV4:collectPoints*. Move the head to collect points for different rotations of the head by clicking on field lines. The number of collected points is shown in the top area of the image view. Please note that the `CameraCalibratorV4` module must be notified to which camera the collected points belong. The current camera of the `CameraCalibratorV4` can be switched by typing *set module:CameraCalibratorV4:currentCamera upper* respectively *set module:-CameraCalibratorV4:currentCamera lower*. It must always match the image view clicked in. The head can be moved via the Representation *HeadAngleRequest* (type *vd representation:HeadAngleRequest*).

5. Run the automatic calibration process using *dr module:CameraCalibratorV4:optimize* and wait until the optimization has converged.

The calibration module allows to arbitrarily switch between upper and lower camera during the point collection phase as mentioned above. Both cameras should be considered for a good result. For the purpose of manual refinement of the robot-specific parameters mentioned, 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). Modify the parameters of *CameraCalibration* so that the projected lines match the field lines in the image (see Fig. 2.1 for a desirable calibration).

2.8.4 Color Calibration

Calibrating the color classes is split into two steps. First of all the parameters of the camera driver must be updated to the environment's needs. The command:

```
get representation:CameraSettings
```

will return the current settings. Furthermore the necessary set command will be generated. The most important parameters are:

whiteBalance: The white balance used. The available interval is [2700, 6500].

exposure: The exposure used. The available interval is [0, 512]. Usually the exposure is 30. High exposure leads to blurred images.

gain: The gain used. The available interval is [0, 255]. Usually the gain is 50 to 70. High gain leads to noisy images.

autoWhiteBalance: Enable(1)/disable(0) the automatic for white balance. This parameter should always be disabled.

autoExposure: Enable(1)/disable(0) the automatic for exposure. This parameter should always be disabled.

Figure 2.2: The left figure shows an image with improper white balance. The right figure shows the same image with better settings for white balance.

After setting up the parameters of the camera driver, the parameters of the color classes must be updated (cf. Sect. 4.1.4). To do so, one needs to open the Calibration View (cf. Sect. 8.1.4.1 and Sect. 8.1.4.1). It might be useful to first update the green calibration since some perceptors, e. g. the BallPerceptor, use the *FieldBoundary*. The save button of the calibration view should be used from time to time to update the calibration on the robot and get direct feedback of the perceptors. After finishing the color class calibration and saving the current parameters, copyfiles/bush (cf. Sect. 2.5) can be used to deploy the current settings. Ensure the updated files *cameraSettings.cfg* and *colorProvider.cfg* are stored in the correct location.

2.9 Configuration Files

Since the recompilation of the code takes a lot of time in some cases and each robot needs a different configuration, the software uses a huge amount of configuration files which can be altered without causing recompilation. All the files that are used by the software[10] are located below the directory *Config*.

Besides the global configuration files there are specific files for each robot. These files are located in *Config/Robots/<robotName>* where *<robotName>* is the name of a specific robot. They are only taken into account if the name of the directory matches the name of the robot where the code is executed on. In the Simulator, the robot name is always "Nao". On a NAO, it is the name stored in the chestboard.

Locations can be used to configure the software for different independent tasks. They can be set up by simply creating a new folder with the desired name within *Config/Locations* and placing configuration files in it. Those configuration files are only taken into account if the location is activated in the file *Config/settings.cfg*.

[10] There are also some configuration files for the operating system of the robots that are located in the directory *Install*.

To handle all these different configuration files, there are fall-back rules that are applied if a requested configuration file is not found. The search sequence for a configuration file is:

1. *Config/Robots/<robot name>/<filename>*
2. *Config/Robots/Default/<filename>*
3. *Config/Locations/<current location>/<filename>*
4. *Config/Locations/Default/<filename>*
5. *Config/<filename>*

So, whether a configuration file is robot-dependent or location-dependent or should always be available to the software is just a matter of moving it between the directories specified above. This allows for a maximum of flexibility. Directories that are searched earlier might contain specialized versions of configuration files. Directories that are searched later can provide fallback versions of these configuration files that are used if no specialization exists.

Using configuration files within our software requires very little effort, because loading them is completely transparent for a developer when using parametrized modules (cf. Sect. 3.3.5).

Chapter 3

Architecture

The B-Human architecture [24] is based on the framework of the GermanTeam 2007 [23], 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.

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.

- 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.

The new GameController (cf. Sect. 8.3) provides the library *libgamectrl* that handles the network packets, sets the LEDs, and handles the official button interface. The library is already integrated into the B-Human project. Since the *libgamectrl* is a NAOqi module, the *libbhuman* (cf. Sect. 3.1) handles the data exchange with the library and provides the resulting game control data packet to the main B-Human executable. The *libbhuman* also sets the team number, team color, and player number whenever a new instance of the main B-Human executable is started, so that the *libgamectrl* resets the game state to *Initial*.

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 from each camera at a frequency of 30 Hz and accepts new joint angles at 100 Hz. For handling the camera images, there would actually have been two options: either to have two processes each of which processes the images of one of the two cameras and a third one that collects the results of the image processing and executes world modeling and behavior control, or to have a single process that alternately processes the images of both cameras and also performs all further steps. We use the latter approach, because each interprocess communication might add delays to the system. Since the images of both cameras are processed, that single process runs at 60 Hz. In addition, there is a process that runs at the motion frame rate of the NAO, i. e. at 100 Hz. Another process performs the TCP communication with a host PC for the purpose of debugging.

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

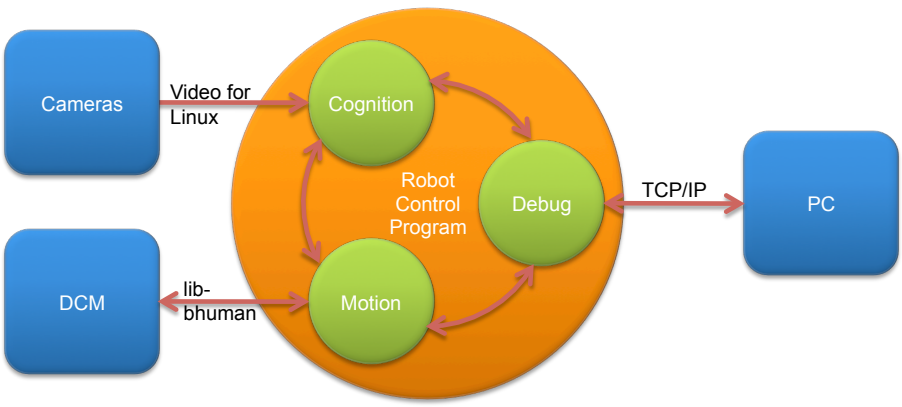


Figure 3.1: The processes used on the NAO

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 past team in the Humanoid League, framework processes were mapped to actual processes of the operating system (i. e. Windows CE). For the sake of consistency, we will use the term "processes" in this document.

3.3 Modules and Representations

A robot control program usually consists of several modules, each performing a certain task, e. g. image processing, self-localization, or walking. Modules require a certain input and produce a certain output (so-called *representations*). Therefore, they have to be executed in a specific order to make the whole system work. The module framework introduced in [23] simplifies the definition of the interfaces of modules and automatically determines the sequence in which the modules must be executed. It consists of the *blackboard*, the *module definition*, and a visualization component (cf. Sect. 8.1.4.5).

3.3.1 Blackboard

The blackboard [9] is the central storage for information, i. e. for the representations. Each process is associated with its own instance of the 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 contains references to all representations; however, only the representations used by the associated process are actually instantiated. For instance, the process *Motion* does not process camera images. Therefore, it does not require to instantiate an image object (approximately 1.2 MB 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 instantiating the module. Here an example:

```
MODULE(SimpleBallLocator)
  REQUIRES(BallPercept)
  REQUIRES(FrameInfo)
  PROVIDES(BallModel)
  DEFINES_PARAMETER(float, positionOffset, 1.1f)
END_MODULE

class SimpleBallLocator : public SimpleBallLocatorBase
{
  void update(BallModel& ballModel)
  {
```

```
    if(theBallPercept.wasSeen)
    {
      ballModel.position = theBallPercept.position * positionOffset;
      ballModel.wasLastSeen = theFrameInfo.frameTime;
    }
  }
}

MAKE_MODULE(SimpleBallLocator, 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. 8.1.4.5). This second parameter is also used to filter modules that can be loaded in the current framework environment, i. e. the process (cf. Sect. 3.2). In process *Cognition* the categories `Cognition Infrastructure`, `Perception`, `Modeling`, and `Behavior Control` are available and in process *Motion* the categories `Motion Infrastructure`, `Motion Control`, and `Sensing`. The list of available categories is defined in the main implementation file of the respective process (*Src/Processes/Cognition.cpp* and *Src/Processes/Motion.cpp*). While the module interface is usually part of the header file, the `MAKE_MODULE` statement has to be part of the implementation file.

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 and for visualization (cf. Sect. 8.1.4.5). Last but not least, the execution time of each module can be determined (cf. Sect. 3.6.7) and the representations provided can be sent to a host PC or even altered by it.

Some of the functionality mentioned above is not provided directly by the `PROVIDES` macro but by one of its variants:

`PROVIDES_WITH_MODIFY`**:** The representation is sent to a host PC upon request and can be altered by it. The representation has to be streamable for this to work (cf. Sect. 3.4.3).

`PROVIDES_WITH_OUTPUT`**:** The representation is continuously being sent to a host PC. This enables host side logging of the representation. The representation has to be streamable for this to work (cf. Sect. 3.4.3) and a corresponding message id must exist (cf. Sect. 3.5.1).

`PROVIDES_WITH_DRAW`**:** A parameterless *draw* method (implemented by the representation itself) is called every frame. This method can be used to visualize the representation.

All combinations of the variants mentioned above are also available (e. g. `PROVIDES_WITH_MODIFY_AND_OUTPUT_AND_DRAW` or `PROVIDES_WITH_MODIFY_AND_DRAW`).

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 provides it or that it is not provided at all. Normally, the configuration is read from the file *Config/Location/<location>/modules.cfg* during the start-up of the process, but it can also be changed interactively when the robot has a debug connection to a host PC using the command `mr` (cf. Sect. 8.1.6.3).

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.

3.3.5 Parameterizing Modules

Modules usually need some parameters to function properly. Those parameters can also be defined in the module's interface description. Parameters behave like protected class members and can be accessed in the same way. Additionally, they can be manipulated from the console using the commands *get parameters:<ModuleName>* or *vd parameters:<ModuleName>* (cf. Sect. 8.1.6.3).

There are two different parameter initialization methods. In the hard-coded approach, the initialzation values are specified as part of the C++ source file. They are defined using the `DEFINES_PARAMETER(<data type>, <name> , <default value>)` macro. This macro is intended for parameters that may change during development but will never change again afterwards.

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

```
  DEFINES_PARAMETER(int, paramA, 4)
  DEFINES_PARAMETER(Vector2<>, paramB, Vector2<>(65.f, 0.f))
END_MODULE
```

In contrast, loadable parameters are initialized to values that are loaded from a configuration file upon module creation. They are defined using the `LOADS_PARAMETER(<data type>, <name>)` macro. It is not possible to mix both types of initialization. A module has to either load all parameters or none at all.

```
MODULE(SimpleBallLocator)
  REQUIRES(BallPercept)
  PROVIDES(BallModel)
  LOADS_PARAMETER(int, paramA)
  LOADS_PARAMETER(Vector2<>, paramB)
END_MODULE
```

By default, parameters are loaded from a file with the same base name as the module, but starting with a lowercase letter[2] and the extension *.cfg*. For instance if a module is named SimpleBallLocator, its configuration file is *simpleBallLocator.cfg*. This file can be placed anywhere in the usual configuration file search path (cf. Sect. 2.9). It is also possible to assign a custom name to a module's configuration file by passing the name as a parameter to the constructor of the module's base class.

Parameters may have any data type as long as it is streamable (cf. Sect. 3.4.3).

3.4 Serialization

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, e. g. lists, trees, or graphs. Our implementation for streaming data follows the ideas introduced by the C++ iostreams library, i. e., the operators << and >> are used to implement the process of serialization. In contrast to the iostreams library, our implementation guarantees that data is streamed in a way that it can be read back without any special handling, even when streaming into and from text files, i. e. the user of a stream does not need to know about the representation that is used for the serialized data (cf. Sect. 3.4.1).

On top of the basic streaming class hierarchy, 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. This information is used to translate between the data in binary form and a human-readable format that reflects the hierarchy of a data structure, its variables, and their actual values.

As a third layer of serialization, two macros allow defining classes that automatically implement the method `serialize(In*, Out*)`.

3.4.1 Streams

The foundation of B-Human's implementation of serialization is a hierarchy of streams. As a convention, all classes that write data into a stream have a name starting with "Out", while

[2] Actually, if a module name begins with more than one uppercase letter, all initial uppercase letters but the last one are transformed to lowercase, e. g. the module `USControl` reads the file *usControl.cfg*.

classes that read data from a stream start with "In". In fact, all writing classes are derived from the class `Out`, and all reading classes are derivations of the class `In`. All classes support reading or writing basic datatypes with the exceptions of `long`, `unsigned long`, and `size_t`, because their binary representations have different sizes on currently used platforms (32/64 bits). They also provide the ability to `read` or `write` raw binary data.

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.

A special case are the `OutMap` and the `InMap` streams. They only work together with classes that are derived from the class `Streamable`, because they use the structural information that is gathered in the `serialize` method. They are both directly derived from `Out` and `In`, respectively.

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
- **OutMap:** Writing into a stream in configuration map format (cf. Sect. 3.4.5). This only works together with serialization (cf. Sect. 3.4.3), i. e. a streamable object has to be written. This class cannot be used directly.

OutMapFile: Writing into a file in configuration map format
OutMapMemory: Writing into a memory area in configuration map format
OutMapSize: Determine memory size for configuration map format storage

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

InMap: Reading from a stream in configuration map format (cf. Sect. 3.4.5). This only works together with serialization (cf. Sect. 3.4.3), i. e. a streamable object has to be read. This class cannot be used directly.
InMapFile: Reading from a file in configuration map format
InMapMemory: Reading from a memory area in configuration map format

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 3.14 "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:

```
13.14Hello 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 Streamable Classes

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 implement the *modify* functionality (cf. Sect. 3.6.6), 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> [, <class>]) streams an attribute, retrieving its name in the process. The second parameter is optional. If the streamed attribute is of an enumeration type (single value, array, or vector) and that enumeration type is not defined in the current class, the second parameter specifies the name of the class in which the enumeration type is defined. The enumeration type streamed must either be defined with the `ENUM` macro (cf. Sect. 3.4.6) or a matching `getName` function must exist.

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`, an attribute `testEnumVector` which is a vector of values of an enumeration type that is defined in this class, and an enumeration variable of a type which was defined in `SomeOtherClass`, the following code can be used:

```
virtual void serialize(In* in, Out* out)
{
  STREAM_REGISTER_BEGIN;
  STREAM(test);
  STREAM(testEnumVector);
  STREAM(otherEnum, SomeOtherClass);
  STREAM_REGISTER_FINISH;
}
```

In addition to the above listed macros `STREAM_*()` there is another category of macros of the form `STREAM_*_EXT()`. In contrast to the macros described above, they are not intended to be used within the `serialize`-method, but to define the external streaming operators `operator<<(...)` and `operator>>(...)`. For this purpose, they take the actual stream to be read from or written to as an additional (generally the first) parameter. The advantage of using external streaming operators is that the class to be streamed does not need to implement a virtual method and thus can save the space needed for a virtual method table, which is especially reasonable for very small classes. Consider an example of usage as follows:

```
template<typename T> Out& operator<<(Out& out, const Vector2<T>& vector)
{
  STREAM_REGISTER_BEGIN_EXT(vector);
  STREAM_EXT(out, vector.x);
  STREAM_EXT(out, vector.y);
  STREAM_REGISTER_FINISH;
  return out;
}
```

3.4.4 Generating Streamable Classes

The approach to make classes streamable described in the previous section has been proven tedious and prone to mistakes in recent years. Each new member variable has to be added in two or even three places, i. e. it must be declared, it must be streamed, and often it also must be initialized. During recent code inspections it became obvious that several variables had been introduced without adding them to the `serialize` method. This resulted for instance in configuration parameters that were never set or unexpected behavior when replaying a log file, because some important data was simply missing. This led to the introduction of two new macros that generate a streamable class and optionally also initialize its member variables. The first is:

```
STREAMABLE(<class>,
{ <header>,
  <comma-separated-declarations>,
  <default-constructor-body>
})
```

The second is very similar:

```
STREAMABLE_WITH_BASE(<class>, <base>, ...
```

The parameters have the following meaning:

class: The name of the class to be declared.

base: Its base class. It must be streamable and its `serialize` method must not be private. The default (without `_WITH_BASE`) is the class `Streamable`.

header: Everything that can be part of a class body except for the attributes that should be streamable and the default constructor. Please note that this part must not contain commas that are not surrounded by parentheses, because C++ would consider it to be more than a single macro parameter otherwise.

comma-separated-declarations: Declarations of the streamable attributes in four possible forms:

```
(<type>) <var>
(<type>)(<init>) <var>
(<enum-domain>, <type>) <var>
(<enum-domain>, <type>)(init) <var>
```

type: The type of the attribute that is declared.

var: The name of the attribute that is declared.

init: The initial value of the attribute, or in case an object is declared, the parameter(s) passed to its constructor.

enum-domain: If an enum is declared, the type of which is not declared in the current class, the class the enum is declared in must be specified here. The `type` and the optional `init` value are automatically prefixed by this entry with `::` in between. Please note that there is a single case that is not supported, i. e. streaming a *vector* of enums that are declared in another class, because in that case, the class name is not a prefix of the typename rather than a prefix of its type parameter. The macro would generate `C::std::vector<E>` instead of `std::vector<C::E>`, which does not compile.

default-constructor-body: The body of the default constructor. Can be empty.

Please note that all these parts, including each declaration of a streamable attribute, are separated by commas, since they are parameters of a macro. Here is an example:

```
STREAMABLE(Example,
{
public:
  ENUM(ABC, a, b, c),

  (int) anInt,
  (float)(3.14f) pi,
  (int[4]) array,
```

```
    (Vector2<>)(1.f, 2.f) aVector,
    (ABC) aLetter,
    (MotionRequest, Motion)(stand) motionId,

    memset(array, 0, sizeof(array));
  });
```

In this example, all attributes except for `anInt` and `aLetter` would be initialized when an instance of the class is created.

The macros can not be used if the `serialize` method should do more than just streaming the member variables. For instance if some of the member variables are not streamed but computed from the values of other member variables instead, additional code is needed to execute these computations whenever new data is read.

3.4.5 Configuration Maps

Configuration maps introduce the ability to handle serialized data from files in a random order. The sequence of entries in the file does not have to match the order of the attributes in the C++ data structure that is filled with them. In contrast to most streams presented in Sect. 3.4.1, configuration maps do not contain a serialization of a data structure, but rather a hierarchical representation.

Since configuration maps can be read from and be written to files, there is a special syntax for such files. A file consists of an arbitrary number of pairs of keys and values, separated by an equality sign, and completed by a semicolon. Values can be lists (encased by square brackets), complex values (encased by curly brackets) or plain values. If a plain value does not contain any whitespaces, periods, semicolons, commas, or equality signs, it can be written without quotation marks, otherwise it has to be encased in double quotes. Configuration map files have to follow this grammar:

```
map     ::= record
record  ::= field ';' { field ';' }
field   ::= literal '=' ( literal | '{' record '}' | array )
array   ::= '[' [ element { ',' element } [ ',' ] ']'
element ::= literal | '{' record '}'
literal ::= '"' { anychar1 } '"' | { anychar2 }
```

`anychar1` must escape doublequotes and the backslash with a backslash. `anychar2` cannot contain whitespace and other characters used by the grammar. However, when such a configuration map is read, each `literal` must be a valid literal for the datatype of the variable it is read into. As in C++, comments can be denoted either by `//` for a single line or by `/* ... */` for multiple lines. Here is an example:

```
// A record
defensiveGoaliePose = {
  rotation = 0;
  translation = {x = -4300; y = 0;};
};
/* An array of
 * three records
 */
kickoffTargets = [
  {x = 2100; y = 0;},
  {x = 1500; y = -1350;},
  {x = 1500; y = 1350;}
];
```

```
// Some individual values
outOfCenterCircleTolerance = 150.0;
ballCounterThreshold = 10;
```

Configuration maps can only be read or written through the streams derived from `OutMap` and `InMap`. Accordingly, they require an object of a streamable class to either parse the data in map format or to produce it. Here is an example of code that reads the file shown above:

```
STREAMABLE(KickOffInfo,
{,
  (Pose2D) defensiveGoaliePose,
  (std::vector<Vector2<>>) kickoffTargets,
  (float) outOfCenterCircleTolerance,
  (int) ballCounterThreshold,
});

InMapFile stream("kickOffInfo.cfg");
KickOffInfo info;
if(stream.exists())
  stream >> info;
```

3.4.6 Enumerations

To support streaming, enumeration types should be defined using the macro `ENUM` defined in *Tools/Enum.h* rather than using the C++ `enum` keyword directly. The macro's first parameter is the name of the enumeration type and all further parameters are the elements of the defined enumeration type. It is not allowed to assign specific integer values to the elements of the enumeration type, with one exception: It is allowed to initialize an element with the symbolic value of the element that has immediately been defined before (see example below). The macro automatically defines a function `static inline const char* getName(Typename)`, which can return the string representations of all "real" enumeration elements, i. e. all elements that are not just synonyms of other elements. In addition, the function will return 0 for all values outside the range of the enumeration type.

The macro also automatically defines a constant `numOf<Typename>s` which reflects the number of elements in the enumeration type. Since the name of that constant has an added "s" at the end, enumeration type names should be singular. If the enumeration type name already ends with an "s", it might be a good idea to define a constant outside the enumeration type that can be used instead, e. g. `enum {numOfClasses = numOfClasss}` for an enumeration type with the name `Class`.

The following example defines an enumeration type `Letter` with the "real" enumeration elements `a`, `b`, `c`, and `d`, a user-defined helper constant `numOfLettersBeforeC`, and an automatically defined helper constant `numOfLetters`. The numerical values of these elements are `a = 0, b = 1, c = 2, d = 3, numOfLettersBeforeC = 2, numOfLetters = 4`. In addition, the function `getName(Letter)` is defined that can return "a", "b", "c", "d", and 0.

```
ENUM(Letter,
  a,
  b,
  numOfLettersBeforeC,
  c = numOfLettersBeforeC,
  d
);
```

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 are streamable (cf. Sect. 3.4), 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. When changing this enumeration by adding, removing, or re-sorting message types, compatibility issues with existing log files or team mates running an older version of the software are highly probable.

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 is 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* (so-called shared representations) are automatically calculated by the ModuleManager based on the representations required by modules loaded in the respective process but not provided by modules in the same process. The directions in which they are sent are also automatically determined by the ModuleManager.

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());
```

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 or Ethernet 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 8.1.6.3.

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 (1384 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. There also exist a macro `TEAM_OUTPUT_FAST(<id>, <format>, <sequence>)` that uses the information from the blackboard entry `theTeamMateData` to even suppress the actual streaming to the message queue in all frames in which the queue will not be sent anyway. This saves computation time. However consequently, at the location where the macro is used, `theTeamMateData` must be accessible.

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. Since the software runs fast enough and the debug mechanisms greatly support developers when working with a NAO, these mechanisms are available in all project configurations. To disable code used for debugging as originally intended, including not to start the process *Debug* (cf. Sect. 3.2), the preprocessor variable `RELEASE` must be defined.

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. However, at the time being, these parts of the source code are also enabled in *Release* configuration as mentioned above.

The debug requests can be used to trigger certain debug messages to be sent as well as to switch on certain parts of algorithms. They can be sent using the SimRobot software when connected to a NAO (cf. command *dr* in Sect. 8.1.6.3). The following 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_ONCE(<id>, <statements>) executes the statements *once* when 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 would also be executed if the symbol `RELEASE` were defined.

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 requests 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 each debug response 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. 8.1.6.3).

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. 8.1.4.1) or in a color space view (cf. Sect. 8.1.4.1). 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>, <width>, <height>) initializes the debug image as black and sets its size to (width, height).

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_RGB(<id>, <xx>, <yy>, <r>, <g>, <b>) converts the RGB color and 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.

COMPLEX_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 additionally in a separate method), only a single test is required when the debug image is not requested.

SET_DEBUG_IMAGE_SIZE(<width>, <height>) sets the debug image's size. The default size is 640×480.

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

```
class Test
{
private:
  DECLARE_DEBUG_IMAGE(test);

public:
  void draw(const Image& image)
  {
    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 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 type "drawingOnImage" can be displayed in an image view (cf. Sect. 8.1.4.1) and all drawings of type "drawingOnField" can be rendered into a field view (cf. Sect. 8.1.4.1).

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> [, <code>]) declares a debug drawing with the specified *id* and *type*. Optionally, code can be specified that is always exe-

cuted while the creation of the drawing is requested. This part is similar to the macro `COMPLEX_DRAWING` described below. 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`, but it can also provide the same functionality as `COMPLEX_DRAWING` anyway.

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 3-D Debug Drawings

In addition to the aforementioned two-dimensional debug drawings, there is a second set of macros in *Src/Tools/Debugging/DebugDrawings3D.h* which provide the ability to create three-dimensional debug drawings.

3-D debug drawings can be declared with the macro `DECLARE_DEBUG_DRAWING3D(<id>, <type> [, <code>])`. The `id` can then be used to add three dimensional shapes to this

drawing. `type` defines the coordinate system in which the drawing is displayed. It can be set to "field", "robot", or any named part of the robot model in the scene description. Note that drawings directly attached to hinges will be drawn relative to the base of the hinge, not relative to the moving part. Drawings of the type "field" are drawn relative to the center of the field, whereas drawings of the type "robot" are drawn relative to the origin of the robot according to Aldebaran's documentation. B-Human uses a different position in the robot as origin, i. e. the middle between the two hip joints. An object called "origin" has been added to the NAO simulation model at that position. It is often used as reference frame for 3-D debug drawings in the current code. The optional parameter allows defining code that is executed while the drawing is requested.

The parameters of macros adding shapes to a 3-D debug drawing start with the `id` of the drawing this shape will be added to, followed, e. g., by the coordinates defining a set of reference points (such as corners of a rectangle), and finally the drawing color. Some shapes also have other parameters such as the thickness of a line. Here are a few examples for shapes that can be used in 3-D debug drawings:

LINE3D(<id>, <fromX>, <fromY>, <fromZ>, <toX>, <toY>, <toZ>, <size>, <color>) draws a line between the given points.

QUAD3D(<id>, <corner1>, <corner2>, <corner3>, <corner4>, <color>) draws a quadrangle with its four corner points given as 3-D vectors and specified color.

SPHERE3D(<id>, <x>, <y>, <z>, <radius>, <color>) draws a sphere with specified radius and color at the coordinates (x, y, z).

COORDINATES3D(<id>, <length>, <width>) draws the axis of the coordinate system with specified length and width into positive direction.

The header file furthermore defines some macros to scale, rotate, and translate an entire 3-D debug drawing:

SCALE3D(<id>, <x>, <y>, <z>) scales all drawing elements by given factors for x, y, and z axis.

ROTATE3D(<id>, <x>, <y>, <z>) rotates the drawing counterclockwise around the three axes by given radians.

TRANSLATE3D(<id>, <x>, <y>, <z>) translates the drawing according to the given coordinates.

An example for 3-D debug drawings (analogously to the example for regular 2-D debug drawings):

```
DECLARE_DEBUG_DRAWING3D("test3D", "field");
SPHERE3D("test3D", 0, 0, 250, 75, ColorClasses::blue);
```

This example initializes a 3-D debug drawing called `test3D` which draws a blue sphere. Because the drawing is of type `field` and the origin of the field coordinate system is located in the center of the field, the sphere's center will appear 250 mm above the center point.

Watch the result in the scene view of SimRobot (cf. Sect. 8.1.3) by sending a debug request (cf. Sect. 8.1.6.3) for a 3-D debug drawing with the ID of the desired drawing prefixed by "debugDrawing3d:" as its only parameter. If you wanted to see the example described above,

you would type "dr debugDrawing3d:test3D" into the SimRobot console. Note that you might have to activate 3-D debug drawings in SimRobot first. You can do this by simply right-clicking on the scene view and selecting the desired type of rendering in the "Drawings Rendering" submenu.

3.6.5 Plots

The macro `PLOT(<id>, <number>)` allows plotting data over time. The plot view (cf. Sect. 8.1.4.5) 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. 8.1.6.3).

For example, the following code statement plots the measurements of the gyro for the pitch axis in degrees. It should be placed in a part of the code that is executed regularly, e. g. inside the update method of a module.

```
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.

3.6.6 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. 8.1.6.3). 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);
SelfLocatorParameters p;
MODIFY("parameters:SelfLocator", 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`.

If a single variable of an enumeration type should be modified, another macro called `MODIFY_ENUM` has to be used. It has an optional third parameter to which the name of the class must be passed in which the enumeration type is defined. It can be omitted if the enumeration type is defined in the current class. This is similar to the `STREAM` macro (cf. Sect. 3.4.3).

```
class Foo
{
public:
  ENUM(Bar, a, b, c);
  void f()
  {
    Bar x = a;
    MODIFY_ENUM("x", x);
  }
```

```
};

class Other
{
  void f()
  {
    Foo::Bar y = Foo::a;
    MODIFY_ENUM("y", y, Foo);
  }
};
```

3.6.7 Stopwatches

Stopwatches allow the measurement of the execution time of parts of the code. The statements, the runtime of which shall be measured, have to be placed inside 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 the time measurement of all stopwatches, the debug request `dr timing` has to be sent. The measured times can be seen in the timing view (cf. Sect. 8.1.4.5). By default, a stopwatch is already defined for each representation that is currently provided.

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

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

Often rudimentary statistics (cf. Sect. 8.1.4.5) of the execution time of a certain part of the code are not sufficient, but the actual progress of the runtime is needed. For this purpose there is another macro `STOP_TIME_ON_REQUEST_WITH_PLOT(<id>, <statements>)` that enables measuring *and* plotting the execution time of some code. This macro is used exactly as the one without `_WITH_PLOT`, but it additionally creates a plot `stopwatch:myCode` (assuming the example above) (cf. Sect. 3.6.5 for the usage of plots).

3.7 Logging

The B-Human framework offers a sophisticated logging functionality that can be used to log the values of selected representations while the robot is playing. There are two different ways of logging data:

3.7.1 Online Logging

The online logging feature can be used to log data directly on the robot during regular games. It is implemented as part of the Cognition process and is designed to log representations in real-time. Representations that are not accessible from the Cognition process cannot be logged online.

Online logging starts as soon as the robot enters the *ready* state and stops upon entering the *finished* state. Due to the limited space on the NAO's flash drive, the log files are compressed on the fly using Google's snappy compression [4].

To retain the real-time properties of the Cognition process the heavy lifting, i. e. compression and writing of the file, is done in a separate thread without real-time scheduling. This thread uses every bit of remaining processor time that is not used by one of the real-time parts of the system. Communication with this thread is realized using a very large ring buffer (usually

around 500MB). Each element of the ring buffer represents one second of data. If the buffer is full, the current element is discarded. Due to the size of the buffer, writing of the file might continue for several minutes after the robot has entered the *finished* state.

Due to the limited buffer size, the logging of very large representations such as the *Image* is not possible. It would cause a buffer overflow within seconds rendering the resulting log file unusable. However, without images a log file is nearly useless, therefore loggable thumbnail images are generated and used instead (cf. Sect. 3.7.6).

In addition to the logged representations, online log files also contain timing data, which can be seen using the TimingView (cf. Sect. 8.1.4.5).

3.7.2 Configuring the Online Logger

The online logger can be configured by changing values in the file *logger.cfg*, which should be located inside the configuration folder.

Following values can be changed:

logFilePath: All log files will be stored in this path on the NAO.

maxBufferSize: Maximum size of the buffer in seconds.

blockSize: How much data can be stored in a single buffer slot, i. e. in one second. Note that maxBufferSize×blockSize is always allocated.

representations: List of all representations that should be logged.

enabled: The online logger is enabled if this is true. Note that it is not possible to enable the online logger inside the simulator.

writePriority: Priority of the logging process. Priorities greater than zero use the real-time scheduler, zero uses the normal scheduler.

debugStatistics: If true the logger will print debugging messages to the console.

minFreeSpace: The minimum amount of disk space that should always be left on the NAO in bytes. If the free space falls below this value the logger will cease operation.

3.7.3 Remote Logging

Online logging provides the maximum possible amount of debugging information that can be collected without breaking the real-time capability. However in some situations one is interested in high precision data (i. e. full resolution images) and does not care about real-time. The remote logging feature provides this kind of log files. It utilizes the debugging connection (cf. Sect. 3.5.3) to a remote computer and logs all requested representations on that computer. This way the heavy lifting is outsourced to a computer with much more processing power and a bigger hard drive. However sending large representations over the network severely impacts the NAO's performance resulting in loss of the real-time capability.

To reduce the network load, it is usually a good idea to limit the number of representations to the ones that are really needed for the task at hand. Listing 3.7.3 shows the commands that need to be entered into SimRobot to record a minimal vision log.

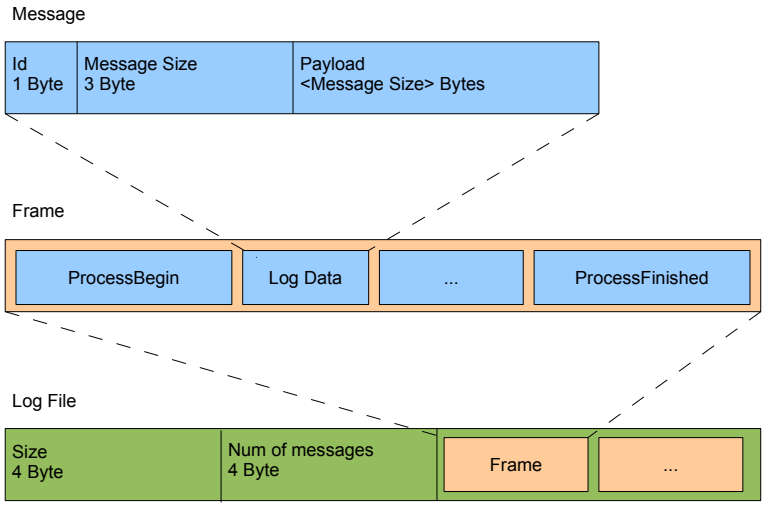


Figure 3.2: This graphic is borrowed from [30] and displays the regular log file format.

```
dr off
dr representation:Image
dr representation:JointData
dr representation:SensorData
dr representation:CameraMatrix
dr representation:ImageCoordinateSystem
dr representation:CameraInfo
(dr representation:OdometryData)
log start
log stop
log save fileName
```

3.7.4 Log File Format

Generally speaking log files consist of serialized message queues (cf. Sect. 3.5.1). Each log file starts with a one byte header. The header determines the kind of log file. Currently there are two different kinds of log files. The enum *LogFileFormat* in *Src/Tools/Debugging/LogFileFormat.h* can be used to distinguish the different log files.

Regular log files start with an eight byte header containing two values. These are the size used by the log followed by the number of messages in the log. Those values can also be set to -1 indicating that the size and number of messages is unknown. In this case the amount will be counted when reading the log file. The header is followed by several frames. A frame consists of multiple log messages enclosed by a `idProcessBegin` and `idProcessFinished` message. Every log message consists of its id, its size and its payload (cf. Fig. 3.2).

Figure 3.3: A compressed log file consists of several regular log files and their compressed sizes.

Compressed log files are lists of regular log files. Each regular file contains one second worth of log data and is compressed using Google's snappy [4] compression (cf. Fig. 3.3). Additionally the list contains the compressed size of each log file.

3.7.5 Replaying Log Files

Log files are replayed using the simulator. A special module, the CognitionLogDataProvider automatically provides all representations that are present in the log file. All other representations are provided by their usual providers, which are simulated. This way log file data can be used to test and evolve existing modules without access to an actual robot.

For SimRobot to be able to replay log files, they have to be placed in *Config/Logs*. Afterwards the SimRobot scene *ReplayRobot.con* can be used to load the log file. In addition to loading the log file this scene also provides several keyboard shortcuts for navigation inside the log file. If you want to replay several log files at the same time, simply add additional *sl* statements (cf. Sect. 8.1.6.1) to the file *ReplayRobot.con*.

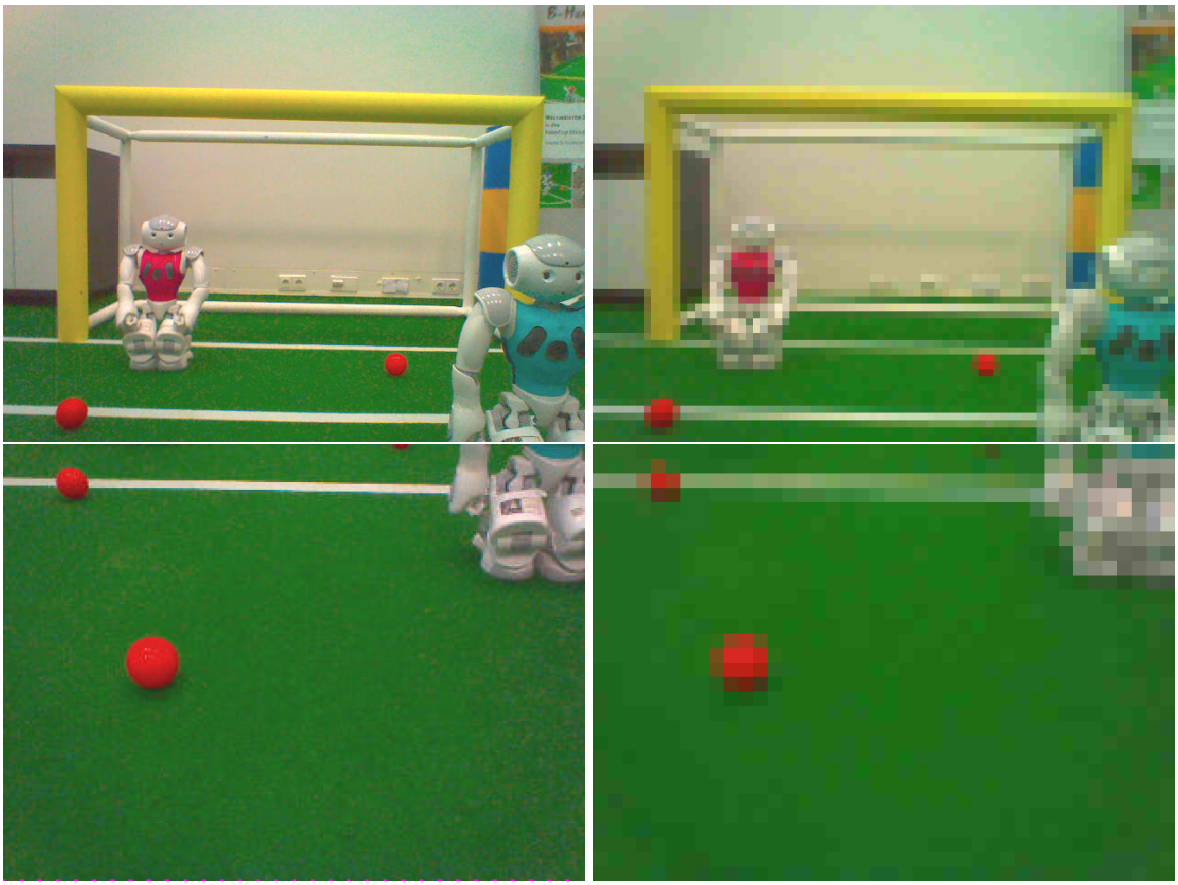
3.7.6 Thumbnail Images

Logging raw images as the cameras capture them seems impossible with the current hardware. One second would consume 45000 KB. This amount of data could not be stored fast enough on the harddrive/USB flash drive and would fill it up within a few minutes. Thus images have to be compressed. This compression must happen within a very small time window, since the robot has to do a lot of other tasks in order to play football. Despite the compression, the resulting images must still contain enough information to be useful.

Compressing the images is done in two steps in order to fulfill the criteria mentioned above. At first, the images are scaled down by averaging blocks of 8×8 pixels. Since this is a very time consuming process, SSE instructions are used. In addition, it is very important not to process block after block since this impedes caching, resulting in great performance losses. Instead, complete rows are processed, buffering the results for eight rows until the average can be computed.

The second step reduces the number of representable colors and removes some unused space from the pixels. The cameras produce images in the YCbCr 4:2:2 format. This means that there are two luma channels for two chroma channels (the data has the form $\{Y_1, Cb, Y_2, Cr\}$). We only use one of the luma channels stored in each pixel since it would cost additional processing time to recreate the full resolution image. Thus, one byte can be removed per pixel for the compressed image. Another byte can be gained by reducing the amount of bits each channel is allowed to use. In our case we use 6 bits for the luma and 5 bits for each chroma channel. Together, this results in another reduction of space by the factor of 2. This step might seem costly but it only takes 0.1 ms, since it is done on a downscaled image.

The resulting compressed images are 128 times smaller than the original ones. One second of images now only needs 351.6 KB of space. Although a lot of detail is lost in the images, it is still possible to see and assess the situations the robot was in (cf. Fig. 3.4).



(a) (b)

Figure 3.4: (a) images from the upper and lower camera of the NAO and (b) the corresponding thumbnails.

Chapter 4

Cognition

In the B-Human system, the process *Cognition* (cf. Sect. 3.2) is subdivided 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 velocity 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 5).

4.1 Perception

Each perception module provides one or more representations for 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, the *GoalPercept* contains information about the goals seen in the image, the *FieldBoundary* separates the image into a foreground (the field) and a background (the surrounding environment), and the *ObstacleSpots* contain information about the location of obstacles in the current 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 Using Both Cameras

The NAO robot is equipped with two video cameras that are mounted in the head of the robot. The first camera is installed in the middle forehead and the second one approx. $4cm$ below. The lower camera is tilted by $39.7°$ with respect to the upper camera and both cameras have a vertical opening angle of $47.64°$. Because of that the overlapping parts of the images are too small for stereo vision. It is also impossible to get images from both cameras at the exact same time, because they are not synchronized on a hardware level. This is why we analyze only one picture at a time and do not stitch the images together. To be able to analyze the pictures from both the upper and lower camera in real-time without loosing any images, the Cognition process runs at 60Hz.

Since the NAO is currently not able to provide images from both cameras at their maximum resolution, we use a smaller resolution for the lower camera. During normal play the lower camera sees only a very small portion of the field, which is directly in front of the robot's feet. Therefore objects in the lower image are close to the robot and rather big. We take advantage

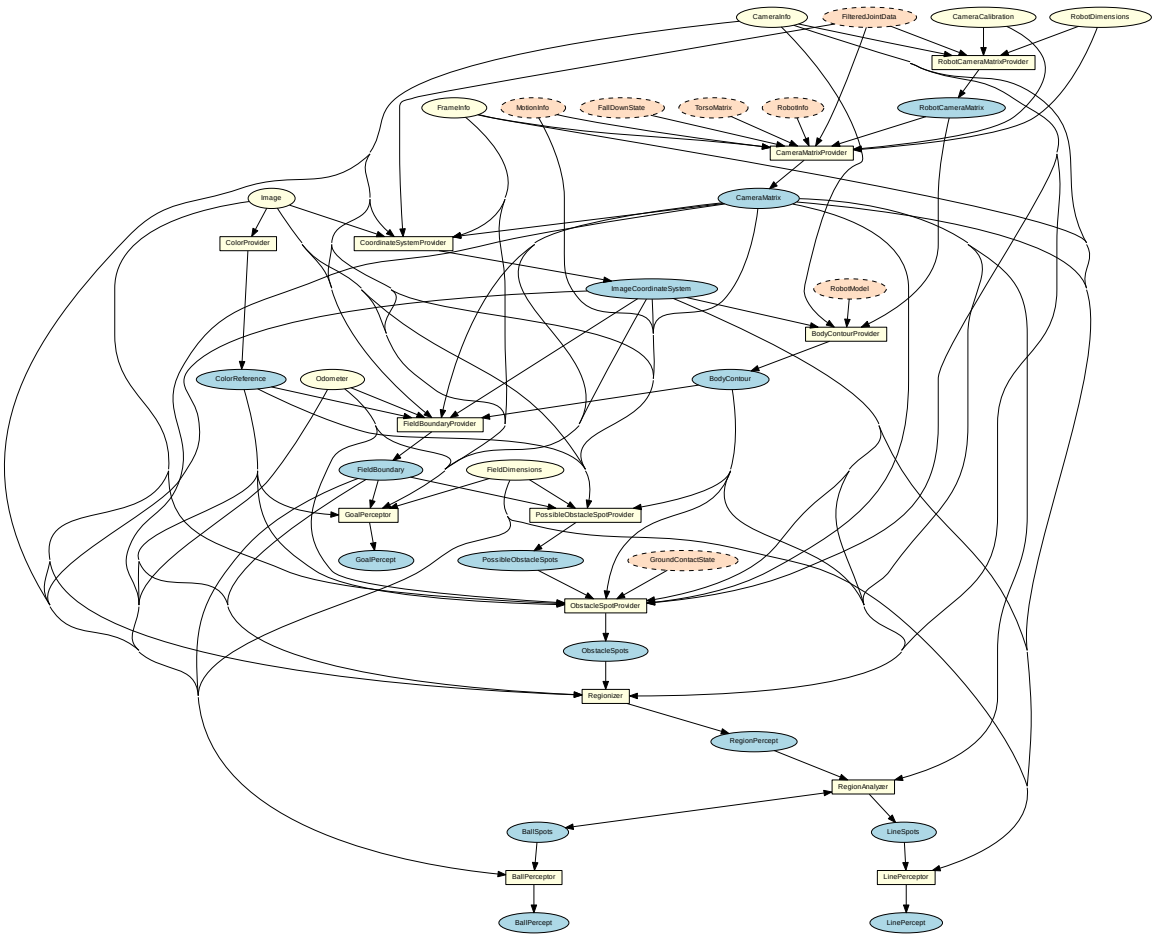


Figure 4.1: Perception module graph. Blue ellipses mark representations provided by Cognition modules, which are marked as yellow squares. Yellow ellipses mark required representations from the Cognition process. In contrast, orange ellipses with a dashed outline mark required representations of the Motion process.

of the fact that objects in the lower image are always big and run the lower camera with half the resolution of the upper camera thus saving a lot of computation time.

Both cameras deliver their images in the *YUV422* format. The upper camera provides 1280×960 pixels while the lower camera only provides 640×480 pixels. They are interpreted as *YUV444* images with a resolution of 640×480 and 320×240 pixels respectively by ignoring every second row and the second Y channel of each *YUV422* pixel pair.

Cognition modules processing an image need to know which camera it comes from. For this reason we implemented the representation *CameraInfo*, which contains this information as well as the resolution of the current image.

4.1.2 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°,

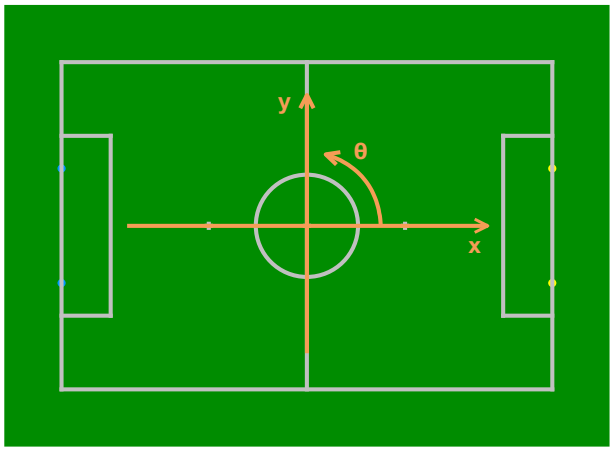


Figure 4.2: Visualization of the global coordinate system (opponent goal is marked in yellow)

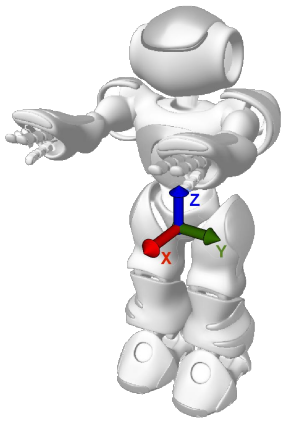


Figure 4.3: Visualization of the robot-relative coordinate system

and the y-axis pointing toward 90°.

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

4.1.2.1 Camera Matrix

The *CameraMatrix* is a representation containing the transformation matrix of the active camera of the NAO (cf. Sect. 4.1.1) that is provided by the CameraMatrixProvider. It is used for projecting objects onto the field as well as for the creation of the *ImageCoordinateSystem* (cf. Sect. 4.1.2.2). 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. 6.1.7). Using the *RobotDimensions* and the current joint angles the transformation of the camera matrix relative to the torso matrix is computed as the *RobotCameraMatrix*. The latter is used to compute the *BodyContour* (cf. Sect. 4.1.3). In addition to the fixed parameters from the *RobotDimensions*, some robot-specific parameters from the *CameraCalibration* are integrated, which 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 process of manually calibrating the robot-specific correction parameters for a camera is a very time-consuming task, since the parameter space is quite large (8 resp. 11 parameters for calibrating the lower resp. both cameras). It is not always obvious, which parameters have to be adapted if a camera is miscalibrated. In particular during competitions, the robots' cameras require recalibration very often, e. g. after a robot returned from repairs.

In order to overcome this problem, we developed the semi-automatic calibration module `CameraCalibratorV4` to simplify the calibration process. Its main features are:

- The user can mark arbitrary points on field lines. This is particularly useful during competitions because it is possible to calibrate the camera if parts of the field lines are covered (e. g. by robots or other team members).
- Since both cameras are used, the calibration module is able to calibrate the parameters of the lower as well as the upper camera. Therefore, the user simply has to mark additional reference points in the image of the upper camera.

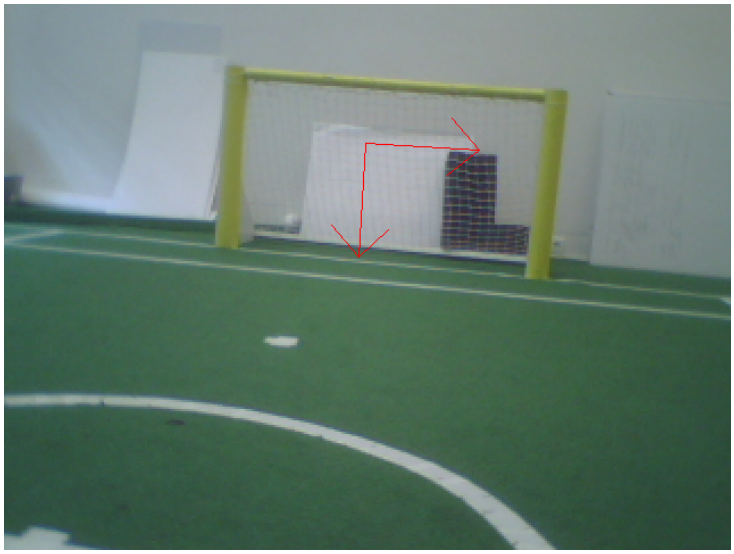


Figure 4.4: Origin of the *ImageCoordinateSystem*

- In order to optimize the parameters, the Gauss-Newton algorithm is used[1] instead of hill climbing. Since this algorithm is designed specific for non-linear least squares problems like this, the time to converge is drastically reduced to an average of 5–10 iterations. This has the additional advantage that the probability to converge is increased.
- During the calibration procedure, the robot stands on a defined spot on the field. Since the user is typically unable to place the robot exactly on that spot and a small variance of the robot pose from its desired pose results in a large systematical error, additional correction parameters for the *RobotPose* are introduced and optimized simultaneously.
- The error function takes the distance of a point to the next line in image coordinates instead of field coordinates into account. This is a more accurate error approximation because the parameters and the error are in angular space.

With these features, the module typically produces a parameter set that requires only little manual adjustments, if any. The calibration procedure is described in Sect. 2.8.3.

4.1.2.2 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. The 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 whereby the y-axis points downwards orthogonal to the horizon (cf. Fig. 4.4). For more information see also [28].

Using the stored camera transformation matrix of the previous cycle in which the same camera took an image 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 [22].

[1] Actually, the Jacobian used in each iteration is approximated numerically

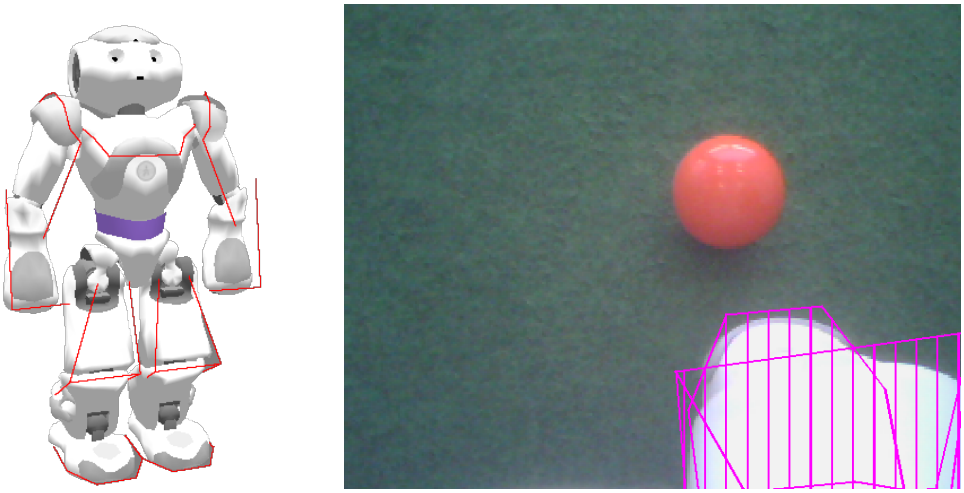


Figure 4.5: Body contour in 3-D (left) and projected to the camera image (right).

4.1.3 Body Contour

If the robot sees parts of its body, it might confuse white areas with field lines or other robots. However, by using forward kinematics, the robot can actually know where its body is visible in the camera image and exclude these areas from image processing. This is achieved by modeling the boundaries of body parts that are potentially visible in 3-D (cf. Fig. 4.5 left) and projecting them back to the camera image (cf. Fig. 4.5 right). The part of the projection that intersects with the camera image or above is provided in the representation *BodyContour*. It is used by image processing modules as lower clipping boundary. The projection relies on the representation *ImageCoordinateSystem*, i. e., the linear interpolation of the joint angles to match the time when the image was taken.

4.1.4 Color classification

A fast identification of color classes is achieved by precomputing the class of each YCbCr color and store it in an array. To save space the dimensions of this color space is reduced from 256×256×256 down to 128×128×128 colors, binning eight adjacent colors into one. This is still precise enough, but saves resources.

HSV color space

To easily define color classes the HSV color space is used. It consists of the three channels hue, saturation, and value. While hue defines the angle of the vector pointing to a color on the RGB color wheel (cf. Figure 4.6), saturation defines the length of this vector. At last, value defines the brightness of this color. Therefore each color class, except for white and black (cf. Sect. 4.1.4), is described through six parameters, minimum hue, maximum hue, minimum saturation, maximum saturation, minimum value, and maximum value.

White and black

Using HSV for the definition of the color classes white and black is not very useful. Furthermore black is very constant, regardless of lighting conditions. Therefore it is defined statically in YCbCr. Finally defining the color class white is handled using the RGB color space in a special

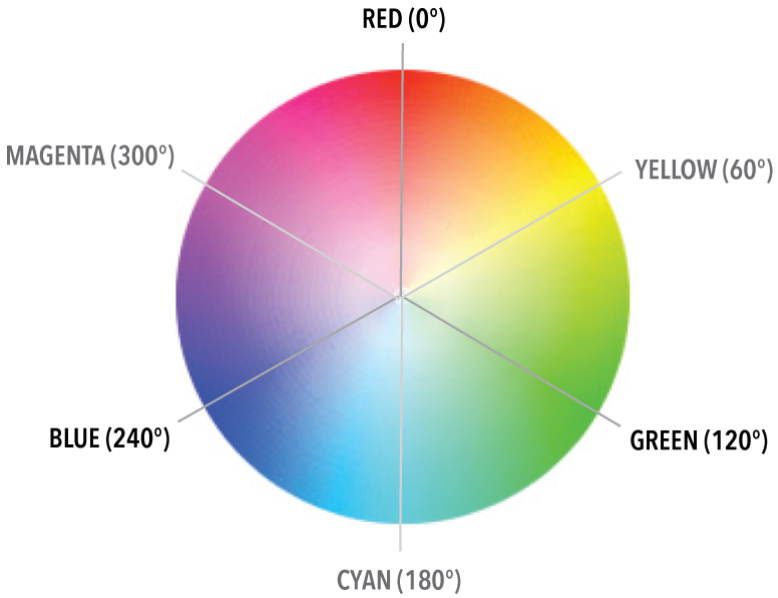


Figure 4.6: RGB color wheel used by the HSV color space.

way. In theory, white is composed through large values for red, green and blue. Due to the noise level of the camera, white pixels - especially at line borders - sometimes appear blueish. Therefore white is defined through a minimum of red, a minimum of blue and a minimum of the sum of red and blue[2]. Furthermore white pixels can't be green and are strictly excluded, otherwise the field might get white, too.

4.1.5 Segmentation and Region-Building

The Regionizer works on the camera images and creates regions based on segments found on scan lines. It provides the *RegionPercept* containing the segments and the regions.

Since all features that are detected based on regions are on the field (ball and field lines), further segmentation starts from the *FieldBoundary* (cf. Sect. 4.1.11). 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. Thus, scan lines running from the top of the image, clipped by *FieldBoundary*, 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 try to find equally colored runs between the current scan line and the last one that 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. 4.7).

Next to the regular scan lines, some additional ones are used between those. These scan lines only recognize orange regions and end 30 pixels under the start point or if the image end is reached. In this way, we can also find balls even if the distance from the robot is that large that

[2]In some locations it might be useful to set the parameter greater than the sum of red and blue themselves, but most of the time it is equal to them.

balls would disappear between usual scan lines.

Furthermore, the result from the representation *ObstacleSpots* is taken to ignore white segments within ban zones. These ban zones are provided by *ObstacleSpots* (cf. Sect. 4.1.10) and enclose found obstacles.

Based on these segments, regions are created using the algorithm shown in Algorithm 1 and Algorithm 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 of the previous column, which 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 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

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
```

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. 4.7). For example, at a corner the change in direction of the white segments should exceed the maximum change in direction. Therefore, they are not united. Alternatively, regions are also not united if the ratio of the length of two segments becomes too large, which is often caused by a robot standing on a field line.

4.1.6 Region Classification

The region classification is done by the RegionAnalyzer. It classifies for 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*.

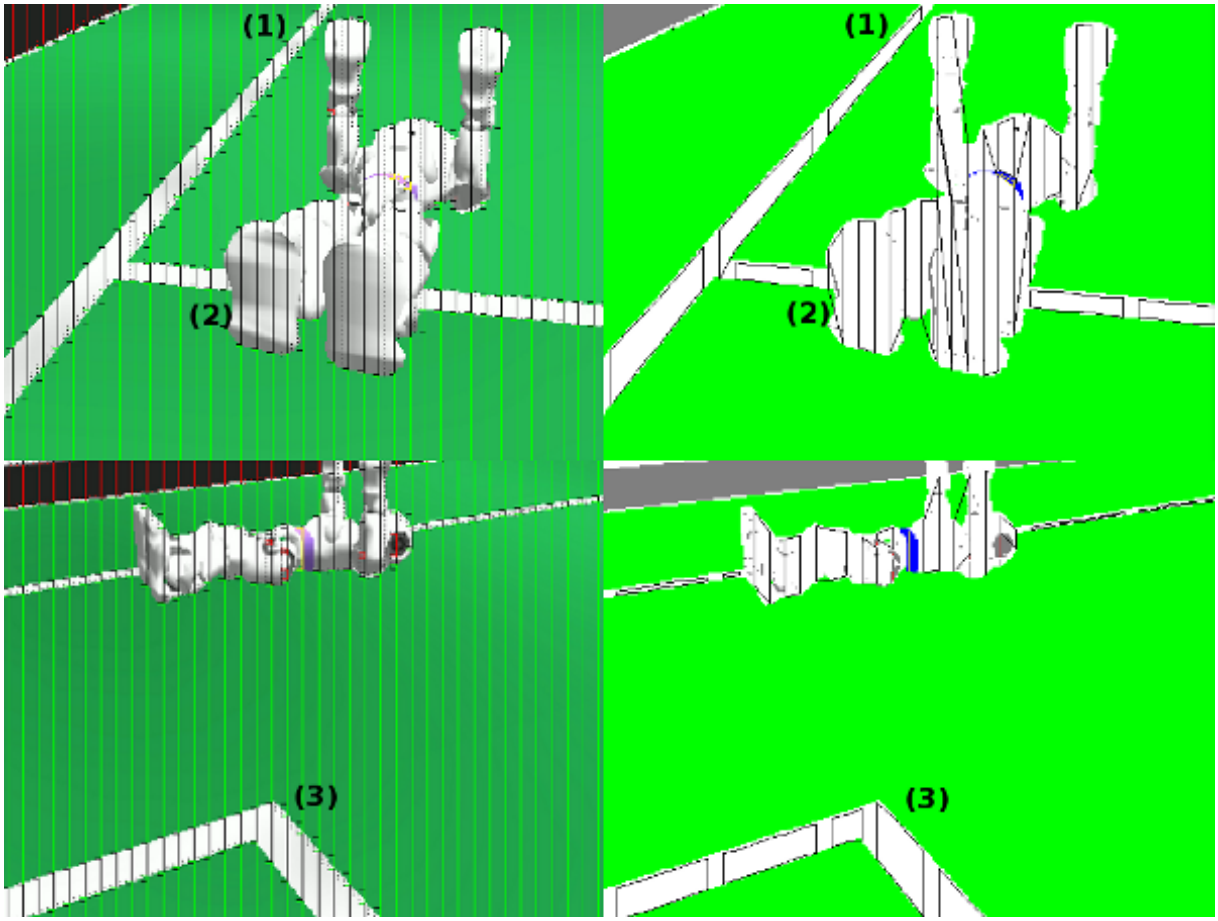


Figure 4.7: Segmentation and region-building: (1) segment is not touching the other one, but the explored runs (dotted lines) are touching the other segment, (2) not connected because of length ratio, (3) not connected because of change in direction.

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

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

For each region that was classified as 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 line spot in image coordinates are stored as *LineSpot* in the *LineSpots*. All

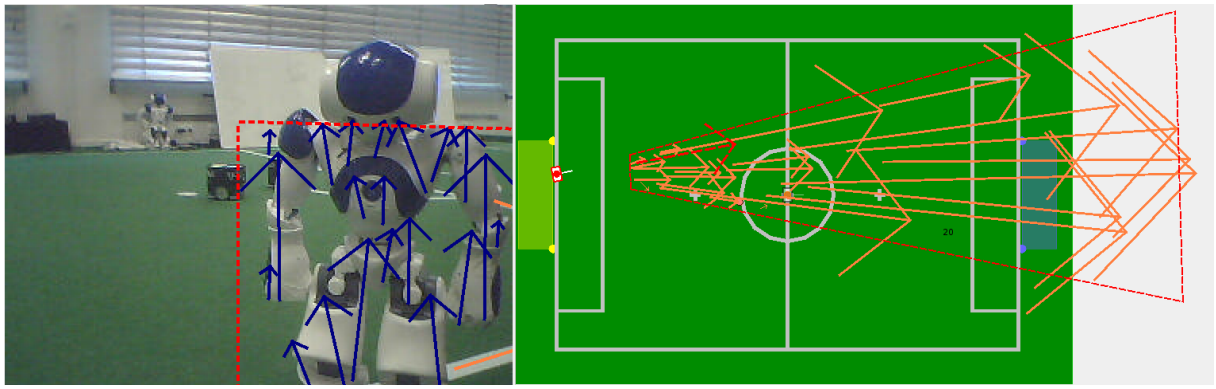


Figure 4.8: Ban sector built from *NonLineSpots*. The red arrow is a false line spot detected in the robot and discarded because of the ban sector.

white regions that did not fulfill the criteria to be part of a line are filtered again by a very basic filter that determines whether the region could be part of a robot. The conditions for this filter are the following:

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

If all of these conditions are met, a *NonLineSpot* is added to the *LineSpots*. These *NonLineSpots* are used in the LinePerceptor to find regions in the image that consist of many *NonLineSpots*. These are excluded from line detection.

Orange regions are accepted as possible balls and stored as *BallSpot* in the *BallSpots*.

All thresholds are configurable through the file *regionAnalyzer.cfg*.

4.1.7 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. 4.8).

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 that 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 Algorithm 3. The basic idea of this algorithm is similar to the quality threshold clustering algorithm introduced by Heyer et al. in [7], 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 accept the two segments 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.

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
```

All remaining line segments are taken into account for the circle detection. For each pair of segments with a distance smaller than a threshold, the intersection of the perpendicular from the middle of the segments is calculated. If the distance of this intersection is close to the real circle radius, for each segment, a spot is generated which has the distance of the radius to the segment. After the spots are created, the same clustering algorithm used for the lines is used to find a cluster for the circle. As soon as a cluster is found that fulfills the criteria to be a circle, it is assumed to be the circle (cf. Fig. 4.9). For all remaining line segments that 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 that are aligned with the circle are deleted. It might also happen that a single line in the image creates multiple lines (because the line was not clustered but the single segments are long enough to create lines on their own). Therefore, lines that are similar (with respect to the Hesse normal form) are merged together. It might happen that single segments were not clustered to a line, which is the case if the line spot is not perfectly

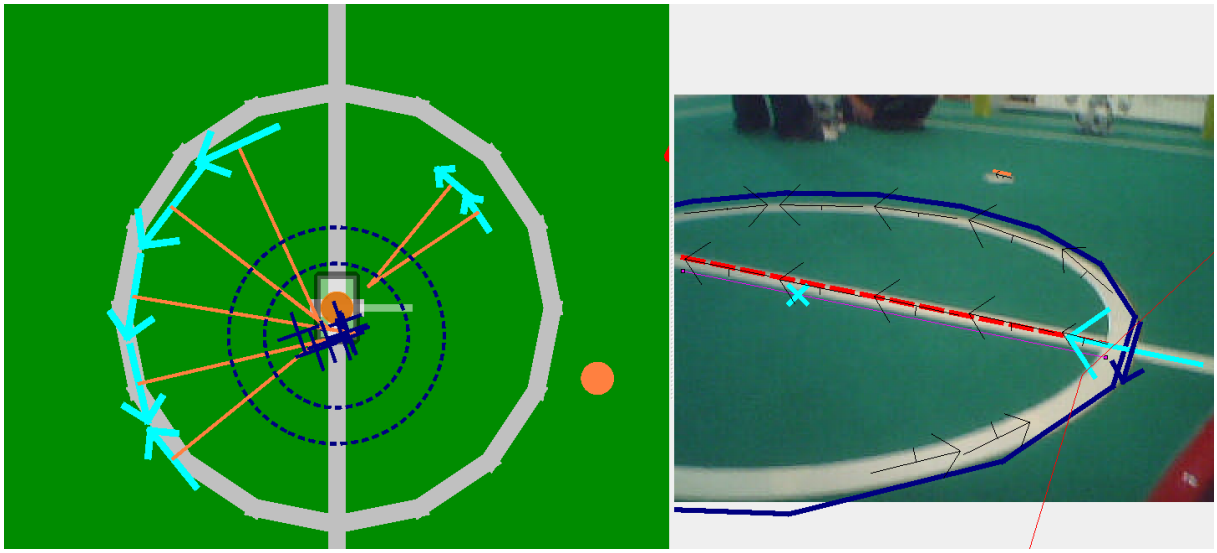


Figure 4.9: Circle detection: blue crosses: circleSpots, blue circles: threshold area, orange lines: perpendiculars of segments

aligned with the line. This can be caused by an inaccurate segmentation. The remaining single segments are merged into 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 that are far away from each other, for example the cross and a part of the circle.

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

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

4.1.8 Detecting the Goal

The GoalPerceptor detects goalposts in the image and provides them in the *GoalPercept*. The general idea is to find yellow regions in the image and evaluate their shape and distance to determine whether they are real goalposts or not. A scan line based method is used to find the yellow regions. As goalposts always intersect the horizon, which is provided by the *ImageCoordinateSystem* (cf. Sect. 4.1.2.2), a first horizontal scan is performed on the height of the horizon in the image. If the horizon is above the image, the scan is done at the top of the image instead.

For each yellow region on the horizon, the height is calculated using vertical scan lines. The height is determined using a simple recursive algorithm that successively scans upward and downward. The first step of the downward scanning is to calculate the middle of the current region. The result is a point in the image from which the algorithm scans downwards, until there is no more yellow. Then in the middle of this vertical scan line (cf. Fig. 4.10), a horizontal scan follows to determine the width of the region. This scan line is used as new current horizontal yellow line. This continues recursively until two consecutive vertical scans end at the same x-coordinate indicating that the bottom of the goalpost has been found. One final horizontal scan is done at the bottom to determine the width of the goalpost. This algorithm gets to the bottom in big steps and is more accurate the nearer it is.

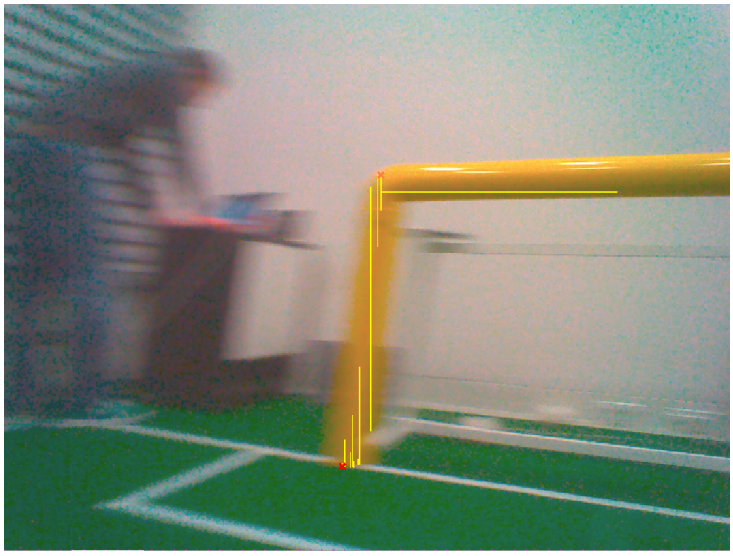


Figure 4.10: Scanning for base and top of a potential goalpost

Scanning upwards works in a similar way. However a second termination condition is introduced. The reason is the crossbar, where the width of a goalpost drastically grows. If such a phenomenon is recognized during an upward scan, a crossbar with a certain direction (left or right in the image) is found.

The data provided by the previous scans is stored. It represents potential goalposts. For each of them the position on the field is calculated. If the current image was taken by the upper camera and if the bottom limit of the yellow region reaches out of the bottom of the image, the goalposts found in the previous image (provided by the lower camera) are compared to the spot. If they match, the position of this post, corrected by the odometry offset is taken. If no goalpost matches, the position is estimated by the width of the region.

The spots are then evaluated for several matching criteria to the appearance real goalposts would have in the image. First there are four exclusion criteria:

1. The distance to the potential goalpost is not allowed to be greater than the field diagonal.
2. The height of the yellow region must be equal or greater than the height a goalpost would have in the image that is positioned one field diagonal away from the robot.
3. The base of the spot must be below the field boundary (cf. Sect. 4.1.11).
4. All width scans must have similar values.

After that three numerical evaluations are made:

- The ratio between width and height of the region is compared to the ratio that the real goalpost has.
- With the calculated distance, the expected width and height can be computed. These are compared to the width and height of the region, which are clipped with the image border first.

These values have a range from 100, representing a perfect match, to large negative values. This ensures that very bad matches outvote good ones.

At last the relation between different spots is evaluated and their rating is increased

- if the distance between two spots matches the distance between real goalposts
- or two spots have matching crossbars.

Afterwards all valuations are combined for each spot. Finally the two best evaluated spots above a minimum rating are provided as the detected goalposts.

Parameters

The module provides two parameters that affect its behavior:

yellowSkipping: The amount of non yellow pixels that can be skipped before not recognizing a yellow region anymore. Increase if there is a high noise in the image. Or see Sect. 4.1.4.

quality: The minimum quality for spots to be considered as correct goalposts. Increase when having false positive detected posts.

Debugging

The module provides three debug drawings to illustrate its activity.

Spots: Draws the scan line on which it searches for yellow regions and marks their start with a green and the end with a red cross.

Scans: Draws the scan lines on which the yellow regions are expanded up- and downwards.

Validation: Draws all validation values for each spot. This can be restricted by modifying `module:GoalPerceptor:low/high`.

Common sources of errors are the color segmentation not providing sufficient yellow association, which will result in not seeing crosses in the *Spots* debug drawing, and the field boundary being detected too low in the image, leading to a bad validation.

4.1.9 Detecting the Ball

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

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

1. Checking for noise: Ball spots that are too small in width and height will be filtered out to exclude noisy orange pixels.
2. Checking the position: Ball spots that are further away than the length of the field diagonal or above the horizon are removed.
3. 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.

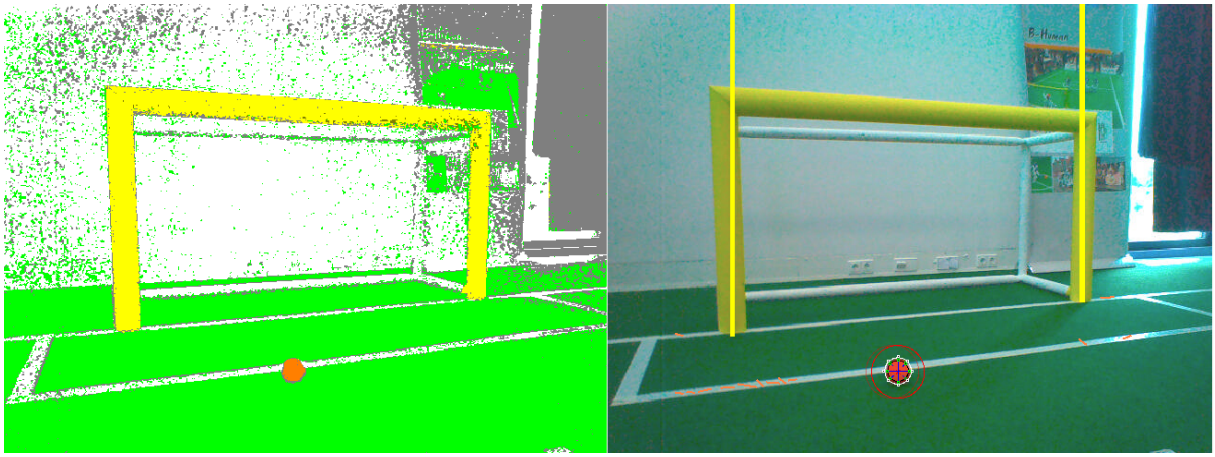


Figure 4.11: The debug drawing of the BallPerceptor. The green dots mark the end of the scan lines for ball points. The white circle should be around the ball, i. e. the color differs too much from the ball spot.

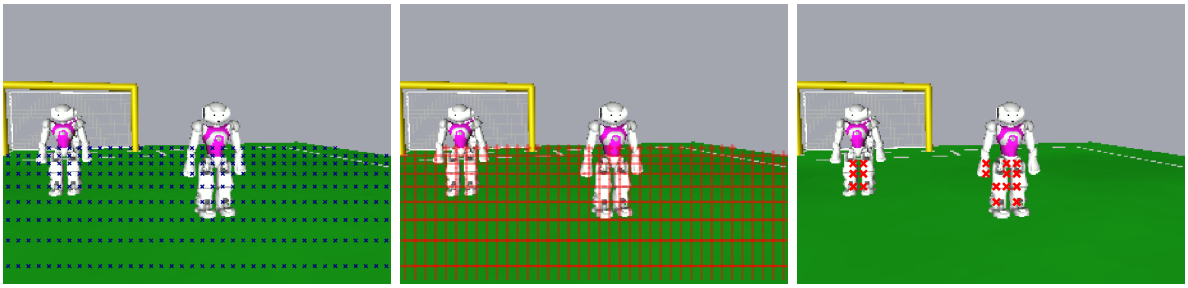
4. Checking the ball points: The ball points will be validated. Duplicates and ball points with a high deviation of the radius will be removed.
5. Calculating the center and the radius of the ball.
6. Checking the environment in the current image: Since the radius of the ball in the image is known, it can be checked whether there are any ball points outside the radius. To accomplish this 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.
7. 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.
8. Checking the field boundary: If the ball is outside the field, using the *FieldBoundary*, the ball will be filtered out.
9. Checking for red jersey color: The calculated ball points will be used to span a rectangle around all points. Inside the rectangle the relation between all pixels and the number of counted red pixels will be calculated. If this relation is bigger than a constant threshold, the ball is ignored.

The BallPerceptor only defines its parameters in the code, e. g.

minBallSpotSize: The minimum width/height for ball spots used in step 1. If the environment is very noise increasing this parameter might be useful.

orangeSkipping: Ball spots themselves might have some none-orange pixels. This value allows the ball spots to have n such pixels while determining its width and height. Typical values are 1–3.

scanMaxColorDistance: The maximum distance for the Cb and Cr component used while scanning for ball points. If this value is too small, ball percepts inside the red jerseys may occur. If the value is to high, balls on field lines may be ignored.



(a) (b) (c)

Figure 4.12: Each blue spot in (a) is checked whether it contains an obstacle or not. The highlighted pixels in (b) show the *neighborhoods* around each grid point. (c) shows the detected possible obstacle spots.

useFieldBoundary: Exclude balls percepts outside the field boundary.

percentRedNoise: The maximum percentage of allowed red jersey pixels inside the rectangle described in step 9. Decrease this value if ball percepts occur in red jerseys.

The resulting *BallPercept* is the input for the module BallLocator (cf. Sect. 4.2.2).

4.1.10 Detecting Other Robots and Obstacles

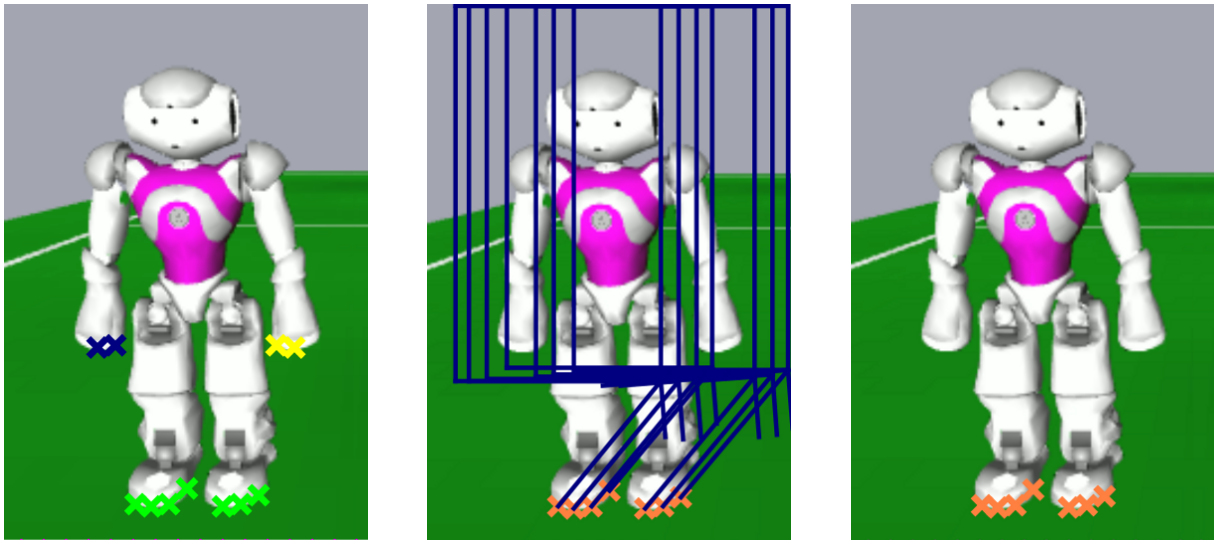
The visual obstacle detection is based on the idea that obstacles are big non-green areas in the image. It is split into an initial detection stage and a refinement stage. The initial detection generates a list of spots in image coordinates that might be anywhere inside a possible obstacle. Afterwards, the refinement step filters these spots, using several sanity checks. In addition, it stitches obstacles together that span both camera images.

Initial Obstacle Detection

The goal of this stage is to detect a few spots inside every obstacle. These spots are later used as starting points in the ObstacleSpotProvider. Their exact position does not matter as long as they are somewhere inside an obstacle.

The initial detection is based on the assumption that each non-green area that is bigger than a field line has to be an obstacle. The whole image is searched for such regions. To be able to search the image in a reasonable time, not every pixel is checked. Instead, a grid is utilized (cf. Fig. 4.12a). The vertical distance between two grid points is variable. It is calculated by projecting points to the image plane that are precisely 100 mm apart in the field plane. The horizontal distance between two grid points is fixed. To further reduce the runtime, pixels above the field boundary (cf. Sect. 4.1.11) are not checked.

For each grid point it is determined whether the point lies inside a non-green region or not. This is achieved by checking whether more than $k\%$ (currently $k = 93$) of the neighborhood around each point is not green. The neighborhood is a cross-shaped region that consists of the $2n$ neighboring pixels in vertical and horizontal direction (cf. Fig. 4.12b). n is the field line width at the given grid point, i. e. half the vertical distance between the grid point and its neighbor. The cross shape has been chosen due to its relative rotation invariance and ease of implementation.



(a) (b) (c)

Figure 4.13: Removal of false obstacle spots at the hands of other robots. The colored crosses in (a) are different obstacle spot clusters. The blue and the yellow clusters are false positives and should be removed. (b) shows the calculated ban regions and (c) shows the result.

Since the horizontal distance between two grid points is fixed, but the width of a neighborhood depends on the distance of the grid point, the neighborhoods of two adjacent grid points overlap. To avoid checking pixels for their greenness multiple times in the overlapping areas, a one-dimensional sum image is calculated for each grid row beforehand. The greenness of a certain region can then be calculated by subtracting the corresponding start and end pixels in the sum image. All grid points that lie inside non-green regions are considered to be possible obstacles.

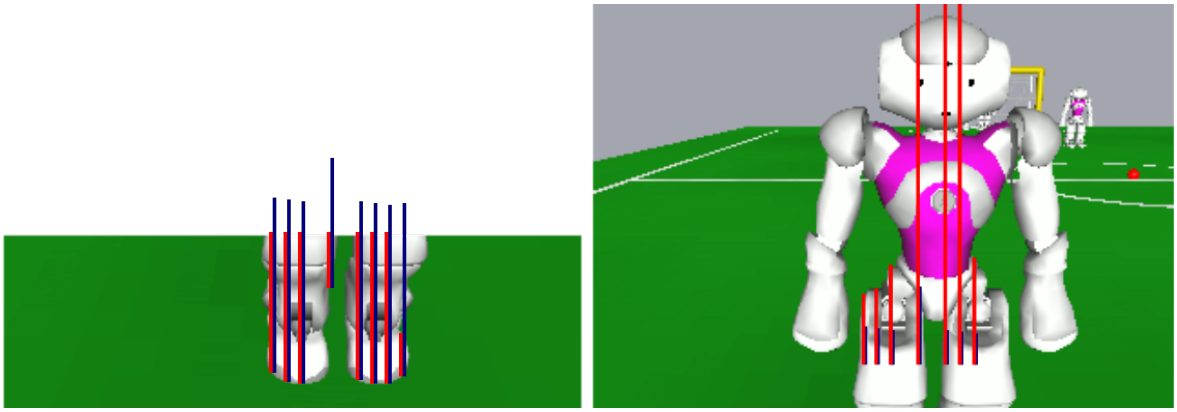
Filtering of Possible Obstacle Spots

The possible obstacle spots show the general location of obstacle regions in the image. However, they usually do not show the foot point of an obstacle and neither do they show its size. In addition, they may contain false positives, in particular at field line junctions. Therefore, additional filtering is required. The filtering process consists of several steps:

Height check: The first step of the filtering process is to determine the height and the foot point of the obstacle at each possible spot. This is done by scanning the image vertically in both directions starting at each possible obstacle spot until either the horizon, i. e. the plane parallel to the field on the height of the camera, or a continuous patch of green is found. This way, the height can be determined well as long as the camera is more or less parallel to the ground. Since the NAO does not need to tilt its head and the body is kept upright while walking, this is usually the case.

If the height is less than a robot's width at the foot point, the obstacle spot will be discarded. This effectively removes most of the false positives. However, due to the fact that the environment around a RoboCup field is usually not green, false positives close to the field border cannot be removed using only a height criterion. Still, since robots are not allowed to leave the field, false positives at the field border do not matter.

Hand removal: Depending on their posture, the hands of other robots may be detected as obstacle spots. Technically, those spots are not false positives since they are part of



(a) (b)

Figure 4.14: Obstacle stitching. Red lines show the measured height while blue lines indicate the minimal expected height. The image from the lower camera (a) does not contain enough of the obstacle to pass the height check; therefore, the height check is continued in the upper camera image (b).

another robot. However, the foot point of such obstacles is not on the field plane. Thus it is impossible to calculate the distance to the obstacle. These obstacles are removed using the assumption that the lowest (in image coordinates) obstacle spot belongs to the foot of a robot. Starting from the lowest spot, an area that should contain the robot's hands is calculated for each spot. All spots inside that area are removed. Figure 4.13 illustrates this process.

Obstacle Stitching: When obstacles are close, they span both the lower and upper camera image. The foot point of such an obstacle is in the lower image. However, the lower image does usually not show enough of the obstacle to pass the height check. In this case, the visible height is calculated and the intersection of obstacle and image border is converted to relative field coordinates and buffered.

When the next image from the upper camera is processed, the buffered points are moved (still in relative field coordinates) according to the odometry and projected back into the upper camera image. Here, the height check that was started in the lower image is continued (cf. Fig. 4.14).

Clustering: Finally, the remaining obstacle spots are clustered and the center of mass is calculated for each cluster.

4.1.11 Detecting The Field Boundary

The 2013 rules state that if fields are further away from each other than 3 m, a barrier between them can be omitted. This means that robots can see goals on other fields that look exactly like the goals on their own field. In addition, these goals can even be closer than a goal on their field. Therefore, it is very important to know where the own field ends and to ignore everything outside. In doing so our old approach determined the field boundary by scanning the image downwards to find the first green of sufficient size. Given the new rules this could easily be found on another field. Thus, a new approach was required. It searches vertical scan lines, starting from the bottom of the image going upwards. Simply using the first non-green pixel for the boundary is not an option, since pixels on field lines and all other objects on the field would

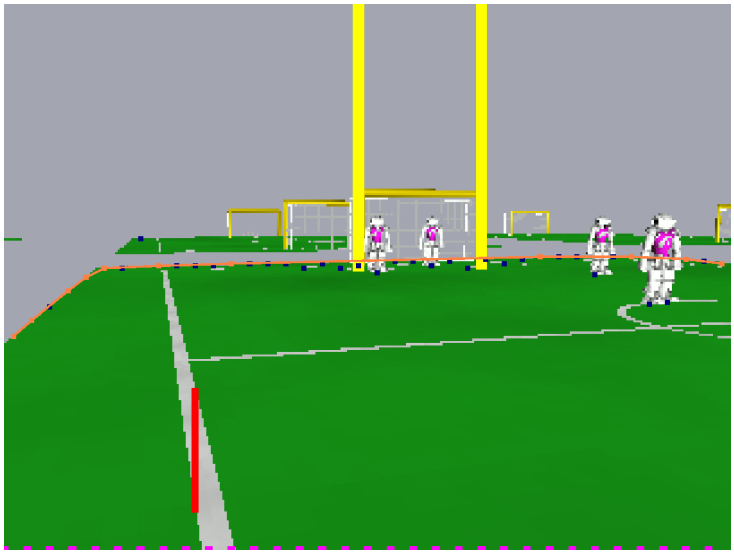


Figure 4.15: Boundary spots (blue) and the estimated field boundary (orange) in a simulated environment with multiple fields.

comply with such a criterion. In addition, separating two or more fields would not be possible and noise could lead to false positives.

Our approach builds a score for each scan line while the pixels are scanned. For each green pixel a reward is added to the score. For each non-green pixel a penalty is subtracted from it. The pixel where the score is the highest is then selected as boundary spot for the corresponding scan line. The rewards and penalties are modified depending on the distance of the pixel from the robot when projected to the field. Field lines tend to have more green pixels above them in the image. As a result, they are easily skipped, but the gaps between the fields tend to be small in comparison to the next field when they are seen from a greater distance. So for non-green pixels with a distance greater than 3.5 meters, a higher penalty is used. This also helps with noise at the border of the own field. Figure 4.15 shows that most of the spots are placed on the actual boundary using this method.

Since other robots in the image also produce false positives below the actual boundary, the general idea is to calculate the upper convex hull from all spots. However, such a convex hull is prone to outliers in upward direction. Therefore, an approach similar to the one described by Thomas Reinhardt [20, Sect. 3.5.4] is used. Several hulls are calculated successively from the spots by removing the point of the hull with the highest vertical distances to its direct neighbors. These hulls are then evaluated by comparing them to the spots. The best hull is the one with the most spots in near proximity. This one is selected as boundary. An example can be seen in Fig. 4.15.

The lower camera does not see the actual field boundary most of the time. So calculating the boundary in every image would lead to oscillating representations. Thus the field boundary is calculated on the upper image. Nevertheless, a convex hull is calculated on the lower image too, since the boundary can start there. The resulting hull is stored for one frame in field coordinates so that the movement of the NAO can be compensated in the upper image. It is then used to provide starting points for the calculations. The resulting hull is used as field boundary over two images.

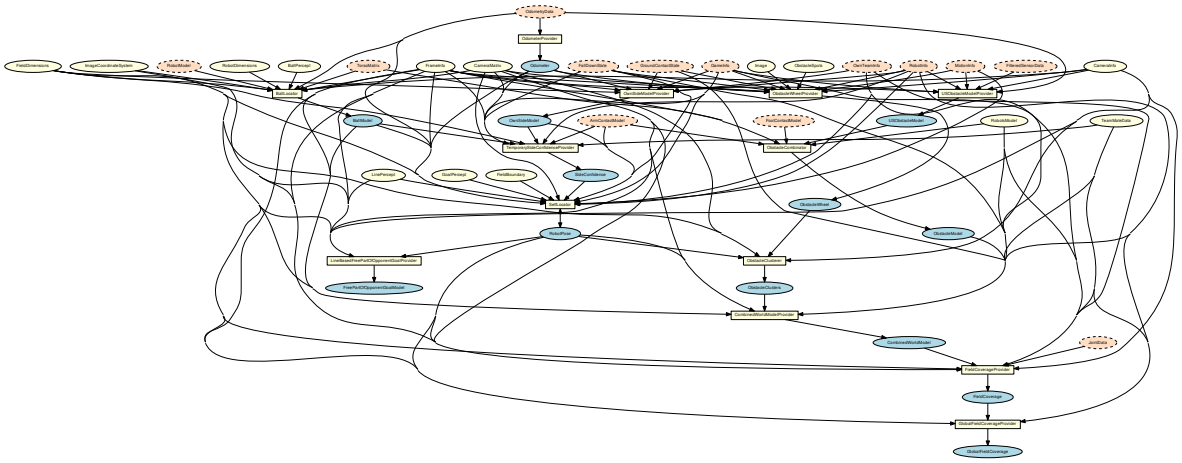


Figure 4.16: Modeling module graph. Blue ellipses mark representations provided by cognition modules, which are marked as white squares. White ellipses mark required representations from the cognition process. In contrast, red ellipses with a dashed outline mark required representations of the motion process.

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

A robust and precise self-localization has always been an important requirement for successfully participating in the Standard Platform League. B-Human has always based its self-localization solutions on probabilistic approaches [29] as this paradigm has been proven to provide robust and precise results in a variety of robot state estimation tasks.

Since many years, B-Human's self-localization is realized by a particle filter implementation [1, 13] as this approach enables a smooth resetting of robot pose hypotheses. However, in recent years, different additional components have been added to complement the particle filter: For achieving a higher precision, an Unscented Kalman Filter [10] locally refined the particle filter's estimate, a validation module compared recent landmark observations to the estimated robot pose, and, finally, a side disambiguator enabled the self-localization to deal with two yellow goals [27].

Despite the very good performance in previous years, we are currently aiming for a new, closed solution that integrates all components in a methodically sound way and avoids redundancies, e. g. regarding data association and measurement updates. Hence, the current solution – the SelfLocator that provides the *RobotPose* – has been designed as a particle filter with each particle carrying an Unscented Kalman Filter. While the UKF instances perform the actual state estimation (including data association and the consequent hypothesis validation), the particle filter carries out the hypotheses management and the sensor resetting.

However, the side disambiguation (based on knowledge about game states and a team-wide ball model) still remains a separate component – the TemporarySideConfidenceProvider that computes

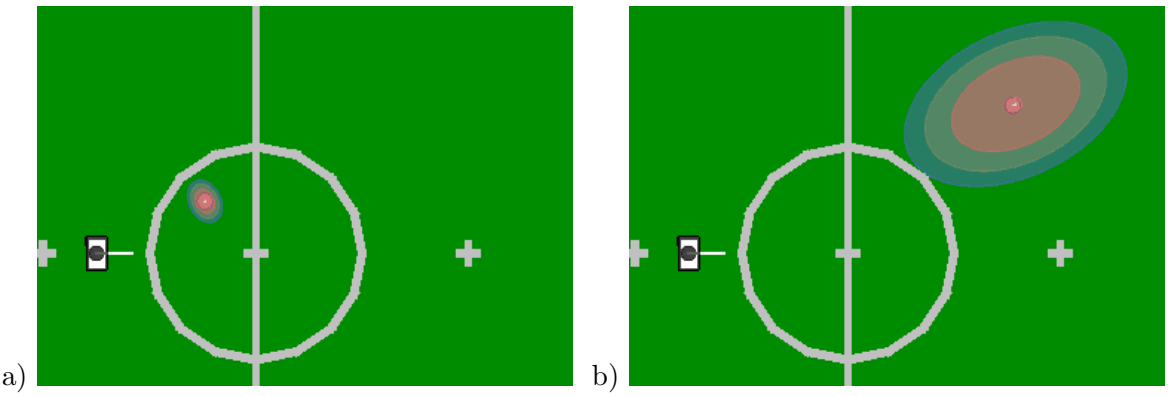


Figure 4.17: Ball model and measurement covariances for short (a) and medium (b) distances. The orange circles show the ball models computed from the probability distribution. The larger ellipses show the assumed covariances of the measurements.

the so-called *SideConfidence*. Current work in progress is to include this side assignment in the main state estimation process by adding it to each particle's state.

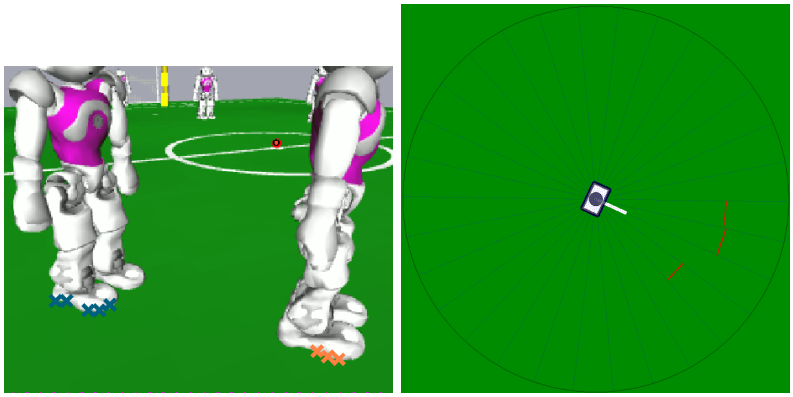
4.2.2 Ball Tracking

The module BallLocator uses Kalman filters to derive the actual ball motion, the *BallModel*, given the perception described in Sect. 4.1.9. Since ball motion on a RoboCup soccer field has its own peculiarities, our implementation extends the usual Kalman filter approach in several ways described below.

First of all, the problem of multimodal probability distributions, which is 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 very accurate estimation 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 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.



(a) (b)

Figure 4.18: Visual obstacle model: (a) shows two detected visual obstacles and (b) their representation in the obstacle model

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 (cf. Fig. 4.17).

This year, we changed the equations that model the movement of the ball. Instead of the uniform deceleration we assumed before, we now use the physical more realistic equation $\dot{v} = cv$. The parameter c depends on the friction between the ball and the ground and has to be configured in the file *fieldDimensions.cfg* (cf. Sect. 2.9). The explicit solution $x(t) = x_0 - \frac{v_0}{c} + \frac{v_0}{c}e^{ct}$ is used for calculations, for example to determine the end position of the ball.

We also added the module FrictionLearner in order to automatically determine the floor-dependent friction parameter. The FrictionLearner can be used as follows: Place the robot on the field and roll the ball several times through it's field of view. While doing that, the robot calculates the parameter c by determining the acceleration and velocity by filtering the ball percepts with a Savitzky-Golay filter and estimating c by using the normal equations. Unfortunately, this procedure does not converge very quickly and is very sensitive to outliers, therefore you need to check the result carefully and adjust it manually if needed.

4.2.3 Obstacle Modelling

B-Human uses several obstacle models. Each has different advantages and drawbacks: The *USObstacleGrid* contains robust information about the presence of obstacles in the robot's vicinity. However, in addition to the low resolution, the NAO's ultrasonic sensors have only a limited range in which they operate reliably. Robots that lie on the ground cannot be perceived at all. The *ObstacleWheel* contains much more precise measurements and is able to detect fallen robots but does not work reliably when the obstacles are very close to the robot. Finally, the *ArmContactModel* contains information about collisions with other robots at the left and right side of the robot, an area that is in many situations not covered by the other two models.

The following sections describe the different modules that provide the models mentioned above in detail.

4.2.3.1 Visual Obstacle Model

Due to the small field of view of the robot's cameras only a small portion of the field is visible in each camera image. Additionally, obstacles may not be detected in each frame. Therefore a model is needed that collects and averages the obstacle locations over time. The *ObstacleWheel* is such a model. It is provided by the ObstacleWheelProvider and represents the environment of the robot using a radial data structure (cf. Fig. 4.18b) similar to the one described in [8].

The radial structure is divided into n cones. Each cone stores the distance to a seen obstacle and the number of times that the obstacle was recently seen. The counter is used to decide whether the distance information in a cone can be trusted and is incremented each time an obstacle is seen and decremented once each second. If it falls below a constant threshold, the obstacle is ignored.

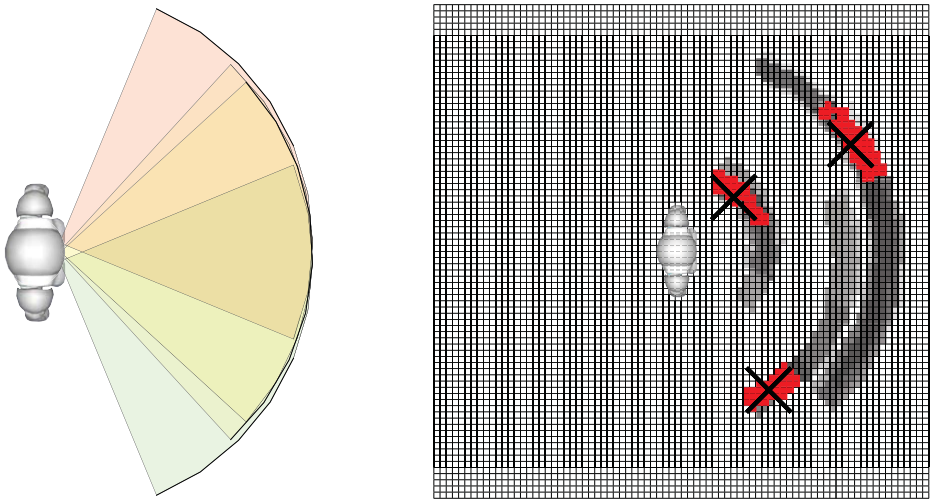
All obstacle locations within the model are constantly corrected using the odometry offset. This ensures that the obstacle locations are still roughly correct even if obstacles have not been seen for several seconds.

4.2.3.2 Ultrasonic Obstacle Detection

Since we were not very satisfied with the ultrasound-based obstacle detection system implemented for RoboCup 2012 [26], we reimplemented it for this year. We started by investigating how the different ultrasound firing modes that NAOqi offers really work, because the documentation is a bit unclear in this aspect[3]. The NAO is equipped with two ultrasound transmitters (one on each side) and two receivers (again one on both sides). It turned out that the firing modes 4–7 always fire both transmitters and read with both receivers. In the modes 4 and 7, the receivers measure the pulse sent by the transmitters on the same side. In the modes 5 and 6, they measure the pulse sent by the transmitters on the opposite side. The measurements do not seem to disturb each other. We also found out that only 70 ms after firing the sensors, the readings seem to be correct. Before that, they still might correspond to previous measurements. This suggests that the two ultrasound sensors are fired one after the other with a delay of about 50 ms. As a result, we toggle between the two firing 4 and 5. Thereby, all four possible combinations of measurements with two transmitters and two receivers are taken every 200 ms.

We model the areas covered by the four different measurement types as cones with an opening angle of 90° and an origin at the position between the transmitter and the receiver used. Since NAO's torso is upright in all situations in which we rely on ultrasound measurements, the whole modeling process is done in 2-D. We also limit the distances measured to 1.2 m, because experiments indicated that larger measurements sometimes result from the floor. The environment of the robot is modeled as an approximately 2.7 m × 2.7 m grid of 80 × 80 cells. The robot is always centered in the grid, i. e. the contents of the grid are shifted when the robot moves. Instead of rotating the grid, the rotation of the robot relative to the grid is maintained. The measuring cones are not only very wide, they also largely overlap. To exploit the information that, e. g., one sensor sees a certain obstacle, but another sensor with a partially overlapping measuring area does not, the cells in the grid are ring buffers that store the last 16 measurements that concerned each particular cell, i. e. whether a cell was measured as free or as occupied. With this approach, the state of a cell always reflects recent measurements. In contrast, when using a simple counter, updating a cell with the measurement of one sensor as occupied and with the measurement of another one as free would simply preserve its previous state and not reflect the fact that there is now a 50% chance of being an obstacle in that region. Experiments have shown

[3] `http://www.aldebaran-robotics.com/documentation/naoqi/sensors/dcm/pref_file_architecture.html#us-actuator-value`



(a) (b)

Figure 4.19: Ultrasound obstacle detection. a) Overlapping measuring cones. b) Obstacle grid that shows that sometimes even three robots can be distinguished. Clustered cells are shown in red, resulting obstacle positions are depicted as crosses.

that the best results are achieved when cells are considered as part of an obstacle if at least 10 of the last 16 measurements have measured them as occupied. All cells above the threshold are clustered. For each cluster, an estimated obstacle position is calculated as the average position of all cells weighted by the number of times each cell was measured as occupied.

Since the ultrasound sensors have a minimum distance that they can measure, we distinguish between two kinds of measurements: close measurements and normal measurements. Close measurements are entered into the grid by strengthening all obstacles in the measuring cone that are closer than the measured distance, i. e. cells that are already above the obstacle threshold are considered to have been measured again as occupied. This keeps obstacles in the grid that our robot approaches, assuming that closer obstacles are still measured with the minimum distance. It does not model other robots approaching our robot, but the whole system is not really suitable to correctly model moving robots anyway, at least not when they move quickly. For normal measurements, the area up to the measured distance is entered into the grid as free. For both kinds of measurements, an arc with a thickness of 100 mm is marked as occupied in the distance measured. If the sensor received additional echoes, the area up to the next measurement is again assumed to be free. This sometimes allows narrowing down the position of an obstacle on one side, even if a second, closer obstacle is detected on the other side as well – or even on the same side (cf. Fig. 4.19). Since the regular sensor updates only change cells that are currently in the measuring area of a sensor, an empty measurement is added to all cells of the grid every two seconds. Thereby, the robot forgets old obstacles. However, they stay long enough in the grid to allow the robot to surround them even when it cannot measure them anymore, because the sensors point away from them.

4.2.3.3 Arm Contact Recognition

Robots should detect whether they touch obstacles with their arms in order to improve close-range obstacle avoidance. When getting caught in an obstacle with an arm, there is a good chance that the robot gets turned around, causing its odometry data to get erroneous. This, in turn, affects the self-localization (cf. Sect. 4.2.1).

In order to improve the precision as well as the reactivity, the ArmContactModelProvider is executed within the *Motion* process. This enables the module to query the required joint angles with 100 frames per second. The difference of the intended and actual position of the shoulder joint per frame is calculated and also buffered over several frames. Thus, allowing the average error to be calculated. Each time this error exceeds a certain threshold, an arm contact is reported. Using this method, small errors caused by arm motions in combination with low stiffness can be smoothed. Hence, we are able to increase the detection accuracy. Due to the joint slackness, the arm may never reach certain positions. This might lead to prolonged erroneous measurements. Therefore a malfunction threshold has been introduced. Whenever an arm contact continues for longer than this threshold all further arm contacts will be ignored until no contact is measured.

The current implementation provides several features that are used to gather information while playing. For instance we are using the error value to determine in which direction an arm is being pushed. Thereby, the average error is converted into compass directions relative to the robot. Additionally, the ArmContactModelProvider keeps track of the time and duration of the current arm contact. This information may be used to improve the behavior.

In order to detect arm contacts, the first step of our implementation is to calculate the difference between the measured and commanded shoulder joint angles. Since we noticed that there is a small delay between receiving new measured joint positions and commanding them we do not compare the commanded and actual position of one shoulder joint from one frame. Instead we are using the commanded position from n frames[4] earlier as well as the newest measured position. Thus the result is more precise.

In our approach the joint position of one shoulder consists of two components: x for pitch and y for roll. Given the desired position p and the current position c, the divergence d is simply calculated as:

$$d.x = c.x - p.x$$
$$d.y = c.y - p.y$$

In order to overcome errors caused by fast arm movements, we added a bonus factor f that decreases the average error if the arm currently moves fast. The aim is to decrease the precision, i. e. increase the detection threshold for fast movements in order to prevent false positives. The influence of the factor f can be modified with the parameter `speedBasedErrorReduction` and is calculated as:

$$f = \max\left(0, 1 - \frac{|handSpeed|}{speedBasedErrorReduction}\right)$$

So for each arm, the divergence value d_a actually being used is:

$$d_a = d \cdot f$$

As mentioned above, the push direction is determined from the calculated error of an arm shoulder joint. This error has two signed components x and y denoting the joint's pitch and roll divergences. One component is only taken into account if its absolute value is greater than its contact threshold.

Table 4.1 shows how the signed components are converted into compass directions for the **right** arm. The compound directions NW, NE, SE, SW are constructed by simply combining the

[4] Currently, $n = 5$ delivers accurate results.

	x	y
is positive	N	E
is negative	S	W

Table 4.1: Converting signed error values into compass directions for the right shoulder and with $-y$ for the left

above rules if **both** components are greater than their thresholds, e. g. $x < 0 \wedge y < 0$ results into direction SE.

The push direction of each arm is used to add an obstacle to the robot's internal obstacle grid causing the robot to perform an evasion movement when it hits an obstacle with one of its arms. In addition, arm contacts might lower the robot's side confidence value as an arm contact with an obstacle might turn the robot.

This year, arm contacts are also used to move the arm out of the way to avoid further interference with the obstacle (cf. Sect. 6.2.7). Please note that while arm motions from the ArmMotionEngine are active, no arm contacts are detected for that arm.

For debugging and configuration purposes, a debug drawing (cf. Fig. 4.21) was added to visualize the error values gathered by the ArmContactModelProvider. It depicts the error vectors for each arm and can be activated by using the following commands in SimRobot:

```
vf arms
vfd arms module:ArmContactModelProvider:armContact
```

Figure 4.20: The left arm of the left robot is blocked and thus pushed back.

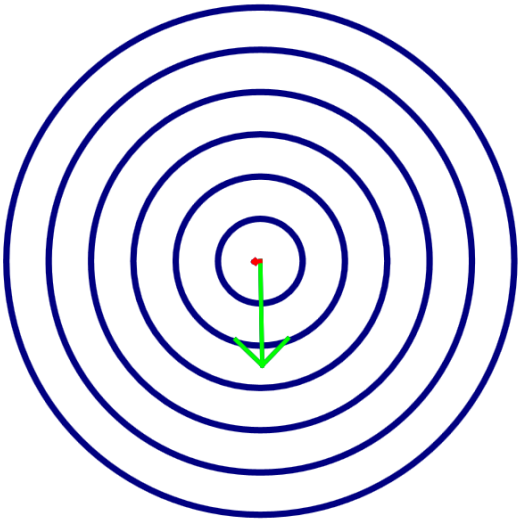


Figure 4.21: The green arrow denotes push direction and strength for the left arm.

Additionally, several plots for the single values that are calculated during contact detection are provided. They can be enabled by calling a script from SimRobot:

```
call ArmContactDebug
```

These plots are useful when parameterizing the ArmContactModelProvider.

The following values can be configured within the file *armContact.cfg*:

errorXThreshold & errorYThreshold: Maximum divergence of the arm angle (in degrees) that is not treated as an obstacle detection.

malfunctionThreshold: Duration of continuous contact (in frames) after which a contact is ignored.

frameDelay: Frame offset for calculating the arm divergence.

debugMode: Enables debug sounds.

speedBasedErrorReduction: At this translational hand speed, the angular error will be ignored (in mm/s).

waitAfterMovement: Number of milliseconds to wait after the ArmMotionEngine moved the arm before a new error measurement is integrated.

detectWhilePenalized: Whether arm contacts should also be detected when the robot is penalized.

4.2.4 Largest Free Part of the Opponent Goal

Because of various inaccuracies, the ball kicked by a NAO never rolls perfectly toward the target aimed at. Thus the robot should aim at the largest free part between the opponent's goal posts. This way, the probability of scoring a goal is maximized. For distributing the information, which part of the opponent goal is the largest free one, there is a representation *FreePartOfOpponentGoalModel*, which is provided by the module LineBasedFreePartOfOpponentGoalProvider.

The *FreePartOfOpponentGoalModel* contains two points defining the largest free part of the opponent goal, i.e. the part that is not blocked by any obstacle. An additional flag (`valid`) states, if there is a free part at all.

As the name of the module indicates, the decision, which part is free and which one is not, is based on the perceived lines provided by the LinePerceptor (cf. Sect. 4.1.7). For each line it is checked, whether it overlaps with the goal line. Thus, this module strongly depends on a good self-localization.

For determining the largest free part of the opponent goal, the module divides the goal line between the goal posts in twenty-eight cells. Each cell stores a value and a timestamp. The value represents the certainty, if this cell is in fact free. It is increased, when a perceived line overlaps with this cell and some time – defined by parameter `updateRate` – has passed since the last value update. However, the value cannot exceed a maximum value. It is decreased over time (parameter `ageRate`) providing that with its current position and orientation the robot should be able to see the cell. The value is also reset after a time-to-live timeout – defined by parameter `cellTTL`. The timestamp stores the frame timestamp when the value was last updated.

If the value exceeds a certain threshold (parameter `likelinessThresholdMin`), the cell is assumed to be free. Neighboring free cells are assembled to lines. The longest line is the result provided as the *FreePartOfOpponentGoalModel*.

The information provided in the *FreePartOfOpponentGoalModel* is currently used by the KickPoseProvider, which is described in Sect. 5.5.

4.2.5 Field Coverage

In the past, the reaction of our robots to losing knowledge about the ball position was to just turn round waiting for the ball to show up in the field of view again. If the latter did not happen, the robot would walk to a few different, fixed positions on the field, hoping to find the ball along his way. As the fourth robot was added to the team, we stopped doing this patrolling to fixed points. Instead, a searching robot now takes into account which parts of the field are actually visible and cooperates with the other robots during patrolling.

Since the introduction of the team-wide ball model (cf. Sect. 4.2.6), a robot only needs to actively search for the ball if all team members lost knowledge of the ball position. Consequently, if a robot is searching, it can be sure its team members do that too. Knowing which parts of the field are visible to the team members, patrolling can happen dynamically in a much more organized way.

4.2.5.1 Local Field Coverage

To keep track of which parts of the field are visible to a robot, the field is divided into a very coarse grid of cells, each cell being a square that has a size of $\frac{3}{4}m^2$. To determine which of the cells are currently visible, the current image is projected onto the field. Then all cells whose center lies within the projected image are candidates for being marked as visible.

There may be other robots obstructing the view to certain parts of the field. Depending on the point of view of the viewing robot, another robot may create an invisible area, a "shadow", on the field. No cell whose center lies within such a shadow is marked as visible. An example local field coverage is depicted in Fig. 4.22.

Having determined the set of visible cells, each of those cells gets a timestamp. These timestamps are later used to build the global field coverage model and to determine the least-recently-seen cell, which can be used to generate the head motion to scan the field while searching for the ball.

A special situation arises when the ball goes out. If that happened, the timestamps of the cells are reset to values depending on where the ball is being put back onto the field. That way the least-recently-seen cell of the grid – the cell which the robot has the most outdated information about – is now the cell in which the ball most likely is. This cell is determined by the last intersection of the trajectory of the ball with a field line before the GameController sent the information that the ball is out. Of course, this grid resetting can only work well if the ball motion was estimated accurately and the referees put the ball on the right position on the field, but without the reset, the information stored in the grid would not be useful anyway.

4.2.5.2 Global Field Coverage

To make use of the field coverage grids of the other robots, each robot has to communicate its grid to its team mates. Given this year's field dimensions, 4 byte timestamps, and a cell's edge length of $\frac{3}{4}m$, there are $4\ bytes * \frac{6m \cdot 9m}{(\frac{3}{4}m)^2} = 384\ bytes$ which have to be sent in every team communication cycle in addition to all other communication our robots do during gameplay. Since the resulting bandwidth requirement would be beyond the bandwidth cap set by this year's rules, the timestamps are 'compressed' to one byte and the grid is not sent as a whole but in intervals. For each cell c that is part of the interval which is to be sent to the other robots, a one byte coverage value $coverage(c)$ is computed, such that $time - (255 - coverage(c)) \cdot tick$ roughly matches the timestamp stored for cell c, with $time$ being the reference, i. e. the current,

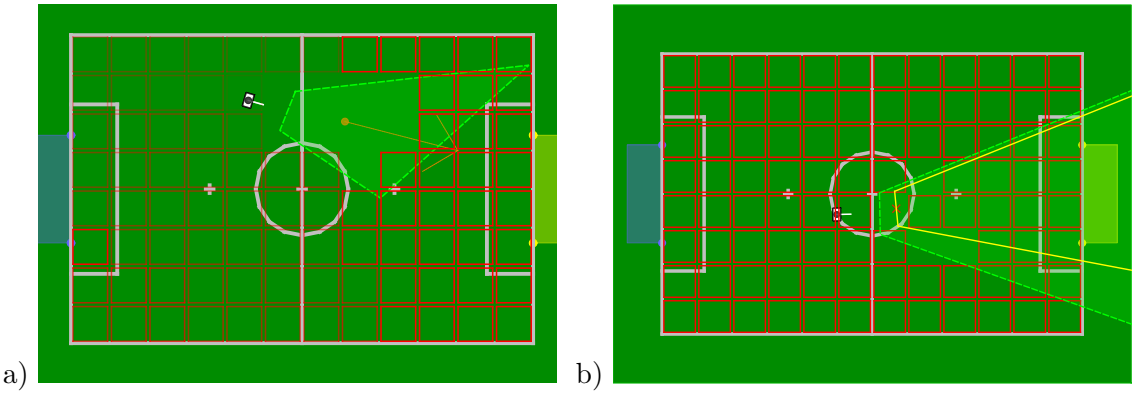


Figure 4.22: The local field coverage. a) The local grid after walking around on the field. The more intense the red borders of the cells are the less these cells are covered. The camera image projected onto the field is depicted by the dashed green line. b) An opponent robot prevents cells from being marked as visible. The yellow line marks the shadow the opponent robot casts.

timestamp. Setting $tick = 300ms$ leads to coverage values which range up to $76.5seconds$ into the past. With $n = 3$ being the number of intervals the grid is divided into, the field coverage only adds $\frac{96}{3} + 4\ bytes = 36\ bytes$ to the team communication. The 4 extra bytes are for the reference timestamp to regain the original timestamps of each cell.

So in addition to its own local field coverage grid, each robot maintains the field coverage grids of its team mates, which are incrementally updated in every team communication cycle. All these grids have to be merged into one global grid which looks roughly the same for all team mates, so that calculations based on the grid come to sufficiently similar results for all team mates.

The merging is done by simply taking the maximum known coverage value for each cell, so that the value for the cell g_i is $g_i = \max(r(1)_i, \ldots, r(\#robots)_i)$, where $r(k)$ is the grid received from the robot with number k.

Based on the values in the global field coverage grid, it has to be decided which parts of the field are covered by the robots and which parts are not, i. e. which parts are unknown to the whole team. The obvious way to do this is a threshold value, which marks the minimum coverage value a cell can have and still be considered to be covered, i. e. 'known'. The only catch is that this threshold value has to be determined dynamically. With a fixed threshold we could easily end up with the entire field being considered uncovered or covered, although there were cells on the field which were covered so much worse than others, so that it would be desirable that the team 're-covers' those cells first and leave the rest of the field as it is. The situation is analogous to determining which parts of a gray scale image are black or white, so we applied the Otsu algorithm [19] to compute the threshold.

The idea is as follows: The ideal situation to separate the coverage values into *covered* and *uncovered* would be the coverage grid containing only two different coverage values v_{low} and v_{high}, so we could just choose the coverage threshold $t = \frac{v_{low}+v_{high}}{2}$ to be in between them. This situation won't occur very often, so we turn the threshold finding procedure upside-down and calculate for every possible threshold $0 \leq t \leq 255$ how well the resulting ideal situation model with only two coverage values fits the coverage grid, if we choose optimal v_{low} and v_{high}.

To do this, we need an error function which determines how well a model fits the actual grid.

$$e(v_{low}, v_{high}) = \sum_{c \in Grid} \min\left((v(c) - v_{low})^2, (v(c) - v_{high})^2 \right) \tag{4.1}$$

Instead of summing over all cells in the grid we build a histogram h of all coverage values, so we can get rid of min and rewrite eq. 4.1 as

$$e(v_{low}, v_{high}) = \underbrace{\sum_{v=0}^{t} h(v) \, (v - v_{low})^2}_{e_{v_{low}}} + \underbrace{\sum_{v=t+1}^{v_{max}} h(v) \, (v - v_{high})^2}_{e_{v_{high}}} \tag{4.2}$$

In eq. 4.2, t is the threshold as defined above, v is a coverage **v**alue and $v_{max} = maxCoverage$ is the maximum coverage value. Now, for a given t, we can find the optimal values for v_{low} and v_{high} by minimizing both $e_{v_{low}}$ and $e_{v_{high}}$. By taking the derivatives and solving for v_{low} and v_{high} respectively, the optimal values turn out to be the average coverage value of all coverage values below t for v_{low} and above t for v_{high}:

$$v_{low} = \frac{\sum_{v=0}^{t} h(v) \cdot v}{\sum_{v=0}^{t} h(v)} \quad v_{high} = \frac{\sum_{v=t+1}^{v_{max}} h(v) \cdot v}{\sum_{v=t+1}^{v_{max}} h(v)} \tag{4.3}$$

By substituting eq. 4.3 into eq. 4.2, one gets an error function which only depends on the choice of the threshold t. After some simplification, this is:

$$e(t) = \sum_{v=0}^{v_{max}} h(v) v^2 - \frac{\left(\sum_{v=0}^{t} h(v) v\right)^2}{\sum_{v=0}^{t} h(v)} - \frac{\left(\sum_{v=t+1}^{v_{max}} h(v) v\right)^2}{\sum_{v=t+1}^{v_{max}} h(v)} \tag{4.4}$$

With eq. 4.4, we calculate the threshold by just trying every possible threshold and then using the threshold with the minimal error.

After having determined which cells are the uncovered ones, each cell as to be assigned to a robot which looks at it. This is done using k-means clustering. k is set to be the number of robots which are able to cover a certain part of the field. To be included, a robot must be standing on its own feet, must be reasonably confident in its self localization, etc. The clusters are initialized with the current positions of the robots and each uncovered cell is assigned to its closest cluster. After that, the new cluster means are computed based on the center positions of the clusters' cells. This process is repeated until the assignments do not change anymore.

Using four-way flood fill on each cell of each cluster, the connected components of each cluster are computed and the largest connected component of each cluster is retained. So for each robot currently able to patrol, there is now one connected component of uncovered cells. Now for each robot, the geometric center of the component is calculated and used as a patrol target for that robot, such that each robot knows the patrol targets of all robots of the team, including its own. Fig. 4.23 shows one example of a global field coverage grid and the corresponding end result.

As the robots move over the field, the patrol targets change with the field coverage, such that each robot which can patrol has its patrol target in its closest, largest region of uncovered cells. Notice that besides leading to meaningful patrol targets, this procedure also has the nice property that the ways of different robots to their patrol target do not cross.

4.2.6 Combined World Model

Unlike some other domains, such as the Small Size League, the robots in the SPL do not have a common and consistent model of the world, but each of them has an individual world

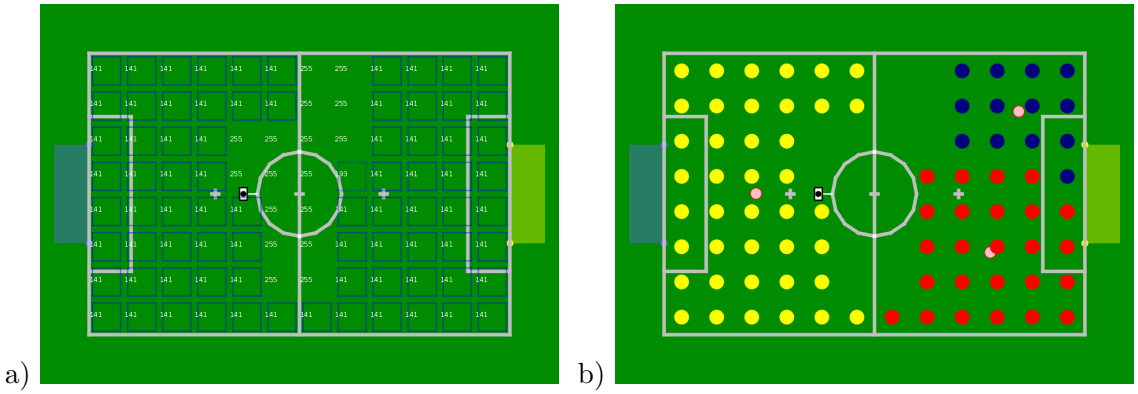


Figure 4.23: The global field coverage. a) The global grid obtained by merging the communicated local grids. b) The largest connected components of each cluster of uncovered cells. Each cell of such a component is marked by a yellow, blue or red circle to indicate the cluster assignment. The pink circles are the resulting patrol targets.

model estimated on the basis of its own limited perception. As such a model is a necessity for creating cooperative behavior, we implemented a combined world model (representation *CombinedWorldModel*) that lets all robots of a team have an assumption of the current state of the world even if parts of it were not seen by the robot itself. This assumption is consistent among the team of robots (aside from delays in the team communication) and consists of three different parts (*global ball model*, *positions of the teammates*, and *positions of opponent players*) that are described below.

An example is given in Fig. 4.24–4.26 in which a playing situation and the corresponding local and combined world models of the robot on the right side are shown.

4.2.6.1 Global Ball Model

The global ball model is calculated locally by each robot, but takes the ball models of all teammates into account. This means that the robot first collects the last valid ball model of each teammate, which is in general the last received one, except for the case that the teammate is not able to play, for instance because it is penalized or fallen down. In this case, the last valid ball model is used. The only situation in which a teammate's ball model is not used at all is if the ball was seen outside the field, which is considered as a false perception. After the collection of the ball models, they are combined in a weighted sum calculation to get the global ball model. There are four factors that are considered in the calculation of the weighted sum:

- The approximated validity of the self-localization: the higher the validity, the higher the weight.
- The time since the ball was last seen: the higher the time, the less the weight.
- The time since the ball *should* have been seen, i. e. the time since the ball was not seen although it should have appeared in the robot's camera image: the higher the time, the less the weight.
- The approximated deviation of the ball based on the bearing: the higher the deviation, the less the weight.

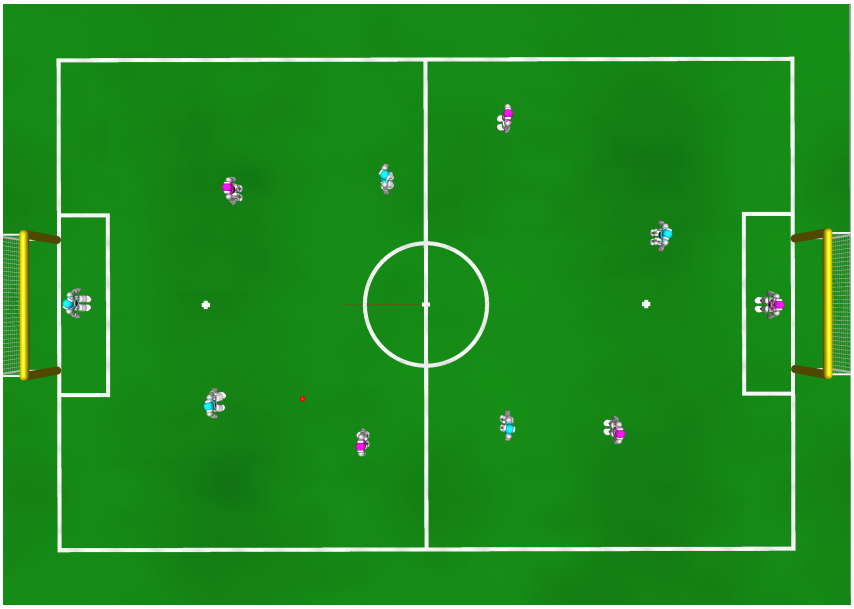


Figure 4.24: Overview of a playing situation where opponent robots are shown in red. The ball is located in the lower left quadrant.

Based on these factors, a common ball model, containing an approximated position and velocity, is calculated and provided by the representation *CombinedWorldModel* as `ballState`. Additionally, another global ball model is calculated using the same algorithm but only considering team mates that have a higher side confidence (cf. Sect. 4.2.1) than the locally calculating robot and that provide a valid ball position, i. e. a ball position within the field borders. This one is stored in the *CombinedWorldModel* as `ballStateOthers`.

Among other things, the global ball model is currently used to make individual robots hesitate to start searching for the ball if they currently do not see it but their teammates agree about its position.

4.2.6.2 Positions of Teammates

For the coordination of the team, for instance for the role selection or execution of tactics, it is important to know the current positions of all teammates. Computing those positions does not require special calculations such as filtering or clustering, because each robot sends its filtered position to the teammates via team communication. This means that each robot is able to get an accurate assumption of all positions of its teammates by listening to the team communication. Besides to the coordination of the team, the positions of the teammates are of particular importance for distinguishing whether perceived obstacles belong to own or opponent players (cf. Sect. 4.2.6.3).

4.2.6.3 Positions of Opponent Players

In contrast to the positions of the teammates, it is more difficult to estimate the positions of the opponent players. The opponent robots need to be recognized by vision and ultrasound and a position as accurate as possible needs to be calculated from it.

Compared to only using the local model of robots perceived by vision or ultrasound, which is solely based on local measurements and may differ for different teammates, it is more difficult

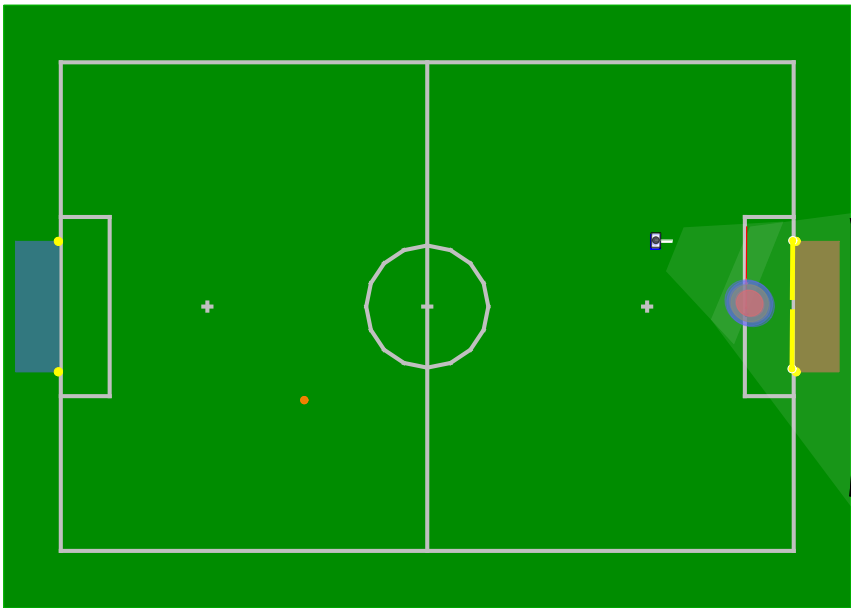


Figure 4.25: The local world model of the robot on the right side. The robot is only able to see one opponent robot. All other opponent robots and the ball could not be seen.

to get a consistent model among the complete team based on all local models. For a "global" model of opponent robot positions, which is intended here, it is necessary to merge the models of all teammates to get accurate positions. The merging consists of two steps, namely clustering of the positions of all local models and reducing each cluster to a single position.

For the clustering of the individual positions an ε-neighborhood is used, which means that all

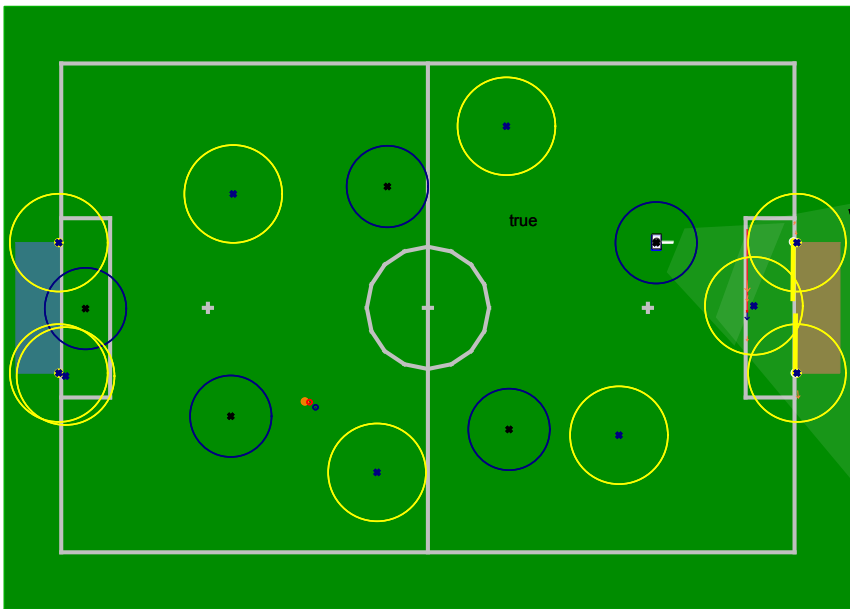


Figure 4.26: The combined world model for the robot on the right side. The blue circles show the own teammates, the yellow circles show obstacles including goal posts and possible opponent players, the small blue circle shows the global ball including the robot's own belief of the ball position, and the small red circle shows the global ball without the robot's own belief of the ball position.

positions that have a smaller distance to each other than a given value are put into a cluster. It is important for the clustering that only positions are used that do not point to a teammate, i. e. all obstacles that are located near a teammate are excluded from the merging process.

After the clustering is finished, the positions of each cluster that represents a single opponent robot have to be reduced to a single position. For the reduction the covariances of the measurements are merged using the *measurement update* step of a Kalman filter. The result of the whole merging procedure is a set of positions that represent the estimated positions of the opponent robots. Also, the four goal posts are regarded as opponent robots because they are obstacles that are not teammates.

Chapter 5

Behavior Control

The part of the B-Human system that performs the action selection is called *Behavior Control*. The behavior is modeled using the C-based Agent Behavior Specification Language (CABSL). The main module – BehaviorControl2013 – provides the representations *BehaviorControlOutput*, *ArmMotionRequest*, *MotionRequest*, *HeadMotionRequest*, *SoundRequest*, *ActivationGraph*, and *BehaviorLEDRequest*. It is accompanied by the KickPoseProvider (cf. Sect. 5.5), the CameraControlEngine (cf. Sect. 5.6) and the LEDHandler (cf. Sect. 5.7) as well as the RoleProvider (cf. Sect. 5.3.1).

This chapter begins with a short overview of CABSL and how it is used in a simple way. Afterwards, it is shown how to set up a new behavior. Both issues are clarified by examples. The major part of this chapter is a detailed explanation of the full soccer behavior used by B-Human at RoboCup 2013.

5.1 CABSL

CABSL is a language that has been designed to describe an agent's behavior as a hierarchy of state machines and is a derivative of the *State Machine Behavior Engine* (cf. [26, Chap. 5]) we used last year. CABSL solely consists of C++ preprocessor macros and, therefore, can be compiled with a normal C++ compiler.

For using it, it is important to understand its general structure. In CABSL, the following base elements are used: *options, states, transitions, actions*. A behavior consists of a set of options that are arranged in an option graph. There is a single starting option from which all other options are called; this is the root of the option graph. Each option is a finite state machine that describes a specific part of the behavior such as a skill or a head motion of the robot, or it combines such basic features. Each option starts with its *intital_state*. Inside a state, an *action* can be executed which may call another option as well as execute any C++ code, e. g. modifying the representations provided by the *Behavior Control*. Further, there is a *transition* part inside each state, where a decision about a transition to another state (within the option) can be made. Like *action*s, *transition*s are capable of executing C++ code. In addition to *options* (cf. Sect. 5.1.2) CABSL supports the use of so-called *Libraries* (cf. Sect. 5.1.2), which may be used inside *action*s and *transition*s. Libraries are collections of functions, variables, and constants, which help keeping the code clean and readable without the need of additional modules and representations, while providing a maximum of freedom if extensive calculations have to be made.

This structure is clarified in the following subsections with several examples.

5.1.1 Options

```
option(exampleOption)
{
  initial_state(firstState)
  {
    transition
    {
      if(booleanExpression)
        goto secondState;
      else if(libExample.boolFunction())
        goto thirdState;
    }
    action
    {
      providedRepresentation.value = requiredRepresentation.value * 3;
    }
  }

  state(secondState)
  {
    action
    {
      SecondOption();
    }
  }

  state(thirdState)
  {
    transition
    {
      if(booleanExpression)
        goto firstState;
    }
    action
    {
      providedRepresentation.value = RequiredRepresentation::someEnumValue;
      ThirdOption();
    }
  }
}
```

Special elements within an option are common transitions as well as target states and aborted states.

Common transitions consist of conditions that are checked all the time, independent from the current state. They are defined at the beginning of an option. Transitions within states are only “else-branches” of the common transition, because they are only evaluated if no common transition is satisfied.

Target states and aborted states behave like normal states, except that a calling option may check whether the called option currently executes a target state or an aborted state. This can come in handy if a calling option should wait for the called option to finish before transitioning to another state. This can be done by using the special symbols `action_done` and `action_aborted`.

Note that if two or more options are called in the same action block, it is only possible to check whether the option called last reached a special state.

```
option(exampleCommonTransitionSpecialStates)
{
  common_transition
```

```
  {
    if(booleanExpression)
      goto firstState;
    else if(booleanExpression)
      goto secondState;
  }

  initial_state(firstState)
  {
    transition
    {
      if(booleanExpression)
        goto secondState;
    }
    action
    {
      providedRepresentation.value = requiredRepresentation.value * 3;
    }
  }

  state(secondState)
  {
    transition
    {
      if(action_done || action_aborted)
        goto firstState;
    }
    action
    {
      SecondOption();
    }
  }
}

option(SecondOption)
{
  initial_state(firstState)
  {
    transition
    {
      if(boolean_expression)
        goto targetState;
      else
        goto abortedState;
    }
  }

  target_state(targetState)
  {
  }

  aborted_state(abortedState)
  {
  }
}
```

In addition, options can have parameters that can be used like normal function parameters.

```
option(firstOption)
{
  initial_state(firstState)
  {
```

```
      action
      {
        OptionWithParameters(5, true)
      }
    }
  }

  option(OptionWithParameters, int i, bool b, int j = 0)
  {
    initial_state(firstState)
    {
      action
      {
        providedRepresentation.intValue = b ? i : j;
      }
    }
  }
```

5.1.2 Libraries

A library is a normal C++ class, a single object of which is instantiated as part of the behavior control and that is accessible by all options. In contrast to options, libraries can have variables that keep their values beyond a single execution cycle. Libraries must be derived from the class `LibraryBase` and an instance variable with their type must be added to the file *Libraries.h*. In addition, if you want your library to be useable by other libraries, it has to be added to the files *LibraryBase.h* and *LibraryBase.cpp*. Please refer to the example behavior contained in this code release on how to do this.

Here is an example for a header file that declares a library:

```
class LibExample : public LibraryBase
{
public:
  LibExample();
  void preProcess() override;
  void postProcess() override;
  bool boolFunction(); // Sample method
};
```

Here is the matching implementation file:

```
#include "../LibraryBase.h"

namespace Behavior2013
{
  #include "LibExample.h"
  #include "AnotherLib.h"

  LibExample::LibExample()
  {
    // Init library (optional)
  }

  void LibExample::preProcess()
  {
    // Called each cycle before the first option is executed (optional)
  }

  void LibExample::postProcess()
  {
    // Called each cycle after the last option was executed (optional)
```

```
  }

  bool LibExample::boolFunction()
  {
    return true;
  }
}
```

5.2 Setting Up a New Behavior

This report comes with a basic behavior but you might want to set up your own one with a new name. To do so, you simply need to copy the existing behavior in *Src/Modules/BehaviorControl/BehaviorControl2013* into a new folder *Src/Modules/BehaviorControl/<NewBehavior>* and subsequently replace all references to the old name within the cloned folder. This basically means to rename the module BehaviorControl2013 to your desired name (this includes renaming the configuration file *Config/Locations/Default/behavior2013.cfg*) and to update the namespace definitions of the source files within *<NewBehavior>/*.cpp* and *<NewBehavior>/*.h*. When renaming the namespace and the module, you may not choose the same name for both.

5.3 Behavior Used at RoboCup 2013

The behavior of 2013 is split in two parts, "BodyControl" and "HeadControl". The "BodyControl" part is the main part of the behavior and is used to handle all game situations by making decisions and calling the respective options. It is also responsible for setting the *HeadControlMode* which is then used by the "HeadControl" to move the head as described in cf. Sect. 5.3.4. Both parts are called in the behaviors main option named *Soccer* which also handles the initial stand up of the robot if the chestbutton is pressed as well the sit down if the chestbutton is pressed thrice.

The "BodyControl" part starts with an option named *HandlePenaltyState* which makes sure that the robot stands still in case of a penalty and listens to chestbutton presses to penalize and unpenalize the robot manually. If the robot is not penalized the option *HandleGameState* is called, which in turn will make sure that the robot stands still if one of the gamestates *inital*, *set*, or *finished* is active. In Ready state it will call the option *ReadyState* which lets the robots walk to their kickoff positions.

When in Playing state, an option named *PlayingState* will be called which handles different game situations. First it handles kickoff situations which are described in detail in its own section (cf. Sect. 5.3.3). In addition it checks whether a special situation, where all robots have to behave the same, has to be handled before invoking the role selection (cf. Sect. 5.3.1.2) where each robot gets chosen for one of the different roles (cf. Sect. 5.3.2).

The special situations are:

StopBall: If the ball was kicked in the direction of the robot, it tries to catch it by getting in its way and execute a special action to stop it.

TakePass: If the striker decides to pass to that robot, it turns in the direction of the ball and behaves as described above.

SearchForBall: If the whole team has not seen the ball for a while, the field players start turning around to find it. While turning, the robot aligns its head to the positions pro-

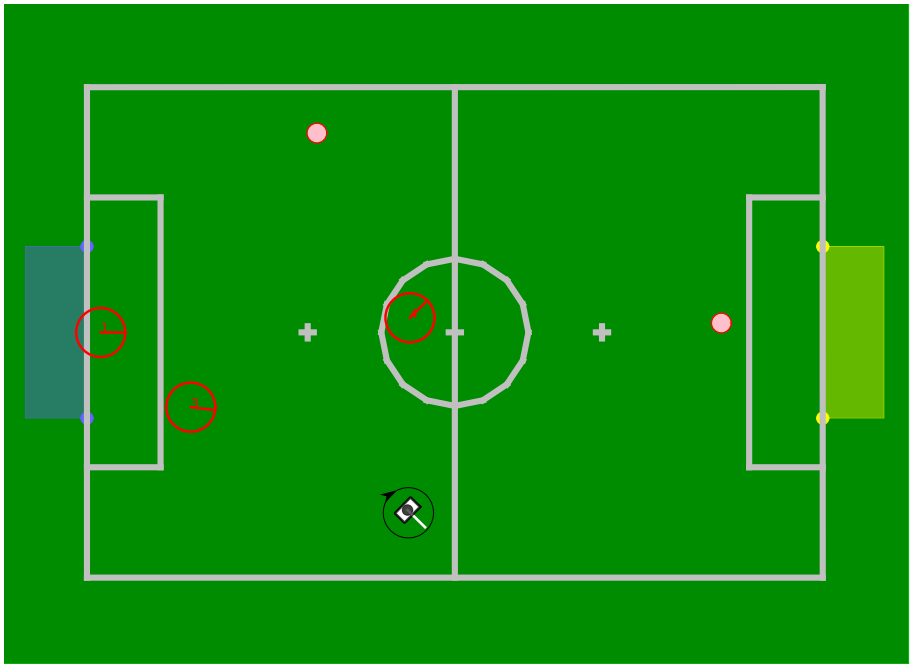


Figure 5.1: Visualization of the search-for-ball procedure. A robot first spins around and then walks to its patrol target (pink circles). The patrol targets change depending on the number of searching robots and on their positions (red circles).

vided by the FieldCoverageProvider (cf. Sect. 4.2.5.1). If, after a whole turn, no ball was observed, the robot patrols to a position provided by the GlobalFieldCoverageProvider (cf. Sect. 4.2.5.2). Figure 5.1 visualizes this procedure.

SearchForBallOnOut: If the ball went out, all robots try to find it using the procedure described above.

CheckMirroredPosition: If the robot uses its teammates' information to find the ball, but cannot see it for a few seconds, it starts to turn around to look at the mirrored position. This is done in case a robot's pose has flipped sides. If it still cannot find the ball after about thirty seconds, it starts to run to the center of the field to get a better look at the whole field. The section about self-localization (cf. Sect. 4.2.1) contains detailed information about how seeing the ball helps to flip the robot back.

CheckBallAfterStandup: If the robot falls down close to the center circle and cannot see the ball after standing up, it will also turn and look at the mirrored ball position to reduce the chance of flipping sides.

5.3.1 Roles and Tactic

5.3.1.1 Tactic

At RoboCup 2013, B-Human used the same tactic for all matches. Thus, each match was played with a keeper, a defender, a supporter, a breaking supporter, and a striker. In this tactic, the striker always walks towards the ball if possible (cf. Sect. 5.3.2.1), the supporter walks next to the striker with a defined offset (cf. Sect. 5.3.2.2), the breaking supporter takes position in the opponent half (cf. Sect. 5.3.2.3), the defender stands in front of the own penalty area placed

sidewards from the keeper (cf. Sect. 5.3.2.4), and the keeper stays inside its penalty area (cf. Sect. 5.3.2.5).

5.3.1.2 Role Selection

The dynamic role selection is made in the module RoleProvider. In general, the role selection is dependent on the current tactic (cf. Sect. 5.3.1.1), their current roles and the current game situation. The role selection only takes place during the game state *PLAYING*. As the robot with the number one is the only player available for the keeper role, the RoleProvider statically assigns this role. All other roles are assigned dynamically.

To assign roles, the RoleProvider needs a set of "rules" that are specified in a configuration file. These rules contain information about whether a robot is applicable for a given role. There is a different set of rules for different tactics, and while we always used the same overall tactic, it is of course possible that one or more robots are removed from the game in case of a penalty or defect. So we use a different tactic for each number of robots available, where the tactics only differ in the roles that are or are not assigned.

In order to decide which robot becomes striker, each robot computes its estimated time to reach the ball. This time depends on the characteristics *distance and angle to the ball*, *angle to the line between ball and opponent goal*, and *time since the ball was last seen*. Furthermore, if the robot is blocked by an obstacle, the time will be increased slightly. This time is sent via team communication (cf. Sect. 3.5.4) to the other connected robots. Each robot compares its own time with the shortest time of the other connected robots on the field. The one with the shortest time becomes the striker.

All other support roles (defender, supporter, breaking supporter) are assigned depending on their position on the field, where for example the defending role mostly gets assigned to the robot with the smallest x position, i. e. the robot closest to its goal. The robot positions are also sent via team communication to the other connected robots, so that each robot can make informed decisions on its role selection.

To prevent an oscillation of the roles, the current role of each robot is also sent to each other robot. If the rules state that a robot is applicable for a given role, the RoleProvider checks whether any other robot currently holds this role. If this is the case, a flag will be set that announces that this robot wants to change its role. Now to switch roles, the other robot has to check the rules to see that it is applicable for the role of the first robot. Since the "change role flag" is set, it can take the role of that robot. Now that one of the roles is not assigned, the first robot will choose that role. In addition to that, the current striker always has a small bonus to its time to reach the ball so that no fast role switches occur if the distance of the ball is nearly the same for two robots.

The only exception to this procedure is if the time to reach the ball of one robot is far less than that of the current striker. In this case, the robot instantly chooses the striker role so that there is no delay to prevent the robot from moving to the ball and kicking it as fast as possible.

It is also possible to set fixed roles for each robot for testing purposes.

5.3.2 Different Roles

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

5.3.2.1 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:

GoToBallAndKick: Whenever no special situation is active (such as kick-off, duel, search for the ball ...), the striker walks towards the ball and tries to score. This is the main state of the striker and after any other state, which was activated, has reached its target, it returns to *goToBallAndKick*. To reach a good position near the ball, the *KickPose* (cf. Sect. 5.5) is used and the striker walks to the pose provided. Depending on the distance to the target, the coordinates are then either passed to the PathPlanner cf. Sect. 5.4 to find the optimal way to a target far away or to the LibWalk when it comes to close range positioning and obstacle avoidance. Each strategy then provides new coordinates, which are passed to the WalkingEngine (cf. Sect. 6.2.1).

After reaching the pose, the striker either executes the kick that was selected by the *KickPoseProvider* (cf. Sect. 5.5) or chooses another state in case of an opponent robot blocking the way to the goal.

Duel: Whenever there is a close obstacle between the ball and the goal, the robot will choose an appropriate action to avoid that obstacle. Depending on the position of the robot, the ball, and the obstacle, there are several actions to choose from (cf. Fig. 5.2):

- The robot is near the left or right sideline.
  - If the robot is oriented to the opponent goal and the straight way is not blocked by an obstacle, it will perform a fast forward kick.
  - If the direct way to the goal is blocked or the robot is oriented away from the opponent goal, the robot will perform a sidewards kick to the middle of the field.
- The robot is in the middle of the field.
  - If the obstacle is blocking the direct path to the goal, the robot will perform a sidewards kick. The direction of the kick depends on the closest teammate and other obstacles. In an optimal situation, a supporter should stand besides the striker to receive the ball.
  - If the obstacle leaves room for a direct kick towards the opponent goal, the robot will perform a fast forward kick.

The kick leg is always chosen by the smallest distance from the leg to the ball. If the robot has not seen the ball for more than three seconds while in duel mode, it will move backwards for a short time to reduce the probability of getting punished for pushing in case the ball is no longer close to the robot. Duel is deactivated in the vicinity of the opponent's penalty box, because we do not want the robot to kick the ball away from the goal if there is an obstacle (i. e. the keeper or a goalpost) in front of it.

DribbleDuel: This is a special kind of duel behavior. It is chosen if there is an opponent between the ball and the goal and the robot stands in front of the ball facing the side of the field. In that case, the robot just takes its arms back to avoid contact with the opponent robot and starts to run to the ball to quickly dribble it away from the opponent's feet.

DribbleBeforeGoal: If the ball is lying near the opponent ground line, so that it is unlikely to score a goal, the striker aligns behind the ball and slowly moves in the direction of the opponent's penalty mark. If the robot drifts too far away from its target it starts again to align. If it is again possible to score directly the state *goToBallAndKick* is executed.

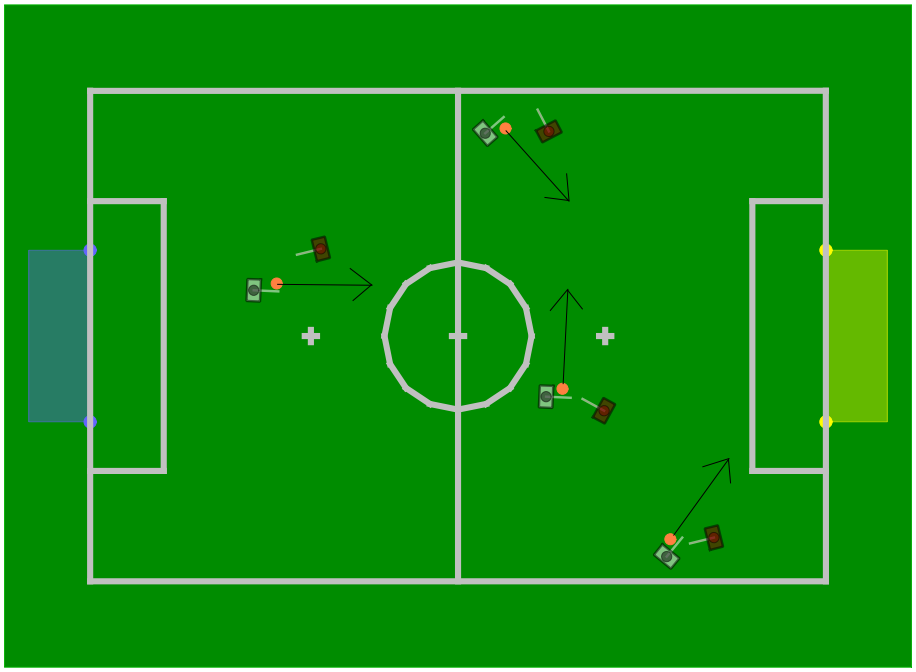


Figure 5.2: A set of possible tackling situations. The black lines mark the desired kick direction.

WalkNextToKeeper: If the ball is inside the own penalty area the robot will try to block the way to the ball so that opponents cannot reach it. If the keeper is ready to kick the ball out of the penalty area, the robot will move out of the way.

5.3.2.2 Supporter

The main task of the supporter is, as the name indicates, to support the striker. Most of the time, when the game state is *PLAYING*, the supporter tries to get a good position to support the striker. This is done by positioning the robot next to the striker with a defined offset. It is important that the supporter is able to see the ball from time to time. Therefore, it is aligned to an angle between the ball and the opponent goal or just to the ball if the supporter is in front of the ball. The supporter also always tries to position itself in a way, that the striker's (cf. Sect. 5.3.2.1) y-position is between its own y-position and the breaking supporter's (cf. Sect. 5.3.2.3) y-position, thereby forming a triangle. Of course this is not possible if the striker stands close to the side line. In addition the supporter will take position near the middle line if the striker moves far into the opponent half. This is done to make sure that we can get to the ball quickly if the striker loses the ball. The supporter is positioned approximately 1.1 m to the side and between 0.2 m and 4 m to the back relative to the striker, depending on the x-position of striker.

Besides positioning, several special operations exist. The *search for the ball* behavior mentioned above (cf. Sect. 5.3.1) 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.

The positioning of the supporter is realized by a potential field [11] as depicted in Fig. 5.3. 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

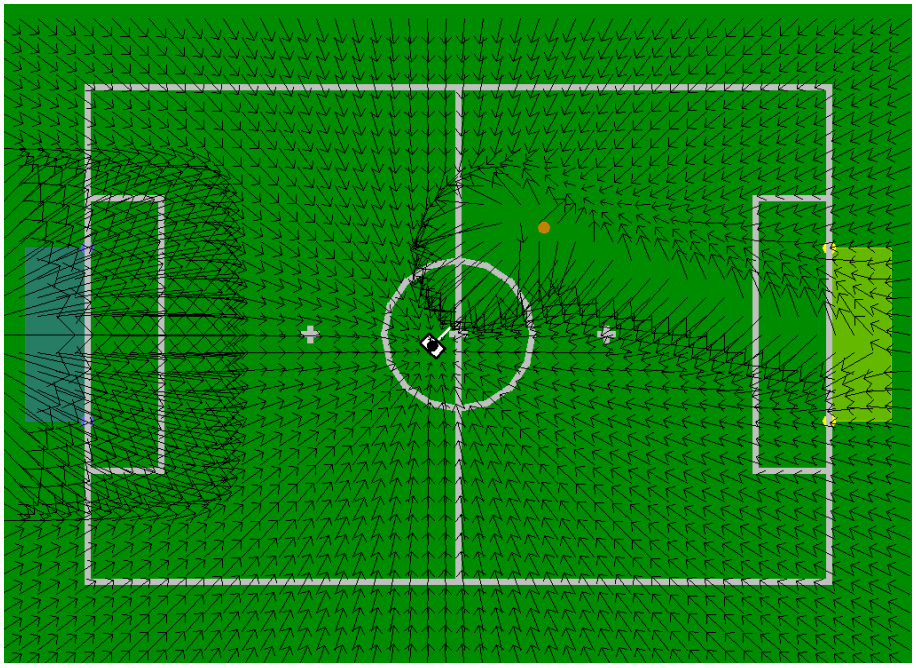


Figure 5.3: Visualization of the potential field for positioning the supporter. The components that are combined can be clearly seen: the rejection of the own penalty area, the attraction of the target position, the circular rejection of the striker with a bias to the own goal, and the rejection of the line between ball and opponent goal.

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 other supporter: The two supporters should cover as much field as possible which lays behind the striker. This means that if both supporters are near to each other, there is a huge part of the field free for other players. This should be avoided.

Rejection of the striker-ball-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 striker-ball-line. This ensures that the striker does not accidentally push the supporter when trying to reach the ball.

Rejection of the ball-goal-line: The supporter 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.

5.3.2.3 Breaking Supporter

The main task of the breaking supporter is to position itself in the opponent half, to become striker (cf. Sect. 5.3.2.1) quickly if the ball is passed to it or kicked in the direction of the opponent goal, and to score a fast goal while the most of its teammates and opponents are still in its own half. The breaking supporter tries to stand on an imaginary line between the corner of the opponent penalty area and the middle line as seen in Fig. 5.4. Its x-position depends on the position of the ball. Like the supporter (cf. Sect. 5.3.2.2) tries to not stand on the same side of the striker as the breaking supporter, the breaking supporter tries to not stand on the same side of the striker as the supporter.

The positioning of the robot is realized with the potential field that is also used for positioning the supporter. Also the obstacle avoidance is completely the same as for the supporter.

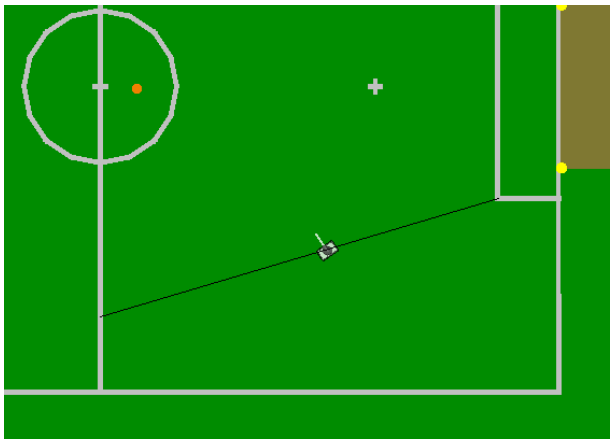


Figure 5.4: The positioning of the breaking supporter. The breaking supporter moves on the shown line while trying to keep an eye on the ball.

5.3.2.4 Defender

The defender's task is, in general, to defend the own goal without entering the own penalty area. 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. 5.5). The positioning of the robot is realized with the potential field that is also used for positioning the supporter (cf. Sect. 5.3.2.2). Also the obstacle avoidance is completely the same as for the supporter.

In the case that the keeper is off the field or attacking, this role is necessary to still have a player on the field that defends the own goal. In the case that the keeper stays in its goal, this role is used to increase the defense capabilities of the own team. The positioning of the defender differs

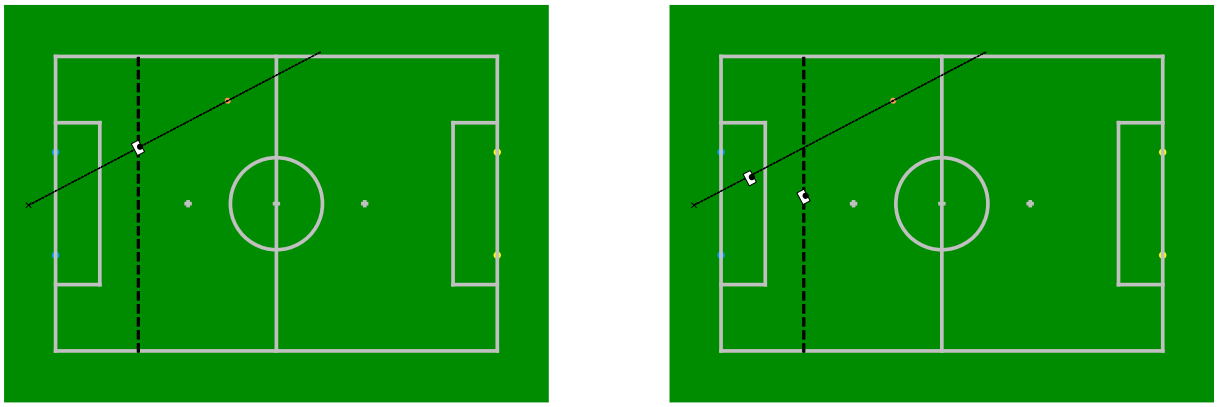


Figure 5.5: The positioning of the defender. The first picture shows the centered position if no keeper is on the field, and the second depicts the defender placed sidewards to the keeper.

for the case that the keeper stays in its goal in that the robot is displaced approx. 50 cm to the side to let the keeper have the ball in sight all the time.

5.3.2.5 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 field, the keeper walks directly towards its own goal. Certainly, the time to reach the own goal should take as little time as possible. To achieve this, the robot walks straight forward to the own goal as long as it is still far away and starts walking omni-directionally when it comes closer to the target to reach the desired orientation. This speed-up compensates for the disadvantage that the robot is probably directed away from the ball. When walking omni-directionally, the keeper avoids obstacles in the same way as the striker and the defender do.

The second situation, when the robot leaves the own goal, is if the ball is close to the goal and has stopped moving. In this case, the robot walks towards the ball in order to kick it away. The keeper uses the same kicking method as the striker with the difference that obstacle avoidance is deactivated. Since there is a special handling for pushing a goalie in its own penalty box, this solution is acceptable.

In any other situation, the robot will not leave its position within the own penalty area and executes different actions according to different situations. Normally, the keeper walks to positions computed by the GoaliePoseProvider module described below. After reaching the desired pose, the goalie will, in contrast to last year's behavior, no longer sit down. Thus, it can observe as much field as possible. So while guarding the goal, the keeper will switch its head control depending on how long the ball was not seen and in which half of the field the ball was seen last. For example, while the ball is in the opponent half, the keeper watches for important landmarks to improve its localization. But once the ball passes the middle line, the keeper invariably tracks the ball.

If the ball is rolling towards the goal, the goalie has to select whether it should spread its legs or dive to catch the ball. The decision is made by considering the velocity of the ball as well as its distance. 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. 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 2013). 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. The decision whether to execute a cover motion is consolidated over multiple frames (about a third of a second in total) to compensate for falsely positive ball estimates and to avoid an unnecessary dive.

Sometimes, after getting up after a diving motion, the self-localization is incorrect, i. e. more precisely the robot's rotation is wrong by 180°. This situation is resolved by the self-locator using the *FieldBoundary*. Additionally, getting no percepts for a certain time is a strong indicator for the rotation to be erroneous. In this case the goalie starts turning around by 180 degrees hoping to gather some new perceptions which can correct its localization.

In case the keeper has not seen the ball for several seconds (the actual value is currently set to 7s) it will, under some further circumstances, initiate a ball searching routine. This is only done if its absolute position on the field is considered to be within the own penalty area and the ball position communicated by the teammates is considered invalid. The first condition grants that the keeper does not start searching for the ball while it is walking back to its goal after returning from a penalty. If the ball position communicated by the teammates is considered valid, the keeper may be able to find the ball by turning towards (either the whole body or only its head) the communicated position. If it is not valid, it will refrain from its normal positioning as described below and take a position in the middle of the goal. Once it reaches this position, it will continually turn its body left and right to cover the greatest possible range of view until either of its teammates or the goalie himself has found the ball.

The GoaliePoseProvider performs position calculations for the goal keeper and provides some additional meta information which is used within the actual behavior. The main idea of the goalie positioning is to always stand between the middle point of the goal and the ball. For the calculation the module takes the current ball position as well as its speed and the information provided by teammates into account. If the keeper itself does not see the ball, but one of its teammates does, it uses this information to take its position and switches its head control in order to find the ball. While the ball is in the opponent half, the keeper will just stand in the middle of its goal until the ball actually passes the middle line or its estimated end position (cf. Sect. 4.2.2) is within the own half.

As the keeper takes its position between the ball and the middle of the goal it moves on a circle with the same middle point and a radius determined by the position of one goal post. To not leave the own penalty area, this circle is clipped to a straight line inside that area as seen in Fig. 5.6. The calculated position is also clipped in x-direction in order to keep enough distance to the goal posts (cf. Sect. 4.1.2 for the definition of the coordinate system).

While moving on the clipped line, the keeper will maintain a rotation towards the ball, which is clipped to a range of $-30°$ to $+30°$ (within the world coordinate system). This threshold exists so that the keeper stands roughly parallel to the ground line, when jumping to cover the greatest possible area. Otherwise, its rotation is directly aligned towards the ball. Of course, the keeper will leave the action area shown in the image if the ball is near or already in the own penalty area in order to get the ball away from the own goal.

An additional information the module provides is, whether the keeper's current position is near to any of the goal posts. This information is used to prevent it from jumping into a post when a ball approaches. It is considered to be near to a post, if its relative rotation towards the post is between 65° and 110° and the relative distance is lower than 600mm. If so, the keeper will not jump into the direction of the post but performs a genuflect motion instead.

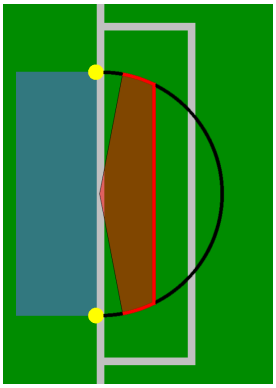


Figure 5.6: Half circle showing the goalie's action area (the red part). The radius is currently configured to exactly hit the goal posts, but may be modified through configuration files. The distance of the clipped line from the ground line as well as the x clipping to avoid running into goal posts can also be configured.

5.3.3 Kickoff

For 2013, the rules of the game have been changed in a way that required some adaptations for the kickoff behavior implementation: the number of robots has been increased by one, the defending team is allowed to take more offensive positions, the distance from the field's center to the goal has been increased significantly, and the defending team is not allowed to enter the center circle before the kickoff team has touched the ball or 10 seconds have elapsed.

Our current implementation is, on the one hand, simpler and less configurable than the previous solution but, on the other hand, more flexible and adaptive during execution. The kickoff options control the positioning behavior of all robots during the whole *READY* and *SET* game states as well as the action selection during the beginning of *PLAYING*.

5.3.3.1 Positioning

For the two main kickoff situations (defensive and offensive), distinct sets of kickoff poses are specified in a configuration file. Each pose set is ordered by the importance of the respective poses, i. e. if robots are missing, the least important positions are not taken. We do not rely on a fixed pose assignment (except for the goalkeeper, of course) based on robot numbers as such an approach has two significant drawbacks. Firstly, an important position might be left empty, if one robot is missing. Secondly, some robots might be forced to walk farther than necessary and thus risk to come too late.

As our self-localization provides a very precise pose estimate and all robots receive the poses of their teammates via team communication, a robust dynamic pose assignment is possible. Our approach is realized by two sequential assignment rules:

1. The robot that is closest to the center position, i. e. the position that is closest to $(0, 0)$ according to the global coordinate system as described in Sect. 4.1.2, will walk to that position.

2. For the remaining field players, the configuration that has the smallest sum of squared distances (between robots and positions) is determined.

The first rule ensures that the central position will always be occupied, if there is at least one field player left. This position is definitely the most important one as it is the one closest to the

ball. Furthermore, assigning this position to the closest robot strongly increases the likelihood that this robot actually reaches it in time. A robot that has a long way to walk during the *READY* state might be blocked by other robots or the referees. These delays could lead to a manual placement position which is too far away from the ball position.

We prefer a correct orientation to a correct position. Thus, to avoid having robots facing their own goal, the positioning procedure is stopped a few seconds before the end of the *READY* state to give a robot some time to take a proper orientation.

5.3.3.2 Actions after the Kickoff

When the game state changes from *SET* to *PLAYING*, not all robots immediately switch to their default playing behavior as some rules have to be considered.

If a team does not have kickoff, its robots are not allowed to enter the center circle until the ball has been touched or 10 seconds have elapsed in the *PLAYING* state. Therefore, our robots switch to a special option that calls the normal soccer behavior but stops the robots before entering the center circle without permission. The robots actively check, if the ball has already been moved in order to enter the center circle as early as possible.

If a team has kickoff, its robots are not allowed to score before the ball has left the center circle. Therefore, the normal playing behavior, which would probably try to make a direct shot into the goal, is postponed and a kickoff action is carried out. Our kickoff taker prefers to make a medium-length shot into the opponent half. The direction of the shot depends on the perceived obstacles around the center circle. If there seem to be no obstacles at all, the kickoff taker dribbles the ball to a position outside the center circle.

Both kickoff variants are terminated, i. e. the behavior is switched to normal playing, in two cases: if the ball has left the center circle or if a certain timeout (currently 15 seconds) has been reached.

5.3.4 Head Control

The head control controls the angles of the two head joints, and thereby the direction in which the robot looks. In our behavior, the head control is an independent option (*HeadControl*) running parallel to the game state handling and is called from the root option *Soccer*. This ensures that only one head control command is handled per cycle. However, the head control command is given by the game state handling option by setting the symbol `theHeadControlMode`. As long as the symbol does not change, the selected head control command 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 instance the GoalPerceptor, BallPerceptor, and the LinePerceptor provide input for the modeling modules (cf. Sect. 4.2) that provides localization, and the ball position. However, providing images from all relevant areas on the field is often mutually exclusive, e. g., when the ball is located in a different direction from the robot than the goal, it cannot look at both objects at the same time. 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 a high motion speed has a negative influence on the robot's walk stability due to the forces resulting from fast rotational movements of the head. With these known limitations, we had to design many head control modes for a variety of needs. 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 (option *SetHeadPanTilt*), a position on the field (option *SetHeadTargetOnGround*), or a position relative to the robot (option *SetHeadTarget*). The head control modes used in our behavior are:

off: In this mode, the robot turns off its head controlling joints.

lookForward: In this mode, the pan and tilt angles are set to static values, so that the robot looks forward. This mode is used during finished and penalized states.

lookLeftAndRight: In this mode, the robot moves its head left and right in a continuous motion. This mode is used for a first orientation after the robot has been penalized and gets back to the game. It is also used, when the robot or the whole team searches for the ball.

lookAtGlobalBall: As the name implies, this head control moves the head to the global ball estimate, which is primarily used by the keeper, whenever the ball is covered by field players.

lookAtGlobalBallMirrored: This mode is basically the same as the one above, but it moves the head toward the position of the global ball mirrored at the center point, assuming the robot might be at the mirrored position from where it thinks to be.

lookAtBall: When using this head control mode, the robot looks at its ball estimate. However, if the ball is not seen for some time, the head will be moved toward the global ball estimate. In case the ball is still not seen after some additional time, the robot just changes to `lookActive` mode. The mode `lookAtBall` is used, whenever a robot kicks the ball or is preparing to kick it as well as when the robot is in a duel. Nonetheless, this mode is important for the keeper, when the ball moves toward the goal and the keeper must prepare to catch it. Additionally, this mode is used by the robot roles used for penalty shoot-out (cf. Sect. 5.3.5).

lookAtBallMirrored: This is basically the same as the mode above, but the robot looks at the estimated ball position mirrored at the center point. This mode is used for checking, if the ball is mirrored from where the robot thinks it is.

lookAtOwnBall: This mode is nearly the same as `lookAtBall`, but – as the name implies – without taking the global ball estimate into account. This mode is used for checking, if the ball is mirrored from where the robot thinks it is.

lookAtOwnBallMirrored: This mode is analog to `lookAtBallMirrored`. It is basically the same as `lookAtOwnBall`, but it takes the mirrored position into account. This mode is used for checking, if the ball is mirrored from where the robot thinks it is.

lookActive: In this mode, the robot looks to the most interesting point on the field, which is calculated by `libLook`. During the game, the importance or unimportance of different points on the field – for example goals, field lines, and the ball position – changes, depending on the current situation. Therefore, our solution is based on having a set of points of interest which consists of static points on the field as well as of dynamic points like the ball estimate or the global ball estimate. All points that are reachable by the head are assigned a value representing the robot's interest in that point, depending on the robot's role and the time that has passed since the point has been seen. It also handles synchronized head control, i.e., when the striker looks away from the ball, all other robots that are currently using this head control mode shall focus the ball. Whether the striker looks

away from the ball is checked in the three basic head control options *SetHeadPanTilt*, *SetHeadTargetOnGround*, and *SetHeadTarget*. The synchronized head control shall ensure that at least one team member sees the ball. Additionally, while using this head control mode, robots that are currently not looking at the ball will turn their head as fast as possible toward the ball, when the global ball estimate moves toward the robot's current position. This head control mode is the default one and is always used when none of the other ones fits better.

lookActiveWithBall: This head control mode is basically the same as `lookActive`, but it limits the reachability of points of interest, because it requires the head control to always keep the ball in the field of sight. This mode is used, when the robot needs to see the ball – for example, because it comes near – but shall still see as much as possible of the field.

lookActiveWithoutBall: As the name implies, this mode is based on `lookActive`, but it completely ignores the ball. It is used during the Ready state.

5.3.5 Penalty Shoot-out Behavior

Due to rule changes for RoboCup 2013, we could not reuse our penalty shoot-out behavior from 2012, because the striker is not allowed to touch the ball multiple times. Therefore, we had to invent a new tactic. Since the keeper can simply track the ball and jump to catch it, the main idea for the striker was to kick the ball as near to the goal post as possible. Also, the ball shall be kicked with as much force as possible without any lack of precision. On the other side, the keeper shall also catch balls that are kicked near the goal post. The keeper's jump on the goal line would leave some space between the robot and the goal post. To remove this possibility for a ball passing the keeper and to cover the whole space between the goal posts, the keeper's position must be closer to the ball. Thus, the keeper shall move toward the ball before the striker kicks it.

The RoboCup 2013 penalty shoot-out behavior was embedded in the behavior described above. It is implemented by introducing two additional roles: *PenaltyStriker*, and *PenaltyKeeper* (cf. Sect. 5.3.1.2). Depending on the secondary game state and the current kick-off team, the RoleProvider assigns one of these two roles to the robot. Game state handling and head control is processed the same way as during an ordinary game.

PenaltyKeeper: The penalty keeper starts moving toward a position in the middle of the penalty area, using the time the striker needs for getting to the ball. As described above, this is done to cover the whole goal with a keeper jump. How much time the keeper has to take the required position is defined in *Behavior2013Parameters*. When the position is reached or the defined time has passed, the keeper has some additional time to check its rotation toward the ball. After that, the penalty keeper sits down as preparation to catch the ball. When the ball moves towards the keeper, depending on where the ball is assumed to pass the robot, the keeper either jumps left, jumps right, or uses the `genuflect` movement, i. e. a sit down movement where the robot spreads its legs. Regarding the head control, the penalty keeper always keeps the ball in sight of view, using `lookActiveWithBall` while taking position and `lookAtBall` after that (cf. Sect. 5.3.4). As fallback, the penalty keeper also has a localization behavior for searching the ball, when it is not seen for some time.

PenaltyStriker: The penalty striker differs between two variants. Normally, the first one shall be used. In this case, the robot randomly decides in the beginning, if it wants to kick at

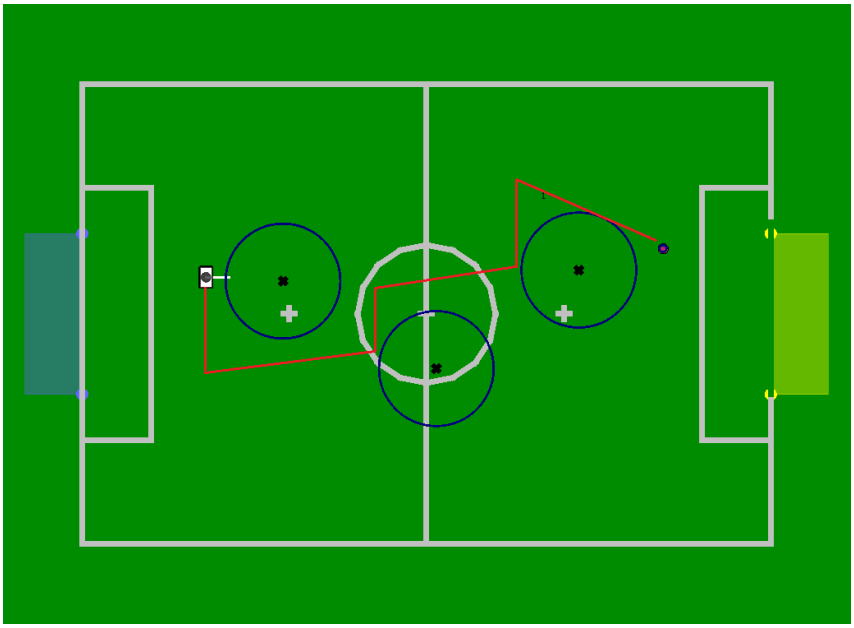


Figure 5.7: A path used to walk toward the ball using the reactive avoidance.

the right edge of the goal or at the left edge. The robot moves towards the ball, already rotating in the direction, it wants to kick to. Reaching a specific distance from the ball, the striker stops for a short time, so it will not accidentally run against the ball. Then the robot makes the last few steps towards the ball, getting in position for the kick. In the end, a special kick for the penalty shoot-out is executed. In order to achieve a repeatable very precise ball trajectory the execution time of this kick is rather large. The second variant is a fallback, when there is just little time left for the shot. This variant lets the robot just move towards the ball and kick it, using the KickPoseProvider (cf. Sect. 5.5). For the head control, the penalty striker simply uses `lookAtBall`, because it needs to focus at the ball all the time. This is acceptable, because due to its position on the field and its rotation toward the goal, both goal posts are also always in sight. Analog to the penalty keeper, the striker has a localization behavior for searching the ball, which is activated when the ball is not seen for some time.

5.4 Path Planner

Until 2010, the team used a reactive obstacle avoidance based on ultrasonic measurements. Such a reactive avoidance is not optimal because the robot can only react to the obstacle just measured, which might lead to the problem that an avoidance ends in a position close to another obstacle. Another problem of a reactive avoidance is that it is executed by walking sidewards, until the distance to the measured obstacle is high enough to walk forward again, which can take a very long time (cf. Fig. 5.7). Moreover, it is possible that the other robot that was detected as obstacle, uses the same kind of obstacle avoidance. This could lead to the situation that both robots avoid in the same direction until they leave the field.

For this reason, we developed a path planner that is based on the combined world model (cf. Sect. 4.2.6). The algorithm used for finding a path is the extended bidirectional Rapidly-Exploring Random Tree (RRT). This non-optimal algorithm is based on the random exploration of the search space and works on continuous values.

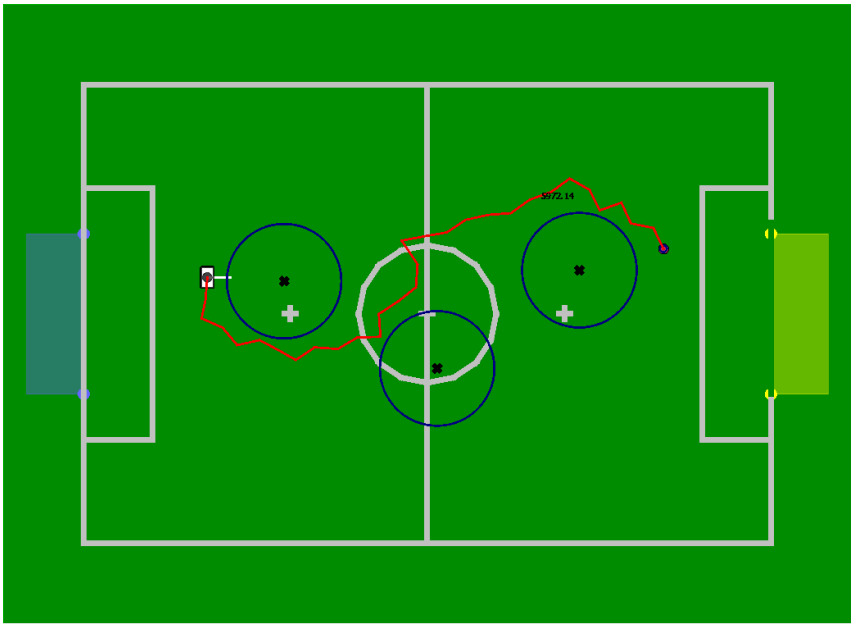


Figure 5.8: A path used to walk toward the ball using the RRT path planner.

In general the Rapidly-Exploring Random Tree algorithm[16] works as follows:

1. Add the starting position to an empty set.
2. Create a random position within the search space.
3. Search for the nearest neighbor in the given set.
4. Expand the found neighbor toward the random position.
5. Check for collisions between the created edge and an obstacle.
6. If no collision was found, add the new position to the current set of nodes.
7. Repeat step 2–6 until the target is reached.

This algorithm builds up a tree that quickly expands in a few directions of the search space, because in each step of the algorithm, the tree is enlarged by one edge in a random direction (cf. Fig. 5.9). For this general algorithm, different variants exist (e. g., RRT-Extend [16], RRT-Connect [12], and RRT-Bidirectional[17]) of which the extend and bidirectional[1] ones were used here (cf. Fig. 5.10). Using the extend variant restricts the expansion toward the random position to a given distance that is the same for each expansion, which has a direct influence on the expansion of the tree, whereas the bidirectionally approach used has no influence on the tree itself but it is used to decrease the run time. This is achieved by creating two separate trees, one beginning from the start point and one from the end point. Another modification is to replace the random position by the target position or a waypoint of the last path found with a given probability. Using these modifications helps to avoid oscillations of the path, for instance if there is an obstacle on the direct way to the target.

In general, the advantage of using a path planner is to include distant obstacles to avoid the situation mentioned above (after the avoidance, a new obstacle is in front of the robot). Moreover, the time for avoiding an obstacle can be decreased because the robot walks on a circular

[1] Actually a slightly modified variant was used here.

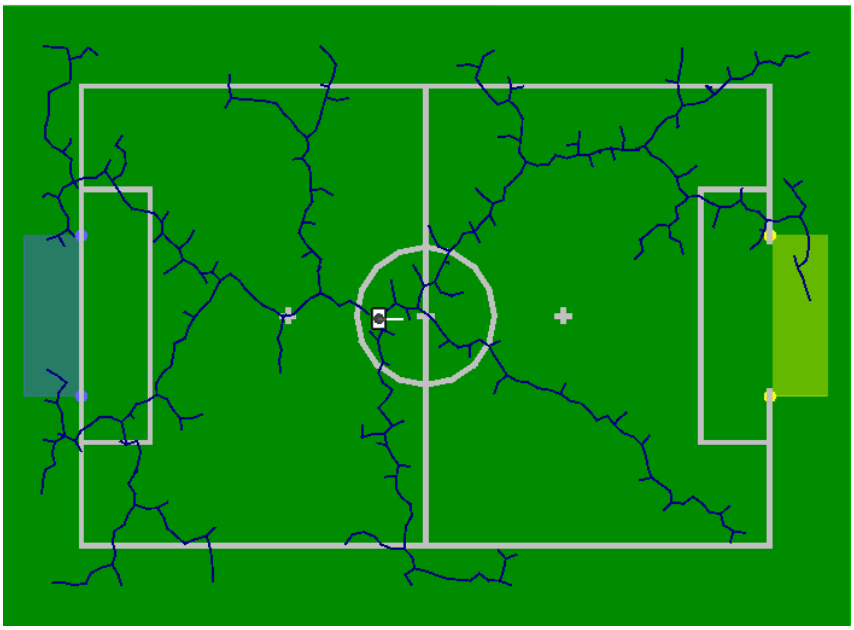


Figure 5.9: A tree built up with the RRT algorithm.

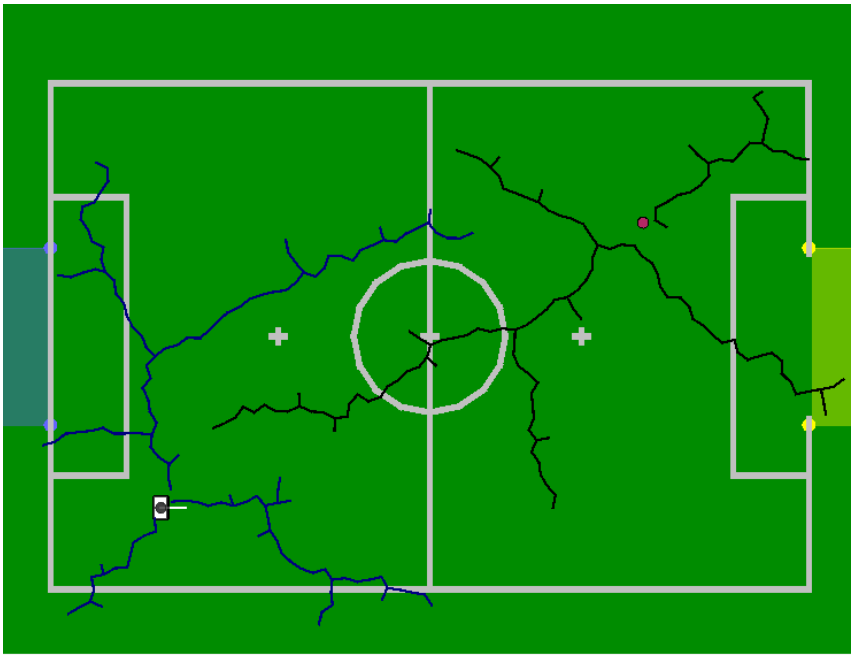


Figure 5.10: Two trees built up with a combination of the RRT-EXTEND and the RRT-BIDIRECTIONAL algorithm.

path around the obstacles. Only if an obstacle is too near, the robot needs to walk sidewards and decrease the forward speed (cf. Fig. 5.8).

For using the path planner, it is sufficient to set the start and end position with the corresponding rotation. With these positions, a path is generated, out of which the speed for walking along the path is calculated.

5.5 Kick Pose Provider

To calculate the optimal position for a kick toward the opponent's goal, a module called KickPoseProvider is used. For every possible kick (forward, sidewards, backwards), the module requires information about how long it takes to perform the kick and what offset the robot has to have to the ball including a rotation offset. The rotation offset defines an angle, by which the target pose for performing the kick must be rotated to have the ball move in the intended direction after the kick. This data is provided by the representation *KickInfo*. However, the *KickInfo* representation's data is not set by any module, it initializes all available kicks during its construction. Based on the information from this representation, the module calculates a *KickPose* for each kick, 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 of the possible poses.

The evaluation of the different poses is based on several criteria. The most important one is 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 instance, 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 for the keeper is allowed to be within the penalty area. Such a situation might occur 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 and thereby giving the keeper more time to kick it out of the penalty area.

The kick is usually directed toward the center of the opponent goal, however there is an exception. If the goal is not free, i. e. a robot is blocking the goal, the kick is directed toward the center of the largest free part of the opponent goal provided by the representation *FreePartOfOpponentGoalModel* (cf. Sect. 4.2.4). However, this modification of the kick direction is only applied if the opening angle toward the opponent's goal is large enough, i. e. it is not less than the configured kick pose inaccuracy (defined by the parameter `kickPoseKickInaccuracy`).

5.6 Camera Control Engine

The CameraControlEngine takes the *HeadMotionRequest* provided by the BehaviorControl2013 and modifies it to provide the *HeadAngleRequest*, which is used to set the actual head angles. The main function of this module is to make sure that the requested angles are valid. In addition it provides the possibility to either use the so called *PanTiltMode*, where the user can set the desired head angles manually, or the target mode. In target mode the CameraControlEngine takes targets on the field (provided by the user) and calculates the required head angles by using inverse kinematics (cf. Sect. 6.2.1.2). It must also be ensured that the cameras cover as much of the field as possible. Therefore, the CameraControlEngine calculates which camera suits best the provided target and sets the angles accordingly. Also, because of this, most of the time

the provided target will not be in the middle of either camera image.

5.7 LED Handler

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.

Right Eye

Color	Role	Additional information
Blue	All	No ground contact
Blue	Keeper	
White	Defender	
Green	Supporter	
Magenta	Breaking Supporter	
Red	Striker	

Left Eye

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

Torso (Chest Button)

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

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, when the robot is the penalty taker, and yellow if it is the goal keeper.

Ears

- The right ear shows the battery level of the robot. For each 10% of battery loss, an LED is switched off.

- The left ear shows the number of players connected through the wireless. For each connected player, two LEDs are switched on (upper left, lower left, lower right, upper right).

Chapter 6

Motion

The process *Motion* comprises the two essential tasks *Sensing* (cf. Sect. 6.1) and *Motion Control* (cf. Sect. 6.2). *Sensing* contains all modules (except camera modules) that are responsible for preprocessing sensor data and *Motion Control* all modules that compile joint angles.

The results of *Sensing* are required by some tasks of the *Cognition* process as well as by the *Motion Control* task of the *Motion* process itself. Hence, the *Sensing* task is executed before the *Motion Control* task just as before the tasks of the *Cognition* process that require data from *Sensing*.

6.1 Sensing

The NAO has an inertial measurement unit (IMU) with three acceleration sensors (for the x-, y-, and z-direction) and two gyroscopes (for the rotation around the x- and y-axes). By means of these measurements the IMU board calculates rough approximations of the robot's torso angles relative to the ground, which are provided together with the other sensor data. Furthermore, the NAO has a sensor for the battery level, eight force sensing resistors on the feet, two ultrasonic sensors with different measurement modes, and sensors for the load, temperature, and angle of each joint.

The NaoProvider receives all sensor readings from the *NaoQi* module *libbhuman* (cf. Sect. 3.1), adds a configured calibration bias for each sensor, and provides them as *SensorData* and *JointData*. The *JointData* passes through the JointFilter module that provides the representation *FilteredJointData* (cf. Sect. 6.1.1). Similarly, the *SensorData* passes the modules InertiaSensorCalibrator (cf. Sect. 6.1.5), InertiaSensorFilter (cf. Sect. 6.1.6), and also the SensorFilter which provides the representation *FilteredSensorData*.

The module InertiaSensorCalibrator determines the bias of the readings from the IMU considering the *RobotModel* (cf. Sect. 6.1.4) and the *GroundContactState* (cf. Sect. 6.1.2). Thereby, the *RobotModel* provides the position of the robot's limbs and the *GroundContactState* the information whether the robot's feet are touching the ground. Based on the *FilteredSensorData* and the *RobotModel* it is possible to calculate the *TorsoMatrix* (cf. Sect. 6.1.7) which describes a transformation from the ground to an origin point within the torso of the NAO. The representation *FilteredSensorData* is also considered to detect whether a robot has fallen down by the module FallDownStateDetector which provides the *FallDownState* (cf. Sect. 6.1.8). The JointDynamicsProvider provides both the *JointDynamics* and the *FutureJointDynamics* for the IndyKickEngine. Details about joint dynamics are described by Wenk and Röfer [31]. Figure 6.1 shows all *Sensing* modules and the provided representations.

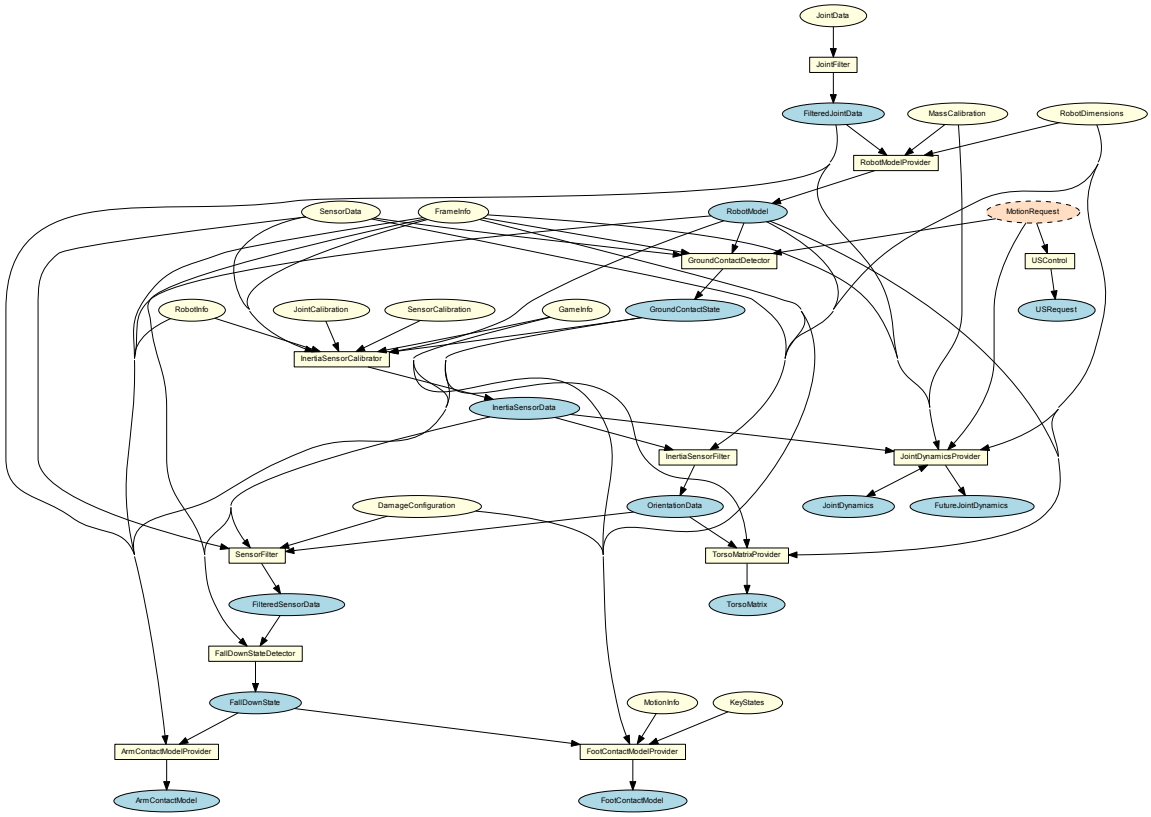


Figure 6.1: All modules and representations in the area *Sensing*. Blue ellipses mark representations provided by sensing modules, which are marked as white squares. White ellipses mark required representations from sensing modules. In contrast, red ellipses with a dashed outline mark required representations from other sources than sensing.

6.1.1 Joint Data Filtering

The measured joint angles of the NAO are very accurate, so the main task of the module JointFilter is to ensure that the representation *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.

6.1.2 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 InertiaSensorFilter and the SensorFilter, the module GroundContactDetector, that provides the *GroundContactState*, is the only module that uses the unfiltered *SensorData*, since

the *GroundContactState* is required by the InertiaSensorCalibrator. Hence it is not possible for the GroundContactDetector to use the *FilteredSensorData*.

Due to the inaccuracy and failure of the NAO V3's force sensing resistors, these are no longer the basis of the ground contact detection. The current GroundContactDetector measures the noise in the accelerometers and gyroscopes and calculates the average values over 60 frames. If all values exceed a preset threshold, the *GroundContactState* becomes false. The assumption for this method is that movements that cause high noise in the accelerometers and gyroscopes tend to be those ones, which cause loosing ground contact like lifting the robot, or the robot falling down.

The NAO V4's force sensing resistors seem to be much more reliable. To use them for ground contact detection, the representation *FsrData* could be used. Since our team still uses NAO V3 bodies, we still rely on the IMU measurements instead.

6.1.3 Fsr Data

This module calculates foot contact and center of pressure using the force sensing resistors. First the weights of all resistors on each foot are summed up. If the sum, in addition with a custom offset, is greater or equal than the total mass of the robot, the corresponding foot has ground contact. If so, also the center of pressure is calculated by accounting for the weight with the positions of the sensor that measured the value. This position lacks a certain accuracy, because of the auto calibration in the force sensing resistors' hardware.

6.1.4 Robot Model Generation

The *RobotModel* is a simplified representation of the robot. It provides the positions and rotations of the robot's limbs relative to its torso as well as the position of the robot's center of mass (CoM). All limbs are represented by homogeneous transformation matrices ($Pose3D$) whereby each limb maps to a joint. By considering the measured joint angles of the representation *FilteredJointData*, the calculation of each limb is ensured by the consecutive computations of the kinematic chains. Similar to the inverse kinematic (cf. Sect.6.2.1.2) the implementation is customized for the NAO, i. e., the kinematic chains are not described by a general purpose convention such as Denavit-Hartenberg parameters to save computation time.

The CoM is computed by equation (6.1) 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.

$$\vec{r}_{com} = \frac{\sum_{i=1}^{n} \vec{r}_i m_i}{\sum_{i=1}^{n} m_i} \tag{6.1}$$

For each limb, $\vec{r}_i$ is calculated by considering the representation *RobotDimensions* and the position of its CoM (relative to the limb origin). The limb CoM positions and masses are provided in the representation *MassCalibration*. They can be configured in the file *massCalibration.cfg*. The values used were taken from the NAO documentation by Aldebaran.

6.1.5 Inertia Sensor Data Calibration

The module InertiaSensorCalibrator determines a bias for the gyroscope and acceleration sensors. Therefore, it considers the *SensorData* and provides the *InertiaSensorData*. It uses the *GroundContactState* to avoid a calibration when the robot is not standing on an approximately even ground.

The gyroscope sensors are hard to calibrate, since their calibration offset depends on the sensor's temperature, which cannot be observed. The temperature changes slowly while the robot is active. Hence, it is necessary to redetermine the calibration offset constantly. To accomplish this, it is hypothesized that the robot has the same orientation at the beginning and the ending of a walking phase, which means that the sum over all gyroscope values collected during a single walking phase should be zero. 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. The result is used as offset for the gyroscope sensor. The collection of gyroscope values is restricted to slow walking speeds. 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 feet are evenly on the ground to calculate the orientation of the robot's body and the expected acceleration sensor readings. These expected acceleration sensor readings are considered to determine the calibration offset.

6.1.6 Inertia Sensor Data Filtering

The InertiaSensorFilter module determines the orientation of the robot's torso relative to the ground. Therefore, the calibrated IMU sensor readings (*InertiaSensorData*) and the measured stance of the robot (*RobotModel*) are processed using an Unscented Kalman filter (UKF) [10].

A three-dimensional rotation matrix, which represents the orientation of the robot's torso, is used as the estimated state in the Kalman filtering process. In each cycle, the rotation of the torso is predicted by adding an additional rotation to the estimated state. The additional rotation is computed using the readings from the gyroscope sensor. When these readings are missing, since they have been dropped (cf. Sect. 6.1.5), the alteration of the feet's rotation relative to the torso is used instead.

After that, two different cases are handled. In the case that it is obvious that the feet of the robot are not evenly resting on the ground (when the robots fell down or is currently falling), the acceleration sensors are used to update the estimated rotation of torso, in order to avoid accumulating an error caused by slightly miscalibrated or noisy sensors. In the other case, an orientation of the robot's torso computed from the *RobotModel*, assuming that at least one foot of the robot rests evenly on the ground, is used to update the estimate.

The resulting orientation (cf. Fig. 6.2) is provided as *OrientationData*.

6.1.7 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 alteration of the position of the center of hip including the odometry. Hence, the *TorsoMatrix* is used by the *WalkingEngine* for estimating the odometry offset. The *CameraMatrix* within the *Cognition* process is computed based on the *TorsoMatrix*.

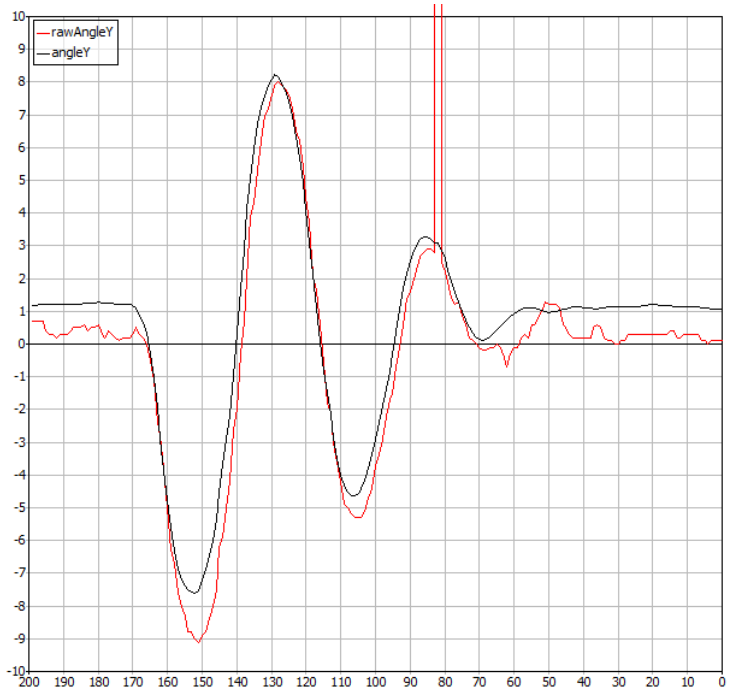


Figure 6.2: Comparison of the estimated pitch angle *angleY* provided by the InertiaSensorFilter module and the pitch angle *rawAngleY* that has been computed on IMU board. Outliers as shown here between frames 85 and 80 do not seem to occur anymore on current NAO robots.

In order 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 the torso to each foot (t_l and t_r). This can be calculated by the kinematic chains, according to the estimated rotation (cf. Sect. 6.1.6). The estimated rotation is represented as rotation matrix R.

$$f_l = -R \cdot t_l \tag{6.2}$$
$$f_r = -R \cdot t_r \tag{6.3}$$

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

$$s = f_r - f_l \tag{6.4}$$

Now, it is possible to calculate the translation part of the torso matrix p_{im} by using the longer leg. The rotation part is already known since it is equal to R.

$$p_{im} = \begin{cases} s/2 + f_l & \text{if } (f_l)_z > (f_r)_z \\ -s/2 + f_r & \text{otherwise} \end{cases} \tag{6.5}$$

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. The odometry offset is calculated by using the change of the span s between both feet and the change of the ground foot's rotation as well as the new torso matrix.

6.1.8 Detecting a Fall

Although we try our best at keeping our robots upright, it occasionally still happens that one of them falls over. In such a case, it is helpful to switch off the joints and bring the head into a safe position to protect the robot's hardware from unnecessary damage. The task of detecting such a situation is realized by the FallDownStateDetector, which provides the *FallDownState*. After detecting a fall, we only have a very small amount of time to perform the necessary actions, because we have to be completely sure that the fall is inevitable. There would be fatal consequences if a fall was mistakenly detected, and instead of protecting the joints by switching them off, the robot would suffer an unnecessary collapse. For that reason, fall detection is split up into 2 phases. The first of these is to detect that the robot is staggering. If that is the case, we bring the head into a safe position, which can be done without risking the robot's stability. 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 been shown that this is a better approach than completely switching them off, 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*. Still, there are some exceptions, when the *FallDownState* is undefined, and thus the safety procedures mentioned above are not triggered, in particular if the keeper is diving, because these motions switch off the joints on their own.

To start an appropriate get-up motion after a fall, the FallDownStateDetector determines, whether the robot is lying on its front, its back, its left side, or its right side. The latter two cases appear to be highly unlikely, but are not impossible. Note that we distinguish cases, in which the robot falls sidewards first, then continuing to fall on its back or front. We use this information to correct the robot's odometry data, in order to speed up regaining knowledge of its position on the field after standing up again.

6.2 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 *walk*, *stand*, *bike*, *indyKick*, *getUp*, *takeBall*, and *specialAction*. The walking motion and a corresponding stand are dynamically generated by the WalkingEngine (cf. Sect. 6.2.1). Some kicking motions are generated by the kick engine BLAME [18] that models motions using Bézier splines and inverse kinematics. The new kick engine Indykick was only used during the passing challenge. It generates the foot trajectory and dynamically balances the body by calculating the zero moment point of the robot using inverse dynamics. as described in detail by Wenk and Röfer [31]. Ball taking and getting up from a position lying on the ground are special tasks, which also have their own motion modules, i. e. the BallTaking and the GetUpEngine. All other whole body motions are provided by the module SpecialActions in the form of sequences of static joint angle patterns (cf. Sect. 6.2.2). In addition to these whole body motions, there are two motion engines only for certain body parts, i. e. the HeadMotionEngine and the ArmMotionEngine.

All six whole body motion modules generate joint angles. The WalkingEngine provides the *WalkingEngineOutput* and the *WalkingEngineStandOutput*. It also takes the *HeadJointRequest* provided by the HeadMotionEngine and the *ArmMotionEngineOutput* from the ArmMotionEngine and includes these partial body movements into the output. The module BLAME provides the *BikeEngineOutput*, the GetUpEngine the *GetUpEngineOutput*, the module BallTaking the *BallTakingOutput*, and the module SpecialActions provides the *SpecialActionsOutput*. According to the *MotionRequest*, the MotionSelector (cf. Sect. 6.2.5) calculates which motions to execute and

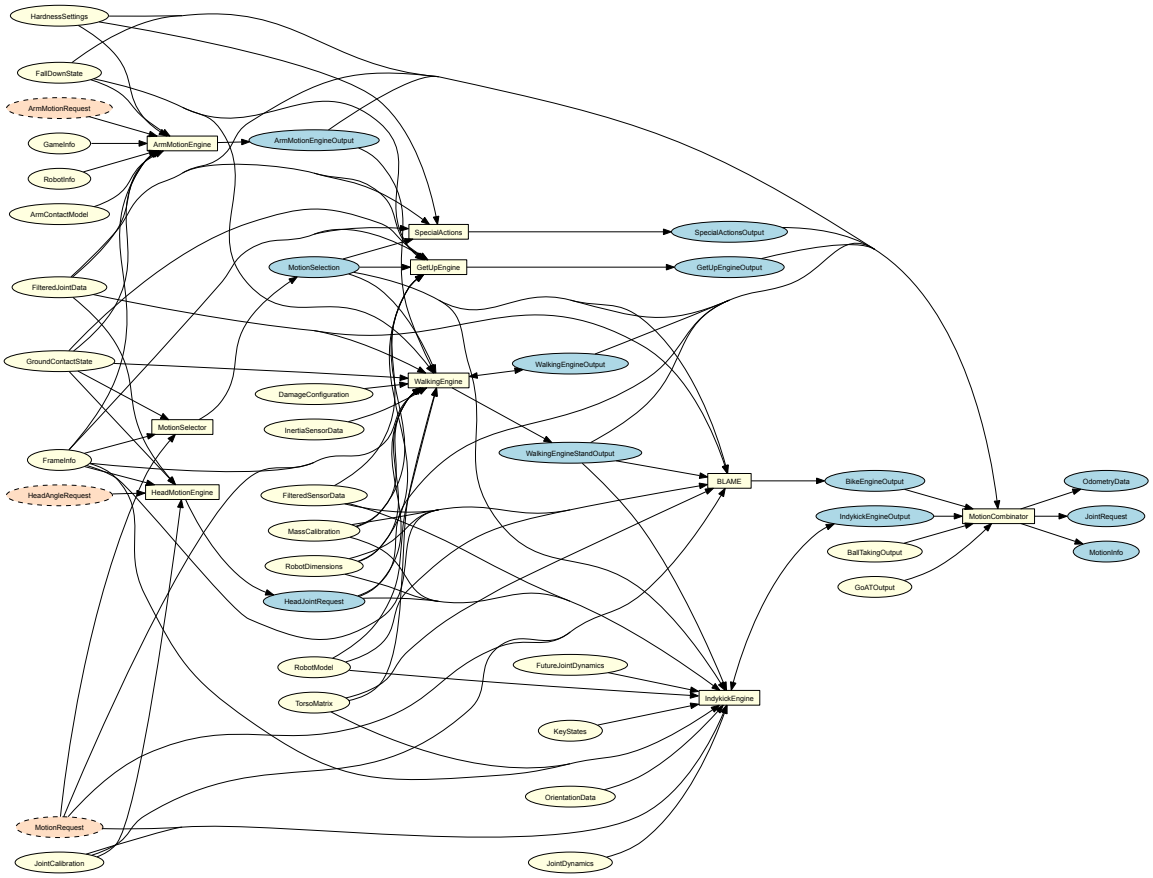


Figure 6.3: All modules and representations in the area *Motion Control*. Blue ellipses mark representations provided by motion control modules that are marked as white squares. White ellipses mark required representations from motion control modules. In contrast, red ellipses with a dashed outline mark required representations from other sources than motion control.

how to interpolate between different motions, while switching from one to another. This information is provided by the representation *MotionSelection*. If necessary, two modules calculate their joint request and the MotionCombinator (cf. Sect. 6.2.5) combines these according to the *MotionSeletion*. The MotionCombinator provides the final *JointRequest* and additional data such as the *OdometryData* and the *MotionInfo*. Figure 6.3 shows all modules in the area *Motion Control* and the representations they provide.

6.2.1 Walking

The module WalkingEngine generates walking motions and joint angles for standing. Therefore, it provides the two sets of joint angles *WalkingEngineOutput* and *WalkingEngineStandOutput* in each cycle of the process *Motion*. The distinction between these two sets exists since the MotionCombinator (cf. Sect.6.2.5) expects two individual joint angle sets for standing and walking. However, both of the provided sets are similar to allow the WalkingEngine to handle the transition between standing and a walking and vice versa on its own.

The walking motions are generated based on the computationally inexpensive model of an inverted pendulum. Using this model, a trajectory for the robot's center of mass can be planned

where the supporting foot does not tip over one of the foot's edges. The planned trajectory can then be transformed into a series of joint angles where the center of mass follows the planned trajectory. To compensate inaccuracies of the pendulum model and to react on external forces, the resulting motion of the center of mass is observed based on data of the *TorsoMatrix* and the *RobotModel*. If the observed motion does not correspond to the planned center of mass trajectory, the planned trajectory is slightly adjusted to ensure that further steps can be performed without causing the robot to fall over. This approach was released and described in further detail in [6] and [5].

The generated gait and the stand posture can be customized using the configuration file *Config/walkingEngine.cfg* (or individually for each robot using *Config/Robots/<robot>/walkingEngine.cfg*). The parameters are categorized in the groups *stand*, *walk*, *observer*, and *balance*. The *stand* parameters control the stand posture, which also determines the initial pose for walking motions. The *walk* parameters control how the feet are moved and how the inverted pendulum is used to compute the center of mass trajectory. The *observer* parameters and *balance* parameters control how the observed center of mass position is used to adjust the planned center of mass trajectory.

6.2.1.1 In-Walk Kicks

Besides walking and standing the WalkingEngine has also minor kicking capabilities. The tasks of walking and kicking are often treated separately, both solved by different approaches. In the presence of opponent robots, such a composition might waste precious time as certain transition phases between walking and kicking are necessary to ensure stability. A common sequence is to walk, stand, kick, stand, and walk again. Since direct transitions between walking and kicking are likely to let the robot stumble or fall over, the WalkingEngine is able to carry out sidewards and forward kicks within the walk cycle.

Such an in-walk-kick is described individually for each kick type (forwards or sidewards) by a number of parameters, which are defined in the configuration files of the *Config/Kicks* directory. On the one hand one configuration defines the sizes and speeds of the step before and during the kick (*setPreStepSize* and *setStepSize*), on the other hand a 6-D trajectory (three degrees in both translation and rotation; *setLeg* and *proceed*). This 6-D trajectory overlays the original trajectory of the swinging foot and thereby describes the actual kicking motion (cf. Fig. 6.4). In doing so the overlaying trajectory starts and stops at 0 in all dimensions and also in position and velocity. Thus, the kick retains the start and end positions as well as the speeds of a normal step. The instability resulting from the higher momentum of the kick is compensated by the walk during the steps following the kick.

6.2.1.2 Inverse Kinematic

The inverse kinematics for the ankles of the NAO are a central component of the module WalkingEngine and are used for generating dynamic motions using BLAME. In general, they are a handy tool for generating motions but 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.

a) b)

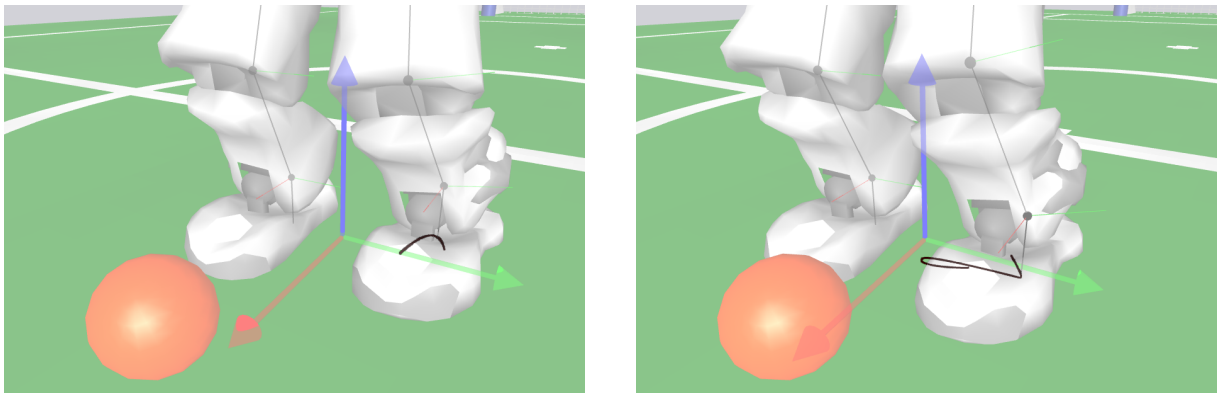


Figure 6.4: Visualization of swing foot trajectories (dark curves) while the robot approaches the ball. a) A swing foot trajectory for walking forwards and b) a modified swing foot trajectory for performing a sidewards in-walk kick to the right.

The target of the feet is given as homogeneous transformation matrices, i. e., matrices containing the rotation and the translation of the foot in the coordinate system of the torso. In order 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 $Foot2Torso$ is given as input, which describes the foot position relative to the torso. The coordinate frames used are depicted in Fig. 6.5.

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]. It is a simple translation along the y-axis[2]

$$Foot2Hip = Trans_y\left(\frac{l_{dist}}{2}\right) \cdot Foot2Torso \tag{6.6}$$

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.

$$Foot2HipOrthogonal = Rot_x(\frac{\pi}{4}) \cdot Foot2Hip \tag{6.7}$$

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 by which means they can be computed directly.

$$HipOrthogonal2Foot = Foot2HipOrthogonal^{-1} \tag{6.8}$$

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 (6.9). Knowing that the angle enclosed by the limbs corresponds to the knee joint, that joint angle is computed by equation (6.10).

[1]The computation is described for one leg and can be applied to the other leg as well.

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

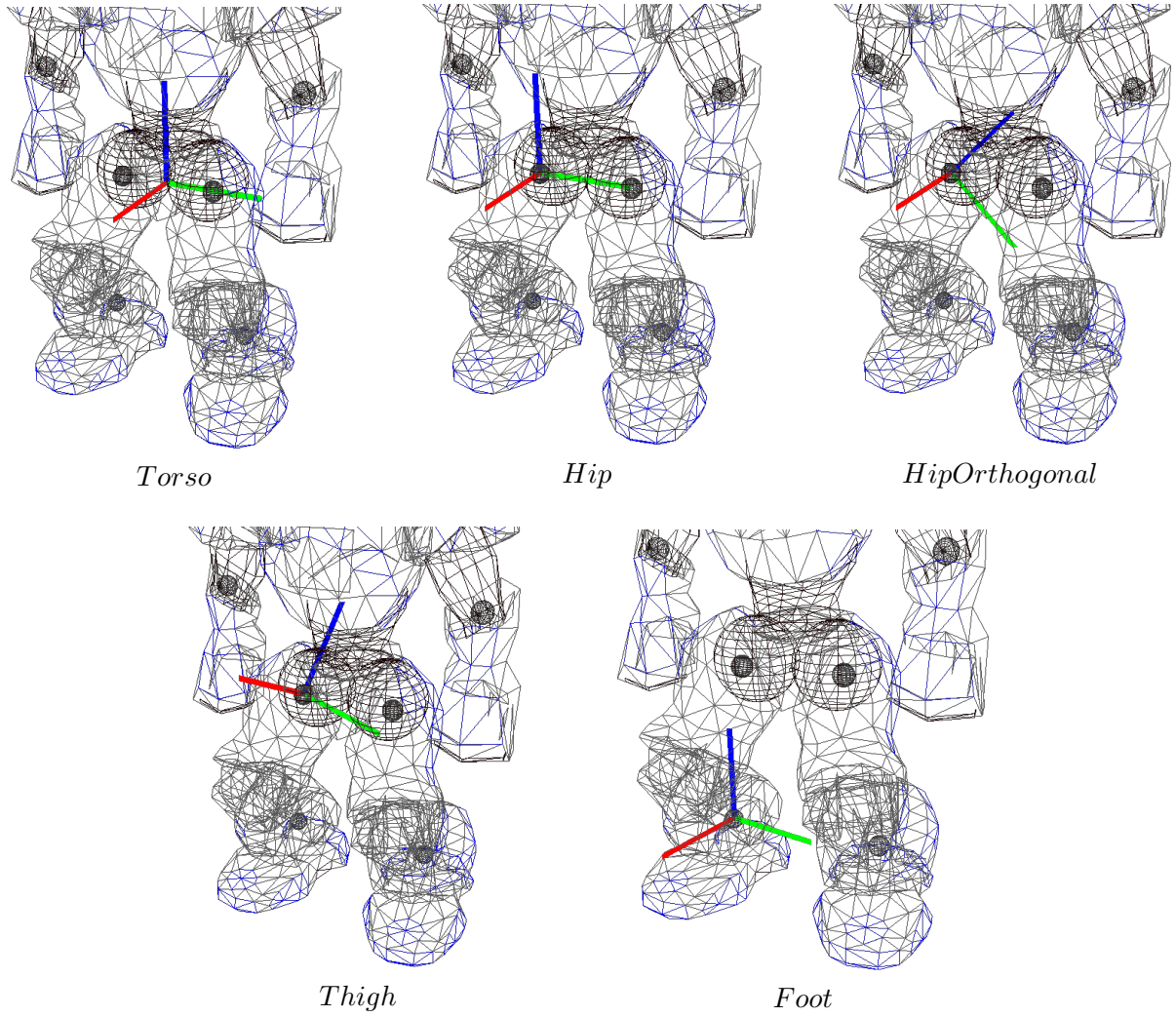


Figure 6.5: Visualization of coordinate frames used in the inverse kinematic. Red = x-axis, green = y-axis, blue = z-axis.

$$c^2 = a^2 + b^2 - 2 \cdot a \cdot b \cdot \cos\gamma \tag{6.9}$$

$$\gamma = \arccos \frac{{l_{upperLeg}}^2 + {l_{lowerLeg}}^2 - {l_{trans}}^2}{2 \cdot l_{upperLeg} \cdot l_{lowerLeg}} \tag{6.10}$$

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

$$\delta_{knee} = \pi - \gamma \tag{6.11}$$

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

$$\delta_{footPitch1} = \arccos \frac{{l_{lowerLeg}}^2 + {l_{trans}}^2 - {l_{upperLeg}}^2}{2 \cdot l_{lowerLeg} \cdot l_{trans}} \tag{6.12}$$

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]

$$\delta_{footPitch2} = \text{atan2}(x, \sqrt{y^2 + z^2}) \tag{6.13}$$

$$\delta_{footRoll} = \text{atan2}(y, z) \tag{6.14}$$

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.

$$\delta_{footPitch} = \delta_{footPitch1} + \delta_{footPitch2} \tag{6.15}$$

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

$$HipOrthogonal2Thigh = Thigh2Foot^{-1} \cdot HipOrthogonal2Foot \tag{6.16}$$

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

$$Thigh2Foot = Rot_x(\delta_{footRoll}) \cdot Rot_y(\delta_{footPitch}) \cdot Trans_z(l_{lowerLeg}) \cdot Rot_y(\delta_{knee}) \cdot Trans_z(l_{upperLeg}) \tag{6.17}$$

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 $\cos\delta_x$).

$$Rot_{Hip} = Rot_z(\delta_z) \cdot Rot_x(\delta_x) \cdot Rot_y(\delta_y) = \begin{pmatrix} c_y c_z - s_x s_y s_z & -c_x s_z & c_z s_y + c_y s_x s_z \\ c_z s_x s_y + c_y s_z & c_x c_z & -c_y c_z s_x + s_y s_z \\ -c_x s_y & s_x & c_x c_y \end{pmatrix} \tag{6.18}$$

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:

$$\frac{-r_{01}}{r_{11}} = \frac{\cos\delta_x \cdot \sin\delta_z}{\cos\delta_x \cdot \cos\delta_z} = \frac{\sin\delta_z}{\cos\delta_z} = \tan\delta_z \tag{6.19}$$

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

$$\delta_z = \delta_{hipYaw} = \text{atan2}(-r_{01}, r_{11}) \tag{6.20}$$

[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.

$$\delta_y = \delta_{hipPitch} = \text{atan2}(-r_{20}, r_{22}) \tag{6.21}$$

At last the rotation by 45 degrees (cf. eq. 6.7) has to be compensated in joint space.

$$\delta_{hipRoll} = \delta_x - \frac{\pi}{4} \tag{6.22}$$

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 setting the ratio. This is necessary, because if the values differ, only one leg can realize the desired target. Normally, the support leg is supposed to reach the target position exactly. By applying this fixed hip joint angle the leg joints are computed again. In order 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, except it is the other way around. The need to invert the calculation is caused by the fixed hip joint angle and the additional virtual foot joint, 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.

6.2.2 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 a goalie dive or the high stand posture. 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 engine), a '–' deactivates the joint. Each line ends with two more values. The first decides whether the target angles will be set immediately (the value is 0); forcing the robot to move its joints as fast as possible, or whether the angles will be reached by interpolating between the current and target angles (the value is 1). The time this interpolation takes is read from the last value in the line. It is given in milliseconds. If the values are not interpolated, the robot will set and hold the values for that amount of time instead.

It is also possible to change the hardness of the joints during the execution of a special action, which can be useful, e. g., to achieve a stronger kick while not using the maximum hardness as default. This is done by a line starting with the keyword *hardness*, followed by a value between

[5]Ignored

0 and 100 for each joint (in the same sequence as for specifying actual joint angles). In the file *Config/hardnessSettings.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. Furthermore, there 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 must 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/specialActions.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.

6.2.3 Get Up Motion

Besides the complex motion engines (e. g. walking and kicking) there are also some smaller, task-specific engines such as the GetUpEngine. As the name implies, this engine specializes in getting a lying robot back on its feet and replaces our previous special-action-based solution.

While using a key-frame-based approach, we included so called critical parts in order to verify certain body poses, e. g. if the robot's torso is near an upright position. Whenever one of the critical parts is unsuccessfully passed, the motion engine stops the current execution and initiates a recovery movement in order to try again from the start. After two attempts, a hardware defect is assumed causing the engine to set the stiffness of all joints to zero. Thus, the robot stays lying down in order to prevent any more damage. Furthermore, a cry for help is initiated causing the robot to play the sound "request for pick-up".

There is only one way to terminate the execution of this motion engine: get the robot into an upright position. This can either be done by a successful standing up movement or (after two

unsuccessful attempts) manually by holding the robot upright as a referee would do following the fallen robot rule.

As we all know robot soccer is often very rough and even robots that are almost finished with a stand up movement could be in distress. Thus, we included the possibility to enable a PID-controlled gyroscope based balancing (similar to the one in BLAME by Müller *et al.* [18]) starting at certain parts of the movement until it is finished, e. g. if the robot's feet have ground contact.

It is desirable that a motion engine is easy to use. For that reason, the last feature is that the engine is able to distinguish on its own whether the robot has fallen on its front, back, or not at all. Thereby, the fall direction of the representation *FallDownState* is used to select the appropriate motion, e. g., standing up from lying on the front side or standing up from lying on the back side. In the case that the robot has not fallen down, the engine terminates immediately, i. e. flags itself as "leaving is possible" and initiates a stand. Furthermore, after each unsuccessful stand up attempt, the engine reviews the fall direction in case an interfering robot has changed it.

Thus, if a fall is detected, all that has to be done is to activate the engine by setting the `motion` to `getUp` in the representation *MotionRequest* and wait for the termination (i. e. the "leaving is possible" flag).

6.2.4 Ball Taking

The BallTaking calculates a motion that moves one leg out to stop an approaching ball. This motion is achieved by using inverse kinematics to calculate 3D poses for each foot. The calculation is divide into several phases:

First of all, some data about the ball is calculated to be able to react to it in the motion phases. This contains checking whether the ball will cross the y-axis of the robot's coordinate system within a certain range, whether the ball will stop behind the robot, whether the angle of the approaching ball is valid for taking, whether the ball is moving, and whether it was seen constantly. All criteria have to be met to have a valid ball for taking.

In phase zero a stand motion is executed, while ensuring the side on which the ball will cross the robot. For one second, the y-axis values for the crossing of the ball are summed up for all valid balls. If this sum clearly favors a side by a high value, it is continued with the next phase or the whole motion will be aborted otherwise.

In the next phase, the robot banks in the foot of the chosen side. The pose of the foot is modified in rotation and position as well as the hip roll rotation. The values are interpolated with a sinus curve over 600ms.

Afterwards the robot waits in the banked in posture until the ball stops rolling, the ball is very close to the robot, or the ball has not been seen for a while. If the ball stops close by, the next motion phase is to move the ball out of the convex hull of the robot. Otherwise, if the ball stopped but not at the foot of the robot, the next motion phase is skipped.

If the ball stopped at the foot of the robot, it has to be moved out of the convex hull of the robots' feet to avoid ball holding, when putting the feet back to their stand positions. For this, a predetermined foot-movement is executed, giving the ball a push with the heel.

At the end both feet positions are brought back to their stand positions, by interpolating from their current position to the stand position, ending the ball taking motion.

This motion can be executed by setting the `takeBall` option of the enum `motion` of the representation *MotionRequest* within the behavior. This motion is unstable when switching from a

full speed walk into it. Therefore, it should only be called when the robot is standing or walking slowly, which might have to be requested earlier.

6.2.5 Motion Selection and Combination

The MotionSelector determines which whole-body motion engine WalkingEngine, BLAME, SpecialActions, GetUpEngine, IndyKickEngine or StandEngine (currently we are using the WalkingEngine for this) is to execute and calculates interpolation ratios in order to switch. In doing so this module takes into account which motion and motion engine is demanded via the *MotionRequest*, which motion and motion engine is currently executed and whether it can be interrupted. Almost all motions can be interrupted but not in all states. The interruptibility of one motion is handled by the corresponding motion engine in order to ensure a motion is not interrupted in an unstable situation.

The MotionCombinator requires the interpolation ratios of each motion engine in execution that are provided by the module MotionSelector. Using these ratios, the joint angles generated by the modules WalkingEngine, BLAME, IndyKickEngine, GetUpEngine, BallTaking and SpecialActions are merged together. The ratios are interpolated linearly. The interpolation time between different motion engines depends on the requested target motion engine.

The MotionCombinator merges the joint angles together to the final target joint angles. When 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.

As the MotionCombinator is the module which finally decides which joint angles are to be set it also performs some emergency actions in case the robot is falling: The head is centered and joint hardness is turned down to 30. If arm motions (cf. 6.2.7) are currently active while falling, the hardness of arm joints is not turned down to allow the motion engine to move the arms to a save position as fast as possible.

6.2.6 Head Motions

Besides the motion of the body (arms and legs), the head motion is handled separately. This task is encapsulated within the separate module HeadMotionEngine for two reasons. The first reason is that the modules WalkingEngine and BLAME manipulate the center of mass. Therefore these modules need to consider the mass distribution of the head *before* its execution in order to compensate for head movements.

The other reason is that the module smoothens the head movement by limiting the speed as well as the maximum angular acceleration of the head. In encapsulating this, the other motion modules do not have to concern about this.

The HeadMotionEngine takes the *HeadAngleRequest* generated by the CameraControlEngine (cf. Sect. 4.1.1) and produces the *HeadJointRequest*.

6.2.7 Arm Motions

In our 2012 code release [26], we introduced arm contact detection to recognize when a robot's arm collides with an obstacle. In this year's system we integrated dynamic arm motions on top of that feature. If a robot detects an obstacle with an arm, it may decide to move that arm out

of the way to be able to pass the obstacle without much interference. In addition, arm motions can be triggered by the behavior control, allowing the robot to move its arm aside even before actually touching an obstacle.

An arm motion is defined by a set of states, which consist of target angles for the elbow joint and the shoulder joint of each arm (similar to special action, cf. Sect. 6.2.2). Upon executing an arm motion, the motion engine interpolates intermediate angles between two states to provide a smooth motion. When reaching the last state, the arm remains there until the engine gets a new request or a certain time runs out. While the arm motion engine is active, its output overrides the default arm motions, which are normally generated during walking.

The ArmMotionEngine, sometimes abbreviated ArME performs all the calculations and interpolations for the desired arm motions. It fills the *ArmMotionEngineOutput*, which is mainly used by the WalkingEngine. As by this, the WalkingEngine is aware of upcoming arm positions. Thereby, it can consider them for balancing calculations during walking. The output is also used by the ArmContactModelProvider to disable checks for arm contact while an arm motion is active.

The representation *ArmMotionRequest* may be filled by the behavior to trigger game state dependent arm motions any time while playing.

Arm motions are configured in *armMotionEngine.cfg*. This file defines the sets of targets as well as some other configuration entries:

actionDelay: Delay (in ms) after which an arm motion can be triggered again by an arm contact.

targetTime: Time (in *Motion* frames) after which an arm should be moved back to its default position (applies only for arm contact triggered motions).

allMotions: Array of different arm motions. The entries with id `useDefault, back`, and `falling` are native motions, which should neither be modified nor should they get another index within the array.

Despite these native motions, you may add further ones by adapting the syntax and choosing a new `id`. This new `id` has to be appended to the end of the enumeration `ArmMotionId` within the representation *ArmMotionRequest*.

An arm motion consists of an array of target angles for the shoulder and the elbow joint. The `WristYaw` and `Hand` joints are not supported, because they are not actuated in most of our robots.

```
// [..] config entries left out
allMotions = [
  {
    // [..] native motions left out
  },{
    id = sampleArmMotion;
    states = [
      {angles=[-2.05, 0.2, 0.174, -0.2];hardness=[80, 80, 80, 80];steps=40;},
      {angles=[-2.11, -0.33, 0.4, -0.32];hardness=[80, 80, 80, 80];steps=40;}
    ];
  }
];
```

Listing 6.1: Example of an arm motion definition

The angle respectively hardness values of each state correspond in the order of occurrence to: `ShoulderPitch`, `ShoulderRoll`, `ElbowYaw`, and `ElbowRoll`. For each hardness entry, the special

value −1 may be used, which means to use the default hardness value for that joint as specified in the file *hardnessSettings.cfg*. The `step` attribute specifies how many *Motion* frames should be used to interpolate intermediate angles between two entries in the `states` array.

Arm motions can be triggered by filling out the representation *ArmMotionRequest*. This is done by either behavior output option *Arm* or *Arms*. The latter will start the chosen motion for both arms. The mentioned representation has four attributes for each arm describing the motion to execute:

motion: The id of the motion to execute. `useDefault` means to use the default arm motions produced by the WalkingEngine.

fast: If set to `true` interpolation will be turned off for this motion.

autoReverse: If set to `true`, the arm motion will be reversed to move the arm back to its default position after a certain amount of time.

autoReverseTime: Time in *Motion* frames after which the arm is moved back to its default position. Only taken into account if `autoReverse` is `true`.

If `autoReverse` is disabled, the arm will stay in its target state (that is the last entry within the motion's `states` array) as long as the `motion` attribute of the *ArmMotionRequest* does not change for this arm.

As mentioned above, there exist three native arm motions, which are treated in a special way by the ArmMotionEngine.

useDefault indicates to move the arms back to their default position and then use the motions generated by the WalkingEngine.

back: Arm motions with this id are started when detecting arm contact (cf. Sect. 4.2.3.3). The arm will be moved to the back to lower the robot's physical width allowing it to slip past an obstacle. The arm will be automatically moved back to its default position after a configurable amount of time.

falling: This motion is executed when the robot is about to fall and there is currently an arm motion different from `useDefault` active. In that case, the arms have to be moved to a safe position as fast as possible in order to avoid falling onto them.

As these arm motions are deeply integrated into the ArmMotionEngine, their index within the `ArmMotionId` enumeration should never be altered to avoid unintended malfunction.

Chapter 7

Technical Challenges

In this chapter, we describe our contribution to the three Technical Challenges of the 2013 SPL competition, i. e. the Open Challenge, the Drop-in Player Challenge, and the Passing Challenge. We won all three of them. Thereby, we also won the whole Technical Challenges competition.

7.1 Open Challenge – Corner Kick

We observed that in contrast to other RoboCup leagues such as the Small Size League, the rules of the Standard Platform League only contain a single set piece: the kickoff. However, set pieces such as throw-in, free kick, in-game penalty kick, and corner kick play an important role in human soccer. Therefore, we think it is time to introduce more set pieces into regular SPL games on our road towards the 2050 goal. As an interesting candidate, we identified the corner kick, as it will not happen too often during a game and therefore does not disturb the flow of a game too much if not done well. However, there is enough to gain or to loose for both teams to invest time in implementing an interesting attack or defense for this set piece.

7.1.1 The Rules

When a robot kicks a ball out over the own ground line, the opponent team is granted a corner kick. The ball will then be placed on the left or right field corner depending on which side of the goal the ball was kicked out. Both teams have 30 seconds to take their positions.

The defending team is allowed to build up a defensive wall while keeping a distance of at least 1.5 m to the ball. One robot of the attacking team takes position to kick the ball in. The other robots of the attacking team also need to keep the same distance as it is depicted in Fig. 7.1.

Once the positioning time is up, a *set* state follows. If robots are standing too close to the ball, they are removed from the game for 30 seconds. Afterwards, the game state is changed to *playing*. No robot is allowed to approach the ball until it has been moved by the kicking robot or 10 seconds have passed, whichever happens first.

7.1.2 Implementation

The main issues in this challenge are the detection of the opponent robots and the positioning of the own robots. Since we already got a pretty solid solution to detect robots (cf. Sect. 4.1.10), we could fully concentrate on the positioning part.

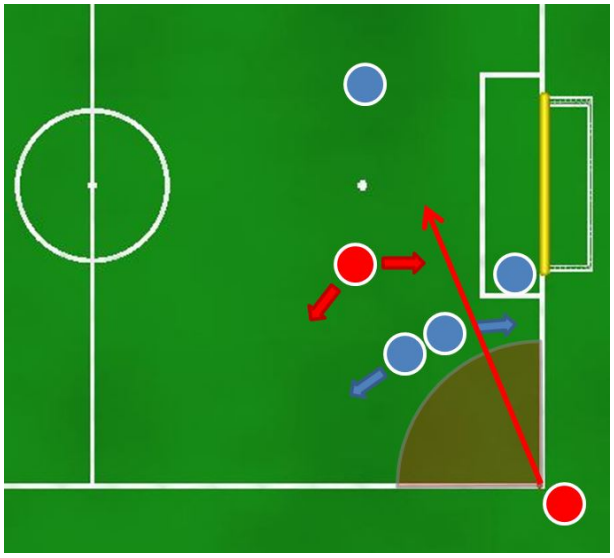


Figure 7.1: Corner kick situation: The red team executes the kick while the blue team defends the goal.

7.1.2.1 Defensive Team

Since the keeper and all robots inside the defensive wall have to look in the direction of the ball to quickly react as soon as the opponent striker shoots the ball, they can not track other opponent robots which try to receive the pass. To solve this problem, another robot was positioned a few meters behind the wall to keep track of opponent movement. Since our presentation only used two opponent robots, the idea was to keep track of the one that ought to receive the pass and change the position of the wall to always be between the striker and the receiver (cf. Fig. 7.2). To ensure that the opponent striker would never be able to score a goal directly, the keeper's position was fixed close to the goal post until the ball has been kicked in by the attacking team.

7.1.2.2 Offensive Team

Since the time to pass the ball – before the opponent team is allowed to attack the striker – is limited, the pass-receiving robot had to move quickly to find a gap in the defensive wall. In our presentation, the offensive team consists of two robots which both look directly at the wall of the defending team. Thereby, they are able to identify the defensive wall using our vision-based obstacle detection. The tactic applied for the receiving player was to move up and down behind the wall, decreasing the chance of the opponent team to intercept the pass by finding a suitable gap.

7.1.3 Discussion of Open Problems

The presented approach performed pretty well at the Open Challenge demonstration at RoboCup 2013 as well as in many previously conducted tests. However, one could see that there are still some issues that have to be solved before it can be used at the actual tournament.

Improve self-localization when looking at the corner of the field. Most robots won't see many field lines or goal posts when a corner kick is performed because they will have to stand near to the corner and/or look in its direction. In addition, there will be much

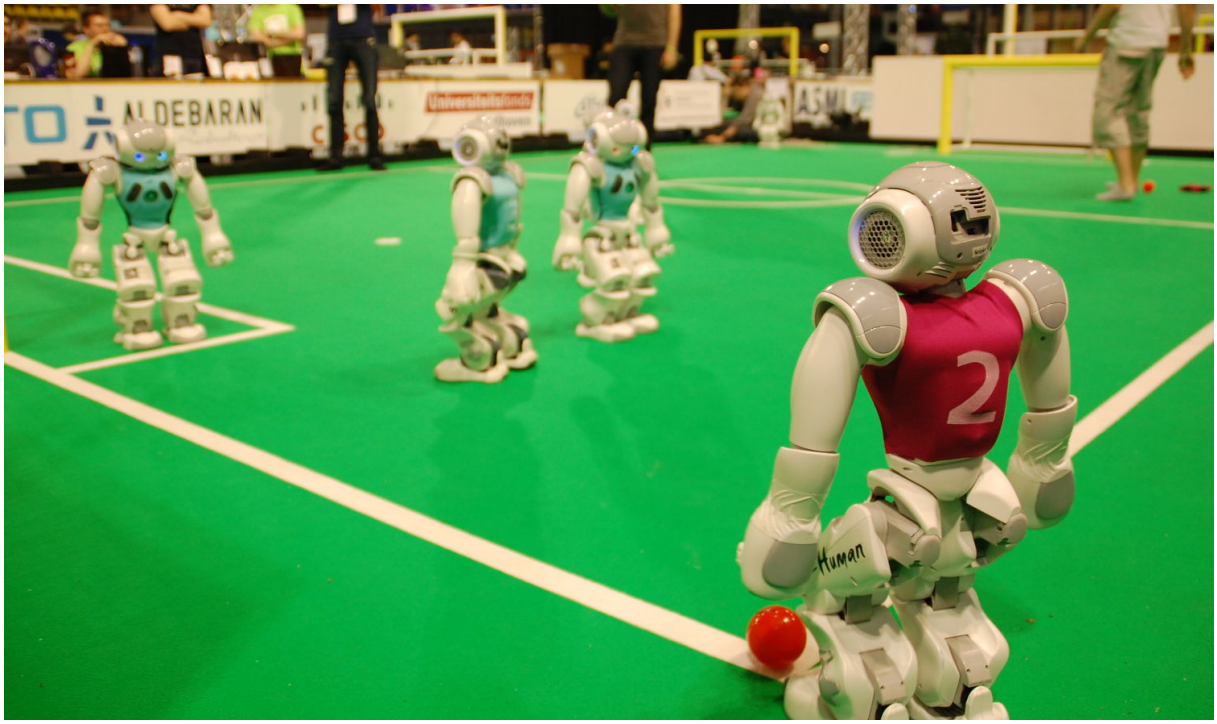


Figure 7.2: Open Challenge demonstration: A robot of the red team is ready to carry out a corner kick. The blue keeper is already in a proper defense position. Two blue field players are about to set up a wall to prevent a pass to the second red player (mostly hidden by the right blue player).

more rivalry about certain positions on the field. So, the robots will push each other very often. In this case, the self-localization will have to be much more stable to ensure that all robots will always find the position they are supposed to be in.

Improve robot detection. As explained above, in a real game, there will be more robots fighting for different positions. Hence, robots will stand very close to each other and it will be difficult to distinguish them from each other. Additionally, they will conceal each other from the cameras of other robots. Thus, there has to be a reliable model for tracking opponent robots that are seen pretty rarely.

More robots. Our demonstration only included two robots in the offensive team. In a real game, there would be at least one other robot ready to receive a pass from the striker. In this case, the defending team would be required to keep track of multiple robots and to adjust its defensive positions in a way that would at least cover opponent robots in particularly threatening positions.

However, we think that these problems can be solved and that the introduction of this new set piece would be a good addition to the SPL because it would force the teams to improve a variety of modules such as self-localization and robot detection.

7.2 Drop-In Player Challenge

The idea behind the Drop-In Player Challenge was for robots from different teams to be able to play together. Moreover they should be able to be good teammates. For that purpose, a

common communication protocol was given. In general, the rules for the challenge were the same as the rules of a normal game. There only was a single exception concerning the goalkeeper. In contrast to the goalkeeper being the robot with the player number one, the first robot that entered the goal box became the goalkeeper instead.

Implementing these requirements was quickly done since our behavior is already designed for team-play. Three modules were implemented, that handle sending and receiving data to and from other robots via the protocol provided, and the conversion between data of the protocol and the internal representations. Since the protocol does not provide all data our robots normally exchange, the behavior had to be slightly adapted to cope with the missing information.

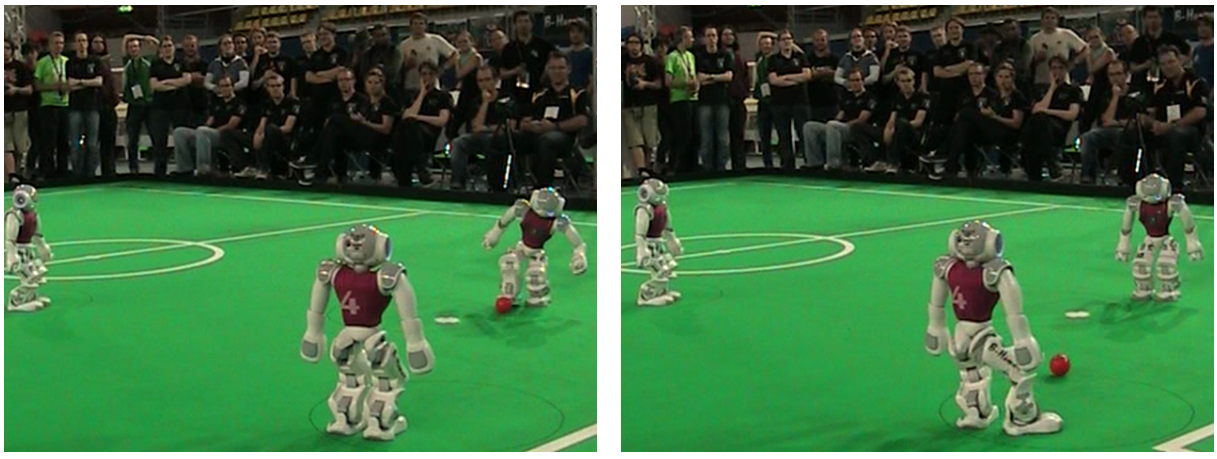
The goalkeeper rule was handled the following way. If a robot is at a position near the own goal in the *ready* state in the beginning of the game, it will try to become the goalkeeper. Should the robot get an illegal defender penalty, because another robot entered the goal box first, it will discard that attempt.

7.3 Passing Challenge

The 2013 Passing Challenge was about circulating the ball among three robots. There were three circular regions defined on the field. A pass was considered successful if the ball was kicked from within such a circle and received by a different robot in a different circle.

One particularly difficult aspect of this challenge was that the circle positions were unknown in advance, so no team could model special kicking motions to kick over just the right distance between a pair of circles. We solved this problem by generating the kick motions completely on the robot, such that the ball was kicked with just the right strength for each passing situation. To generate and execute the dynamic kick motions, we used a newly developed motion engine, which has been published separately by Wenk and Röfer [31]. Figure 7.3a shows the engine in action.

Since the competition took place on a field not perfectly flat with robots neither perfectly modeled nor perfectly calibrated, the kick strength and the execution of the kick were not always accurate enough to reach the rather small circles exactly. In order to overcome that problem, we implemented a small ball taking motion engine that stops the ball based on the



(a) (b)

Figure 7.3: (a) Execution of the dynamic kick motion adjusted to the pass target and ball position (b) Ball taking movement executed predictively correct

ball position and velocity. Thus, the main task of this engine, which is described in detail in Sect. 6.2.4, was to ensure that the ball is not accidentally kicked out of the circle. A ball taking move is depicted in Fig. 7.3b.

The final part of our implementation is the behavior. It mainly ensures that at any time, only a single robot is responsible to kick the ball. For that reason, we introduced two different roles: `receiver` and `kicker`. At the beginning, each robot is a `receiver` and performs a quick spin to observe the environment in order to optimize the localization.

As soon as the ball enters the field, all three robots agree via team communication, which is nearest to the ball. Once that is clarified, only the closest robot changes its role to `kicker` and chooses one of the two remaining `receivers` as the pass target. Next, the `receiver` informs the chosen pass taker via team communication about the planned pass, which causes the chosen `receiver` to adjust to the ball position. As soon as the `receiver` has finished the adjusting process, it informs the `kicker` about that, which causes the `kicker` to perform the actual kicking movement as shown in Fig. 7.3a. While the ball is rolling, the `receiver` decides to use the ball taking motion engine in case the ball trajectory and speed is not precise enough, which is depicted in Fig. 7.3b. As soon as the ball has stopped, the `kicker` changes its role back to `receiver` allowing the previous `receiver` to become the `kicker` now.

Chapter 8

Tools

The following chapter describes B-Human's simulator, SimRobot, as well as the B-Human User Shell (*bush*), which is used during games to deploy the code and the settings to several NAO robots at once. As the official *GameController* was also developed by B-Human, it is also briefly described at the end of this chapter, as well as the integration of the *libgamectrl* that comes with the *GameController* into the B-Human system.

8.1 SimRobot

The B-Human software package uses the physical robotics simulator SimRobot [15, 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.

8.1.1 Architecture

Four dynamic libraries are created when SimRobot is built. These are *SimulatedNao*, *SimRobotCore2*, *SimRobotHelp*, and *SimRobotEditor* (cf. Fig. 8.1)[1].

SimRobotCore2 is an enhancement of the previous simulation core. It is the most important part of the SimRobot application, because it models the robots and the environment, simulates sensor readings, and executes commands given by the controller or the user. The core is platform-independent and it is connected to a user interface and a controller via a well-defined interface.

With the new core, the scene description language RoSiML [3] was extended and some new elements were included to improve the visualization and physics of the robots.

The library *SimulatedNao* is in fact the controller that consists of the two projects *SimulatedNao* and *Controller*. *SimulatedNao* creates the code for the simulated robot. This code is linked together with the code created by the *Controller* project to the *SimulatedNao* library. In the scene files, this library is referenced by the `controller` attribute within the `Scene` element.

The other two libraries contain add-ons, which can be loaded on demand. *SimRobotHelp* contains the help menu, which is shown by clicking *Help* in the menu and then *View Help*. Unfortunately, the help system is currently both highly incomplete and outdated. *SimRobotEditor* contains a simple editor with XML syntax highlighting. This editor is intended for modifying the scene description files.

[1] The actual names of the libraries have platform-dependent prefixes and suffixes, i. e. *.dll*, *.dylib*, and *lib .so*.

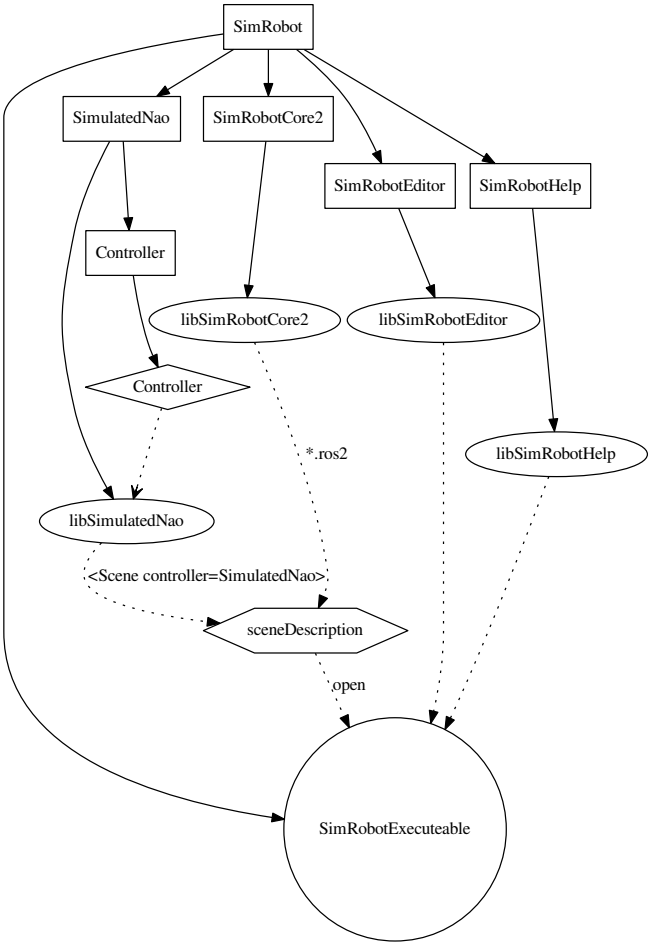


Figure 8.1: This figure shows the most important libraries of *SimRobot* (excluding foreign libraries). The rectangles represent the projects, which create the appropriate library. Dynamic libraries are represented by ellipses and the static one by a diamond. Note: The static library will be linked together with the *SimulatedNao* code. The result is the library *SimulatedNao*.

8.1.2 B-Human Toolbar

The B-Human toolbar is part of the general SimRobot toolbar which can be found at the top of the application window (see Fig. 8.2). Currently there are two buttons to command the robot to either stand up or sit down. They overwrite the *MotionRequest* of the selected robots and set either *motion=stand* or *motion=sitDown*. A third button is used to simulate a press on the chest button.

8.1.3 Scene View

The scene view (cf. Fig. 8.3 right) appears if the *scene* is opened from the scene graph, e. g., by double-clicking on the entry *RoboCup*. The view can be rotated around two axes, and it

Figure 8.2: This figure shows the three buttons from the *BHToolBar*.

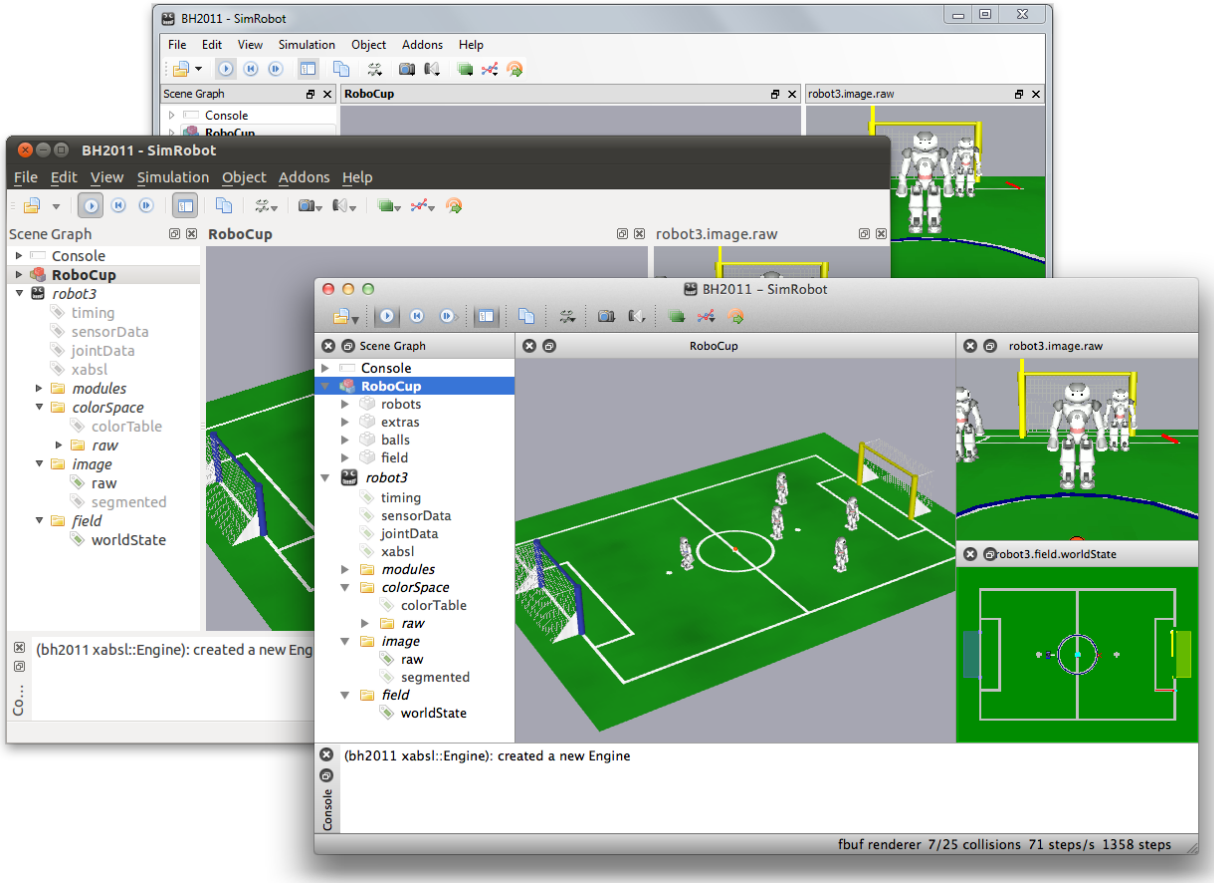


Figure 8.3: SimRobot running on Windows, Linux, and Mac OS X. The left pane shows the scene graph, the center pane shows a scene view, and on the right there are an image view and a field view. The console window is shown at the bottom.

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 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. 8.1.5).

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

8.1.4 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* (Sect. 3.5). 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 fourteen kinds of information views, which are structured here into the five categories *cognition*, *behavior control*, *sensing*, *motion control*,

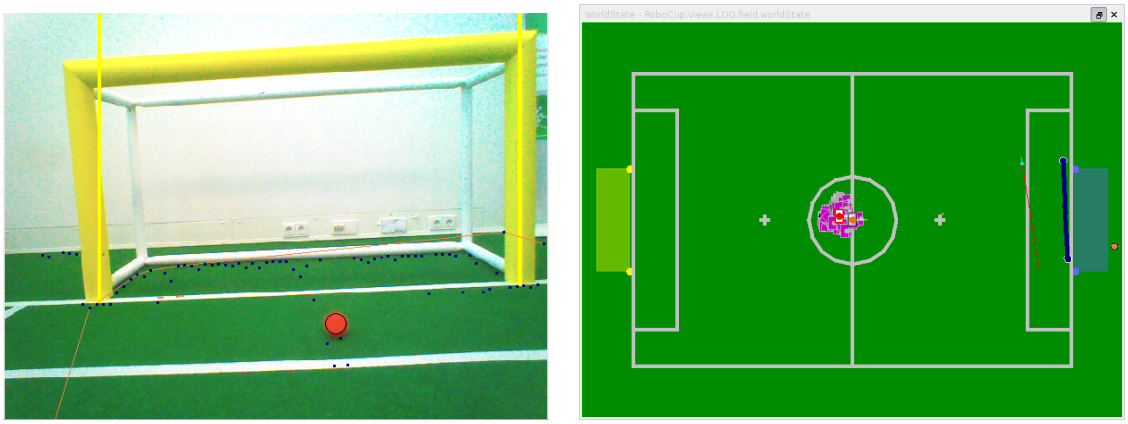


Figure 8.4: Image view and field view with several debug drawings

and *general debugging support*. All information views can be selected from the scene graph (cf. Fig. 8.3 left).

8.1.4.1 Cognition

Image Views

An image view (cf. left of Fig. 8.4) displays debug information in the coordinate system of the camera image. It is defined by a background image, an optional switch for JPEG compression, an optional switch for segmentation, an optional flag for the background image and a name using the console command *vi*. More formally its syntax is defined as:

```
vi background_image [jpeg] [segmented] [upperCam] name
```

The background image can be any image that has been defined using the `DECLARE_DEBUG_IMAGE` macro. The following debug images are currently defined:

image: The image provided by the camera.

corrected: Similar to *image*, but without rolling shutter effects.

horizonAligned: Similar to *image*, but aligned to the horizon.

thumbnailDI A down scaled compressed version of the image.

none: This is actually an option not to show any background image.

The switch *jpeg* will cause the NAO to compress the images before sending them to SimRobot. This might be useful if SimRobot is connected with the NAO via Wifi. The switch *segmented* will cause the image view to classify each pixel and draw its color class instead of the pixel's value itself. The default is to show data based on the images taken by the lower camera. With *upperCam*, the upper camera is selected instead.

The console command *vid* adds the debug drawings. 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
vid raw representation:BodyContour
```

Image views are linked with a toolbar providing some useful features. The following table provides a details description of the toolbar.

Allows moving the robot's head by hand.

Shows the image of the lower camera in the current view.

Shows the image of the upper camera in the current view.


Toggle button: Allows moving the robot's head by clicking into the image view.


Saves the current camera/color calibration and updates the *ColorReference* of the robot.

Image views created with the flag *segmented* additionally provide features for calibrating colors (cf. Sect. 2.8.4). Each of the buttons listed below will cause the segmented image view to only draw pixels classified as the corresponding color class.


field color


goal color


ball color


red jersey color


blue jersey color


field line color


shadow/misc color


all colors

Furthermore each button – except for *all colors* and *black*, since it is not configurable – will show the current settings of the corresponding color class in the *colorCalibration* view.

Color Calibration View

The *colorCalibration* view provides direct access to the parameter settings for all color classes but black. The current settings of a color class can be achieved by selecting the class in the segmented image view as described in Section 8.1.4.1. The parameters of a color class can be changed by moving the sliders. All color classes but white use range selectors for hue, saturation, and value. Hue's interval is [0.0, 2.0π] while saturation's and value's intervals are [0.0, 1.0]. White itself uses threshold selectors for red, blue, and the sum of red and blue with an interval of [0, 254] for red and blue and [0, 508] for the sum of both (cf. Sect. 4.1.4). Instead of using the sliders, one can directly input the values into the text boxes. Figure 8.5 shows an example calibration for green and orange. Since the range selection for hue is periodic, the sliders allow defining a minimum higher than the maximum, which is typically for the ball color.

Changing the parameters of a color will only be applied on the local calibration. To update the settings for the NAO, use the save button described in Section 8.1.4.1.

Color Space Views

Color space views visualize image information in 3-D (cf. Fig. 8.6). 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:

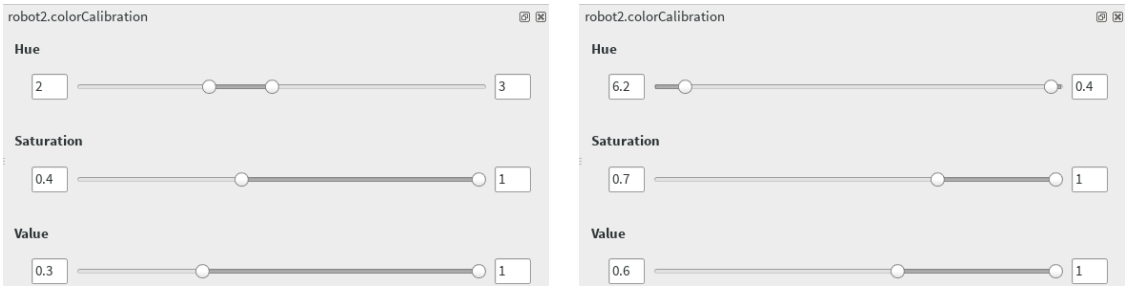


Figure 8.5: An example calibration for green (left) and orange (right)

Color Reference: This view displays the current color classification of a single color in YCbCr space (cf. Fig. 8.6 bottom right).

Image Color Space: This view displays the distribution of all pixels of an image in a certain color space (*HSI*, *RGB*, or *YCbCr*). It can be displayed by selecting the entry *all* for a certain color space in the scene graph (cf. Fig. 8.6 top right).

Image Color Channel: This view displays an image while using a certain color channel as height information (cf. Fig. 8.6 left).

While the color reference 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

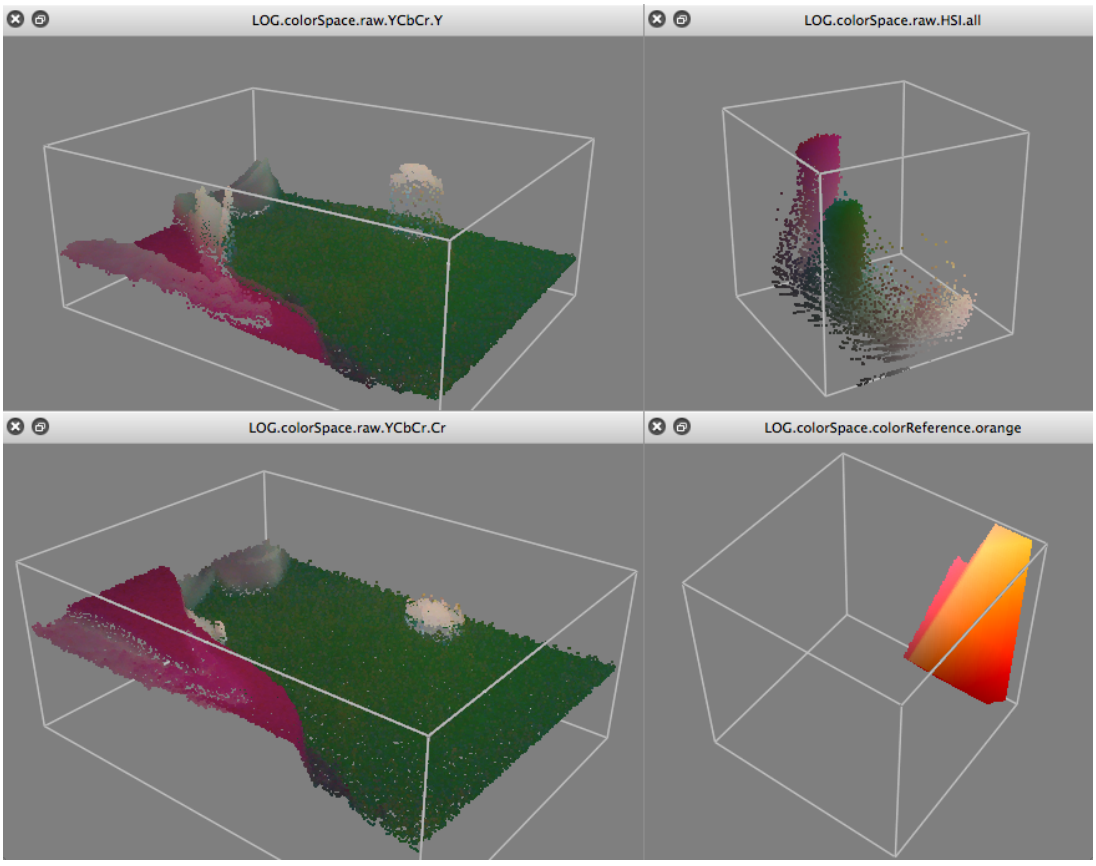


Figure 8.6: Color channel views (left), image color space view (top right), and color reference view (bottom right)

of views for the camera image under the name *raw*, the following command has to be executed:

```
v3 image raw
```

Field Views

A field view (cf. right of Fig. 8.4) displays information in the system of coordinates of the soccer field. The command to create and manipulate it is defined similar to the one for the image views. For instance, the view *worldState* is defined as:

```
# field views
vf worldState
vfd worldState fieldLines
vfd worldState fieldPolygons
vfd worldState representation:RobotPose
vfd worldState module:FieldCoverageProvider:FieldView

# 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 representation:MotionRequest
vfd worldState representation:FieldBoundary:Field

# 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. Double clicking the view resets it to its initial position and scale.

8.1.4.2 Behavior Control

Option Graph View

The *option graph view* (cf. Fig. 8.7) can be found under *Docs* in the scene graph. It is a static view that only displays a graph with all the options of a CABSL behavior. It has the same name as the behavior the option graph of which it displays, i. e. currently *BehaviorControl2013*. The colors of the options visualize to which part of the behavior each option belongs:

Skills are shown in yellow.

GameControl options are shown in red.

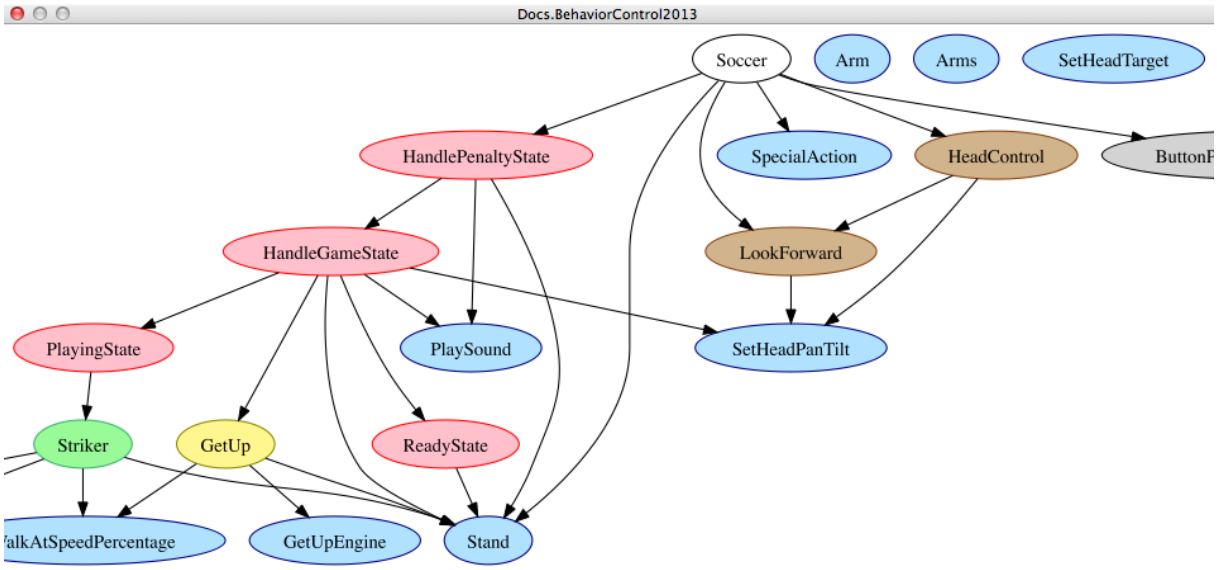


Figure 8.7: Option graph view

HeadControl options are shown in brown.

Output options are shown in blue.

Tools are shown in gray.

Roles are shown in green.

Everything else is shown in white.

The 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.

Activation Graph View

The *activation graph view* (cf. left of Fig. 8.8) shows all the options and states that are currently active in the behavior. It also displays how long they have already been active continuously.

8.1.4.3 Sensing

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. 8.8). To display this information, the following debug requests must be sent:

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

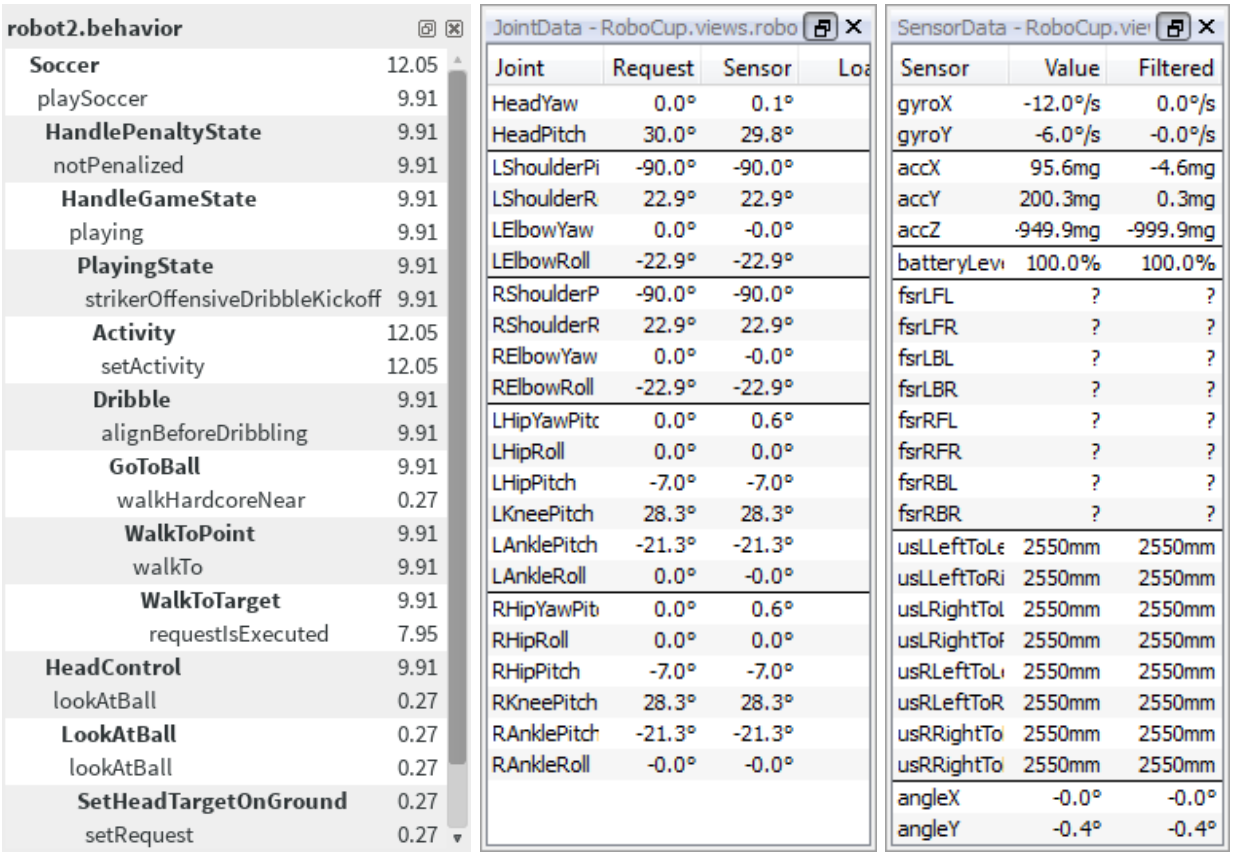


Figure 8.8: Behavior view, joint data view, and sensor data view

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. 8.8). To display this information, the following debug requests must be sent:

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

Foot View

The *foot view* (cf. Fig. 8.9) displays the position of the Zero Moment Point (ZMP) within the support foot while the IndyKickEngine performs a kick. The view actually displays the contents of the representation *RobotBalance*. Before the view can show it, filling and sending this representation must be activated:

```
mr RobotBalance IndyKickEngine
dr representation:RobotBalance
```

8.1.4.4 Motion Control

Kick View

The basic idea of the kick view shown in Figure 8.10 is to visualize and edit basic configurations of motions for the dynamic motion engine described in [18]. In doing so the central element of

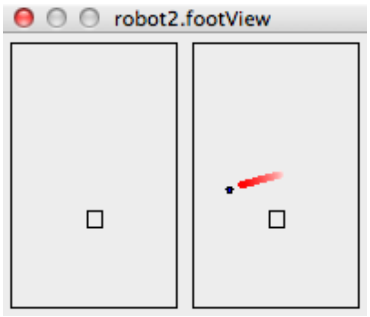


Figure 8.9: Foot view. The black dot shows the expected position of the ZMP. In red, the preview of future ZMP positions is displayed.

this view is a 3-D robot model. Regardless of whether the controller is connected to a simulated or a 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 three-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 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. 8.10 at E) and drag it to the desired position. In order to visualize the current dragging plane, a light green area (cf. Fig. 8.10 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. 8.10 at C).

1-D/2-D View: In some cases a movement only happens in the 2-dimensional or 1-dimensional space (for example: Raising 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 context menu, which opens by clicking with the right mouse button and choosing *Display 1D Views* or *Display 2D Views*. This only works within the left area, where the control buttons are (cf. Fig. 8.10 left of D). By clicking with the right mouse button within the 3-D view, the context menu for choosing the drag plane appears. The

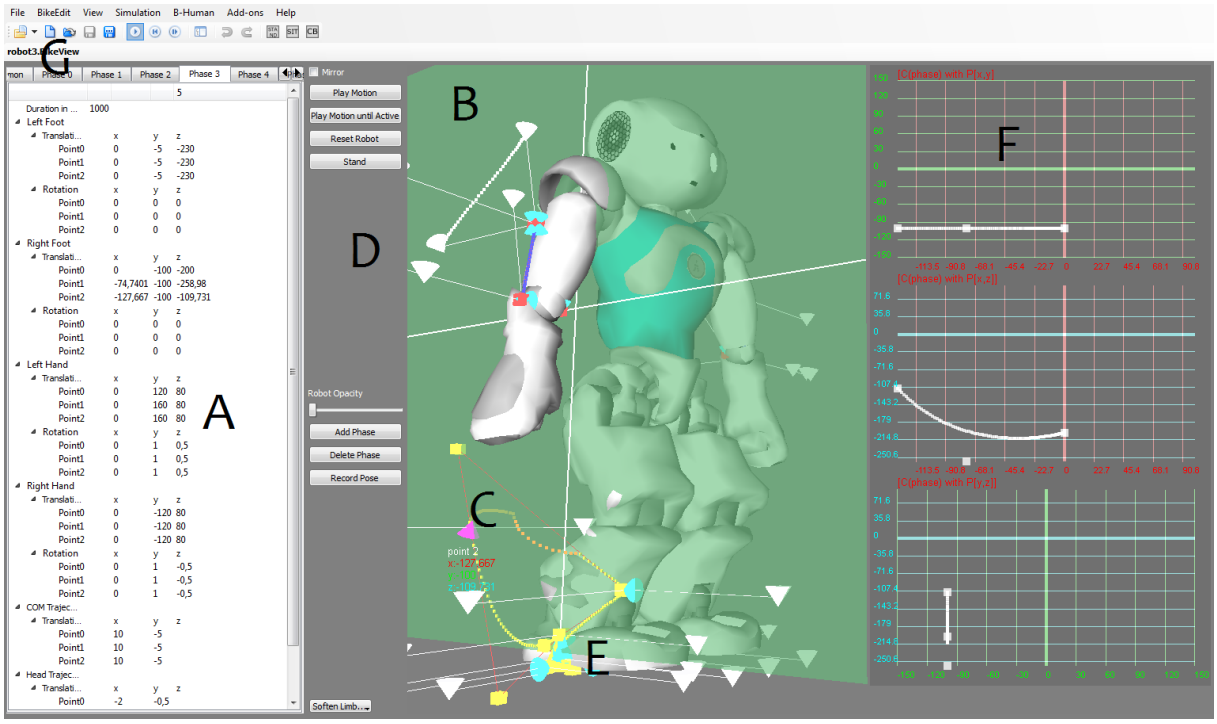


Figure 8.10: The kick view. *A* marks the editor view, *B* denotes the drag and drop plane, *C* points at a clipped curve, *D* tags the buttons that control the 3D-View, e. g. play the motion or reset the robot to a standing position, *E* labels one of the control points, *F* points at the subviews and *G* points at the tool bar, where a motion can be saved or loaded.

second opportunity to display a sub view is by clicking at the *BikeEdit* entry in the menu. So the same menu, which appears as context menu, will be displayed.

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. 8.11) 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 $P_i = \begin{pmatrix} c_{x_i} \\ c_{y_i} \end{pmatrix}$, $P_i = \begin{pmatrix} c_{x_i} \\ c_{z_i} \end{pmatrix}$ or $P_i = \begin{pmatrix} c_{y_i} \\ c_{z_i} \end{pmatrix}$.
The 1-D sub views (cf. Fig. 8.11) 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 $P_i = c_{z_i}$.

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

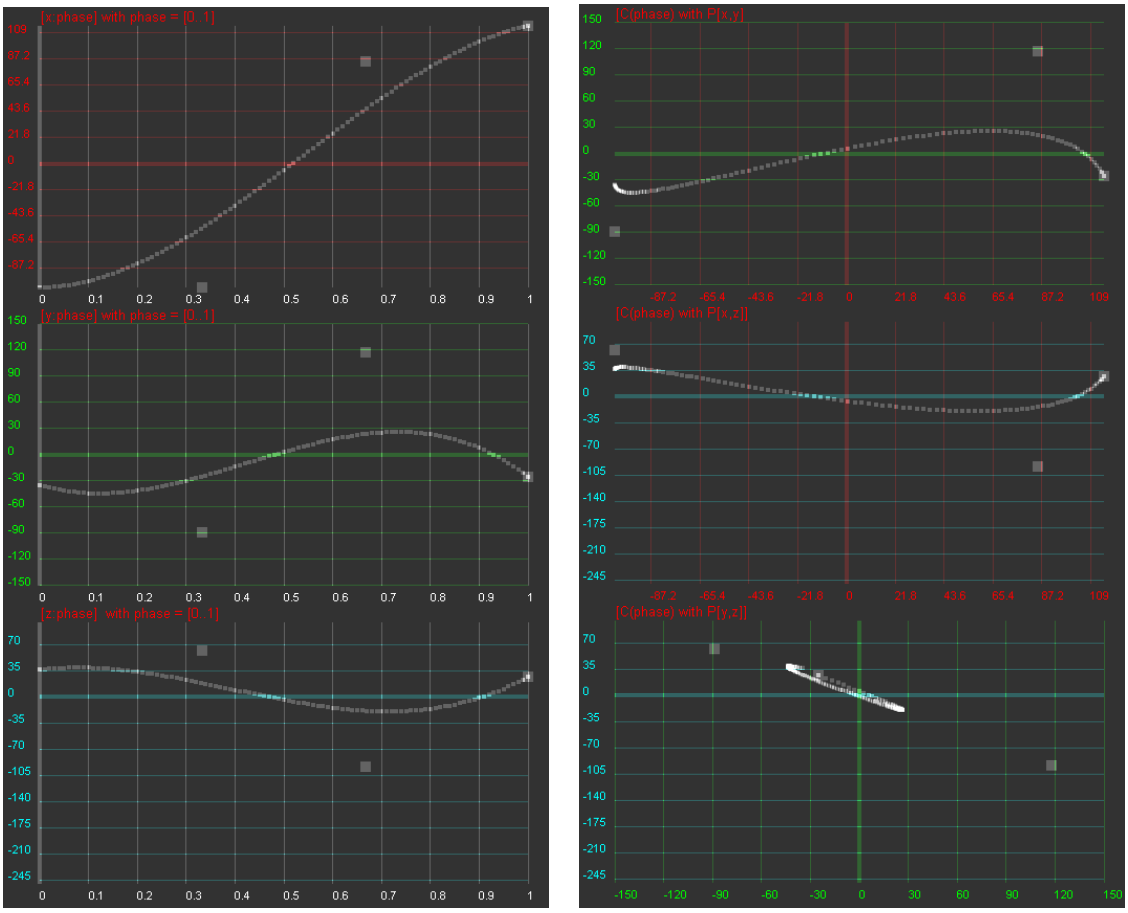


Figure 8.11: Left: 1-D sub views. Right: 2-D sub views

drag and drop, to add new phases, or to delete phases.

To save or load a motion the kick view has to be the active view, so some buttons to do this appearing at the tool bar (cf. Fig. 8.10 at *G*).

8.1.4.5 General Debugging Support

Module Views

Since all the information about the current module configuration can be requested from the robot control program, it is possible to generate a visual representation automatically. The graphs, such as the one that is shown in Figure 8.12, are generated by the program *dot* from the *Graphviz* package [2]. 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 were 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*

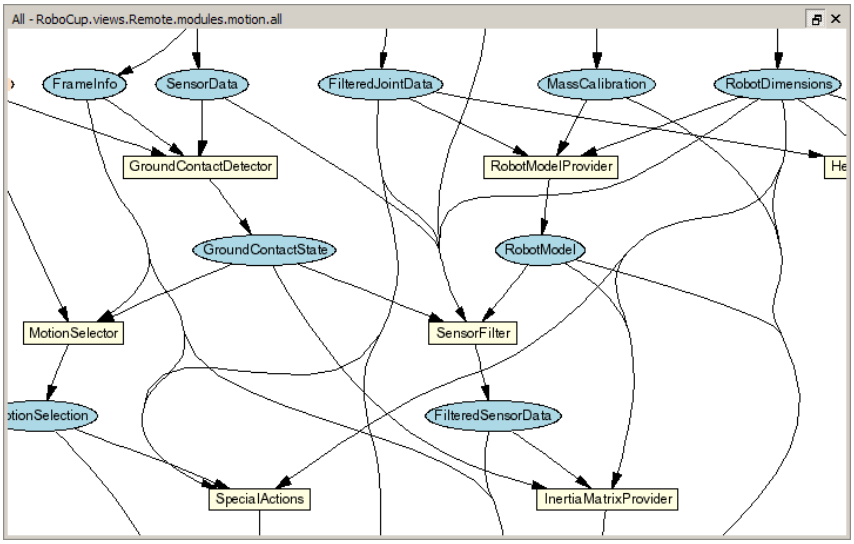


Figure 8.12: The module view shows a part of the modules in the process *Motion*.

buttons. It can also be dragged around with the left mouse button.

Plot Views

Plot views allow plotting data sent from the robot control program through the macro `PLOT` (cf. Fig. 8.13 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. 8.1.6.3).

For instance, the view on the left side of Figure 8.13 was defined as:

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

Timing View

The timing view displays statistics about all currently active stopwatches in a process (cf. Fig. 8.13 right). It shows the minimum, maximum, and average runtime of each stopwatch in milliseconds as well as the average frequency of the process. All statistics sum up the last 100

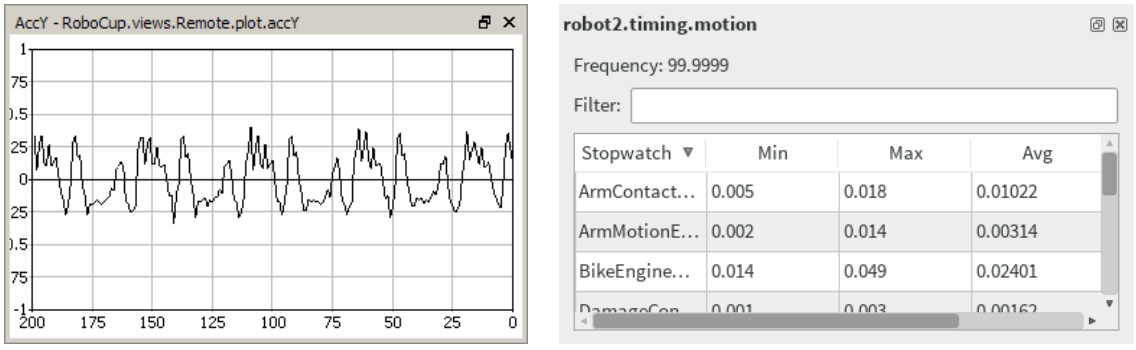


Figure 8.13: Plot view and timing view

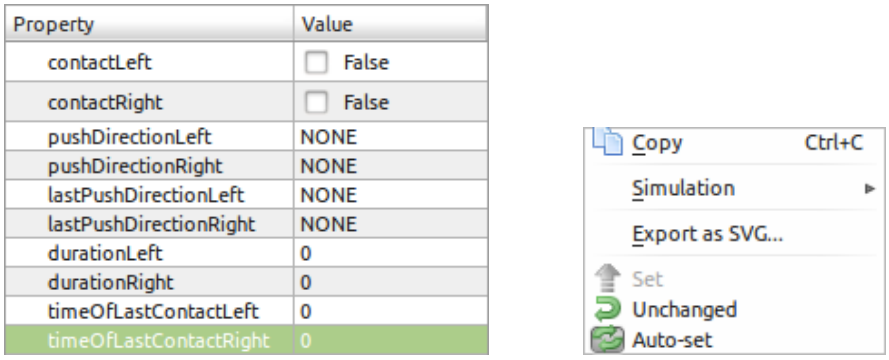


Figure 8.14: The data view can be used to remotely modify data on the robot.

invocations of the stopwatch. Timing data is transferred to the PC using debug requests. By default the timing data is not sent to the PC. Execute the console command *dr timing* to activate the sending of timing data. 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.

Data View

SimRobot offers two console commands (*get* & *set*) to view or edit anything that the robot exposes using the MODIFY macro. While those commands are enough to occasionally change some variables, they can become quite annoying during heavy debugging sessions.

For this reason, we introduced a new dynamic data view. It displays modifiable content using a property browser (cf. Fig. 8.14 left). Property browsers are well suited for displaying hierarchical data and should be well known from various editors e. g. Microsoft Visual Studio or Eclipse.

A new data view is constructed using the command *vd* in SimRobot. For example *vd representation:ArmContactModel* will create a new view displaying the *ArmContactModel*. Data views can be found in the data category of the scene graph.

The data view automatically updates itself three times per second. Higher update rates are possible, but result in a much higher CPU usage.

To modify data, just click on the desired field and start editing. The view will stop updating itself as soon as you start editing a field. The editing process is finished either by pressing enter or by deselecting the field. By default, modifications will be sent to the robot immediately. This feature is called the auto-set mode. It can be turned off using the context menu (cf. Fig. 8.14 right). If the auto-set mode is disabled, data can be transmitted to the robot using the *send* item from the context menu.

Once the modifications are finished, the view will resume updating itself. However you may not notice this since modification freezes the data on the robot side. To reset the data, use the *unchanged* item from the context menu. As a result, the data will be unfrozen on the robot side and you should see the data changing again.

8.1.5 Scene Description Files

The language of scene description files is an extended version of RoSiML [3]. To use this new version and the new SimRobotCore2, the scene file has to end with *.ros2*, such as *BH2013.ros2*. In the following, the most important elements, which are necessary to add robots, dummies,

and balls, are shortly described (based upon *BH2013.ros2*). For a more detailed documentation see Appendix A.

<Include href="...">: First of all the descriptions of the NAO, the ball and the field are included. The names of include files end with *.rsi2*.

<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. In the tag *Body*, the attribute *ref* specifies which NAO model should be used and *name* sets the robot name in the scene graph of the simulation. Legal robot names are "robot1" ... "robot10", where the first five robots are assumed to play in the blue team (with player numbers 1...5) while the other five play in the red team (again with player numbers 1...5). The standard color of the NAO's jersey is set to blue. To set it to red, use <Set name="NaoColor" value="red"> within the tag *Body*.

<Compound name="extras">: This compound 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* , but the referenced model has to be changed from "NaoDummy" to "Nao".

A lot of scene description files can be found in *Config/Scenes*. There are two types of scene description files: the ones required to simulate one or more robots, and the ones that are sufficient to connect to a real robot.

8.1.5.1 SimRobot Scenes used for Simulation

BH2013: A simple scene that simulates only a single robot and five dummies that do not move.

BikeScene: Simulates a single robot on an empty field. The kick view (cf. Sect. 8.1.4.4) is enabled by default.

Game2013: Full simulation of a five vs. five game.

Game2013Fast: Simulation of a five vs. five game without perception. Things such as the *RobotPose* and the *BallModel* are directly provided by the simulator. This scene can be used to test behavior on slow computers.

ReplayRobot: Used to replay log files (cf. Sect. 3.7.5).

8.1.5.2 SimRobot Scenes used for Remote Debugging

BikeSceneRemoteWithPuppet: Does the same as the *BikeScene* but with a remote robot instead of simulated one. The IP address or a list of possible IP addresses of the remote robot to connect to is configured in the script *Config/Scenes/connect.con*.

RemoteRobot: Connects to a remote robot and displays images and data. This scene is also configured through the script *Config/Scenes/connect.con*.

TeamComm3D: Visualizes the data of the team communication in a joint 3-D view. The UDP ports used for up to two teams and their broadcast addresses are configured in the script *Config/Scenes/teamPort.con*.

TeamComm: Visualizes most of the information that the robots send about themselves in separate 2-D views. This scene is also configured through the script *Config/Scenes/teamPort.con*.

8.1.6 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).

8.1.6.1 Initialization Commands

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

sl <name> <file> [<offset>]
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. The third parameter is optional, it defines the offset in milliseconds to a video and other logs, so that all logs and the video run synchronously.

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.

8.1.6.2 Global Commands

ar | on | off
Enable or disable the automatic referee.

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>
Prints 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.

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 10 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.

8.1.6.3 Robot Commands

ac both | upper | lower
Change the process that provides drawings in the field and 3-D views.

ac <image> upper | lower
Change the camera that is used in the specified image view.

bc <red%> <green%> <blue%>
Defines the background color for 3-D views. The color is specified in percentages of red, green, and blue intensities. All parameters are optional. Missing parameters will be interpreted as 0%.

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. 4.1.2.1.

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 60. 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.

dr ? [<**pattern**>] | **off** | <**key**> **(off | on)**
Sends a debug request. B-Human uses debug requests to switch *debug responses* (cf. Sect. 3.6.1) 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 or deactivated. They are deactivated by default. Specifying just *off* as only parameter returns to this default state. 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 40. 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...$8 can be used as placeholders for up to eight 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> <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> [ ]
Set axis maximum speed and ignore threshold for command *jc motion <num>*. *axis* is the number of the joystick axis to configure (1...8). *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 itself, e. g., to compensate for the bad calibration of a joystick.

log ? mr [list] | start | stop | pause | forward [image] | backward [image] | fast_forward | fast_backward | repeat | goto <number> | clear | (keep | remove) <message> {<message>} | (load | save | saveImages [raw]) <file> | saveTiming <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. 8.1.6.1). The different parameters have the following meaning:

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

mr [list]
Sets the provider of all representations from the log file to CognitionLogDataProvider. If **list** is specified the module request commands will be printed to the console instead.

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.

fast_forward | fast_backward
Jump 100 steps forward or backward.

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 in the format defined by the extension of the filename specified. If the extension is omitted, *.bmp* is used. The files saved contain either RGB or YCbCr images. The latter is the case if the option *raw* is specified.

saveTiming <file>
Creates a comma separated list containing the data of all stopwatches for each frame. <file> can either be an absolute or a relative path. In the latter case, it is relative to the directory *Config*. If no extension is specified, *.csv* is used.

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 | disable | enable | 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.

pr none | ballHolding | playerPushing | inactivePlayer | illegalDefender | leavingTheField | playingWithHands | requestForPickup
Penalizes a simulated robot with the given penalty or unpenalizes it, when used with *none*. When penalized, the simulated robot will be moved to the sideline, looking away from the field. When unpenalized, it will be turned, facing the field again, and moved to the sideline that is further away from the ball.

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
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 60 images per second from the robot.

queue
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
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 */home/nao/Config/logfile.log*. No messages will be sent to the PC until another queue fill request is sent.

si reset | (upper | lower) [**number**] [<**file**>]
Saves the raw image of a robot. The image will be saved as bitmap file. If no path is specified, *Config/raw_image.bmp* will be used as default option. If *number* is specified, a number is appended to the filename that is increased each time the command is executed. The option *reset* resets the counter.

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. 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.

save ? [<**pattern**>] | <**key**> [<**path**>]
Save debug data to a configuration file. The keys supported can be queried using the question mark. An additional pattern filters the output. If no path is specified, the name of the configuration file is looked up from a table, and its first occurrence in the search path is overwritten. Otherwise, the path is used. If it is relative, it is appended to the directory *Config*.

v3 ? [<**pattern**>] | <**image**> [**jpeg**] [<**name**>]
Adds a set of 3-D color space views for a certain image (cf. Sect. 8.1.4.1). 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. 8.1.4.1). 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**] [**upperCam**] [<**name**>] [**gain** <**value**>]
Adds an image view (cf. Sect. 8.1.4.1). 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 default is to show data based on the images taken by the lower camera. With *upperCam*, the upper camera is selected instead. The next 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. With the last parameter the image gain can be adjusted, if no gain is specified the default value will be 1.0. 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 mark 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. 8.1.4.5). 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.

vd <debug data> (on | off)
Show debug data in a window or disable the updates of the data. Data views can be found in the data category of the scene graph. Data views provide the same functionality as the *get* and *set* commands (see above). However they are much more comfortable to use.

wek Sends walking engine kicks to the robot.

8.1.6.4 Input Selection Dialog

In scripting files, it is sometimes necessary to let the user choose between some values. Therefore we 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 8.15 shows the corresponding dialog.

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

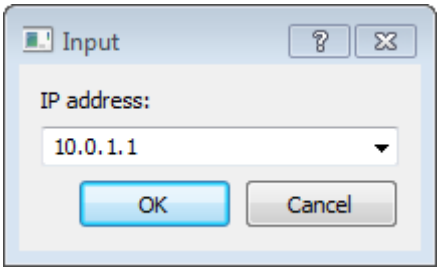


Figure 8.15: A dialog for selecting an IP address

8.1.7 Recording an Offline Log File

To record a log file, the robot shall at least send images, camera info, joint data, sensor data, key states, odometry data, the camera matrix, and the image coordinate system. The following 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 cursor 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 both the IP address in the second line and the filename behind the line *log save* have to be changed.

```
# connect to a robot
sc Remote 10.1.0.101
```

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

# print some useful commands
echo log start
echo log stop
echo log save <filename>
echo log clear
```

8.2 B-Human User Shell

The B-Human User Shell (`bush`) accelerates and simplifies the deployment of code and the configuration of the robots. It is especially useful when controlling several robots at the same time, e. g., during the preparation for a soccer match.

8.2.1 Configuration

Since the *bush* can be used to communicate with the robots without much help from the user, it needs some information about the robots. Therefore, each robot has a configuration file *Config/Robots/<RobotName>/network.cfg*, which defines the name of the robot and how it can be reached by the *bush*.[2] Additionally you have to define one (or more) teams which are arranged in tabs. The data of the teams is used to define the other properties, which are required to deploy code in the correct configuration to the robots. The default configuration file of the teams is *Config/teams.cfg* which can be altered within the *bush* or with a text editor. Each team can have the configuration variables shown in Table 8.1.

8.2.2 Commands

The *bush* supports two types of commands. There are local commands (cf. Tab. 8.2) and commands that interact with selected robot(s) (cf. Tab. 8.3). Robots can be selected by checking their checkbox or with the keys *F1* to *F10*.

8.2.3 Deploying Code to the Robots

For the simultaneous deployment of several robots the command *deploy* should be used. It accepts a single optional parameter that designates the build configuration of the code to be deployed to the selected robots. If the parameter is omitted the default build configuration of the currently selected team is used. It can be changed with the drop-down menu at the top of the bush user interface.

Before the *deploy* command copies code to the robots, it checks whether the binaries are up-to-date. If needed, they are recompiled by the *compile* command, which can also be called

[2] The configuration file is created by the script *createRobot* described in Sect. 2.4.2.

Entry	Description
number	The team number.
port	The port, which is used for team communication messages. This entry is optional. If this value is omitted, the port is generated from the team number.
location	The location, which should be used by the software (cf. Sect. 2.9). This entry is optional. It is set to `Default` if it is omitted.
color	The team color in the first half. This entry is optional. It is only required if no game controller is running, which overwrites the team color.
wlanConfig	The name of the configuration file, which should be used to configure the wireless interface of the robots.
buildConfig	The name of the configuration, which should be used to deploy the NAO code (cf. Sect. 2.2).
players	The list of robots the team consists of. The list must have ten entries, where each entry must either be a name of a robot (with an existing file *Config/Robots/<RobotName>/network.cfg*), or an underscore for empty slots. The first five robots are the main players and the last five their substitutes.

Table 8.1: Configuration variables in the file *Config/teams.cfg*

Command	Parameter(s) Description
compile	[<config> [<project>]] Compiles a project with a specified build configuration. The default is Develop Nao.
exit	 Exit the bush.
help	 Print a help text with all commands.

Table 8.2: General bush commands.

independently from the *deploy* command. Depending on the platform, the *compile* command uses *make*, *xcodebuild*, or *MSBuild* to compile the binaries required. On OS X, you have to manually call *generate* to keep the project up-to-date (cf. Sect. 2.3).

After all the files required by the NAO are copied, the *deploy* command calls the *updateSettings* command, which generates a new *settings.cfg* according to the configuration tracked by the `bush` for each of the selected robots. Of course the *updateSettings* command can also be called without the *deploy* in order to reconfigure several robots without the need of updating the binaries. After updating the file *settings.cfg*, the *bhuman* software has to be restarted for changes to take effect. This can easily be done with the command *restart*. If it is called without any parameter, it restarts only the *bhuman* software but it can also be used to restart NAOqi and *bhuman*, and the entire operating system of the robot if you call it with one of the parameters *naoqi*, *full*, or *robot*. To inspect the configuration files copied to the robots, you can use the command *show*, which knows most of the files located on the robots and can help you finding the desired files with tab completion.

Command	Parameter(s) Description
changeWireless	[<wifi config>] Changes the active wireless configuration.
deploy	[<config>] Deploys code and all settings to the robot(s) using *copyfiles*.
downloadLogs	 Downloads all logs from the robot(s) and stores them at Config/Logs. Afterwards the logs are deleted from the robot(s).
dr	<debug response> Sends a debug response to the robot(s).
ping	 Pings the robot(s)
restart	[bhuman | naoqi | full | robot] Restarts bhumand, naoqid, both or the robot. If no parameter is given bhuman will be restarted.
scp	@<path on NAO > <local path> | <local path> <path on NAO > Copies a file to or from the robot(s). The first argument is the source and the second the destination path.
show	<config file> Prints the config file stored on the robot(s).
shutdown	 Executes a shutdown on the robot(s).
ssh	<command> Executes the command via ssh on the robot(s).
updateSettings	 Updates the file "settings.cfg" on the robot(s).
updateWireless	 Updates the wireless configurations on the robot(s)

Table 8.3: Bush commands that need at least one selected robot.

8.2.4 Managing Multiple Wireless Configurations

Since the robot soccer competition generally takes place on more than just a single field and normally each field has its own WLAN access point, the robots have to deal with multiple configuration files for their wireless interfaces. The `bush` helps to manage these various files with the commands *updateWireless* and *changeWireless*. The first command can be used to copy all new configuration files to the NAO robot, while the latter activates a specific one on the robot. Which configuration is used, can be specified in the first argument of *changeWireless*. If the argument is omitted, the wireless configuration of the selected team is used.

8.2.5 Substituting Damaged Robots

The robots known to `bush` are arranged in two rows. The entries in the upper row represent the playing robots and the entries in the lower row the robots which stand by as substitutes. To select which robots are playing and which are not, you can move them by drag&drop to the appropriate position. Since this view only supports ten robots at a time, there is another view

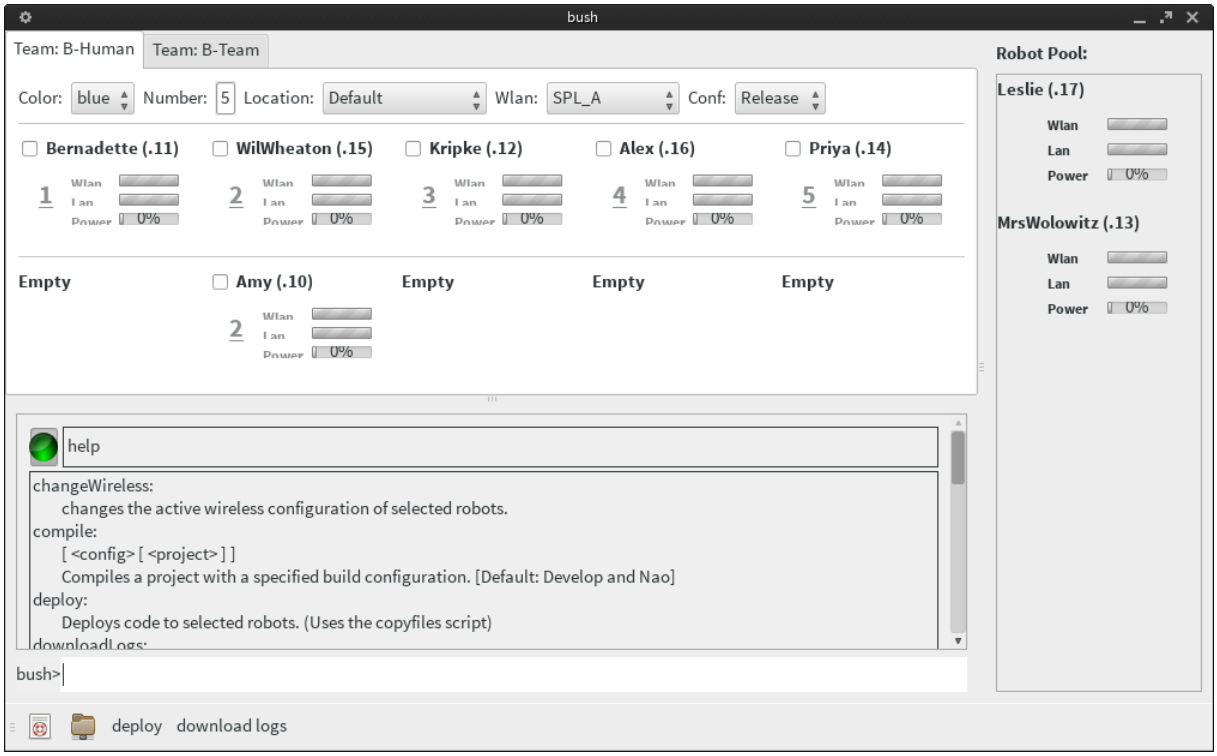


Figure 8.16: An example figure of the *bush*

called *RobotPool*, which contains all other robots. It can be pulled out at the right side of the *bush* window. The robots displayed there can be exchanged with robots from the main view.

8.2.6 Monitoring Robots

The *bush* displays some information about the robots' states as you can see in Figure 8.16: wireless connection pings, wired connection pings, and remaining battery load. But you cannot always rely on this information, because it is only collected properly if the robots are reachable and the *bhuman* software is running on the robot.

The power bar shows the remaining battery load. The *bush* reads this information from the team communication (i. e. representation *RobotHealth*). The robot has to be at least in the state *initial* in order to send team communication messages, i. e. it must stand. The power bar will freeze until the next message is received.

8.3 GameController

A RoboCup game has a human referee. Unfortunately the robots cannot understand him or her directly. Instead the referee's assistant relays the decisions to the robots using a software called *GameController*. From this year on the official *GameController* is the one we created. It is written in Java 1.6.

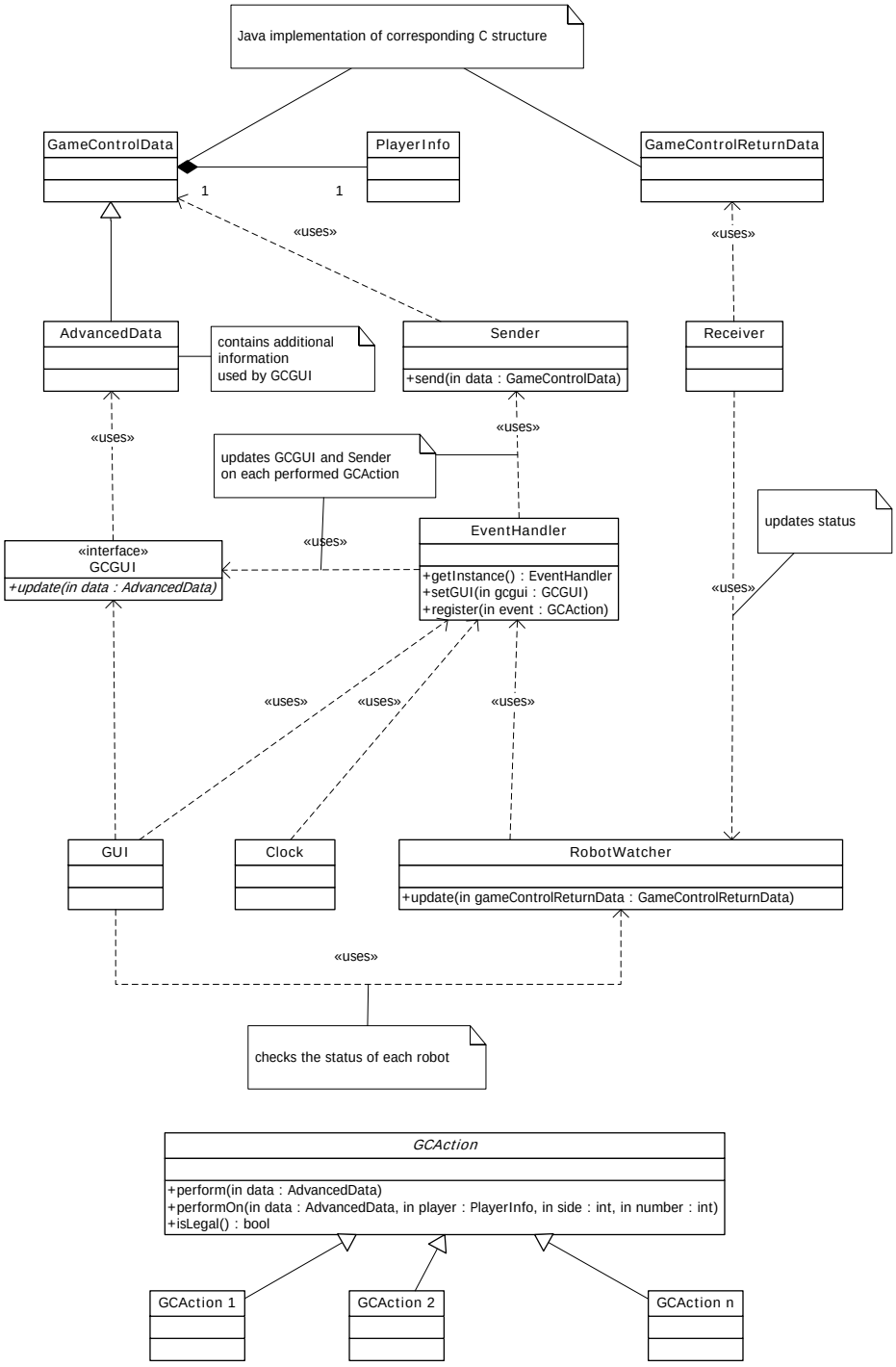


Figure 8.17: The architecture of the *GameController*

8.3.1 Architecture

The architecture (cf. Fig. 8.17) is based on a combination of the model-view-controller (MVC) and the command pattern.

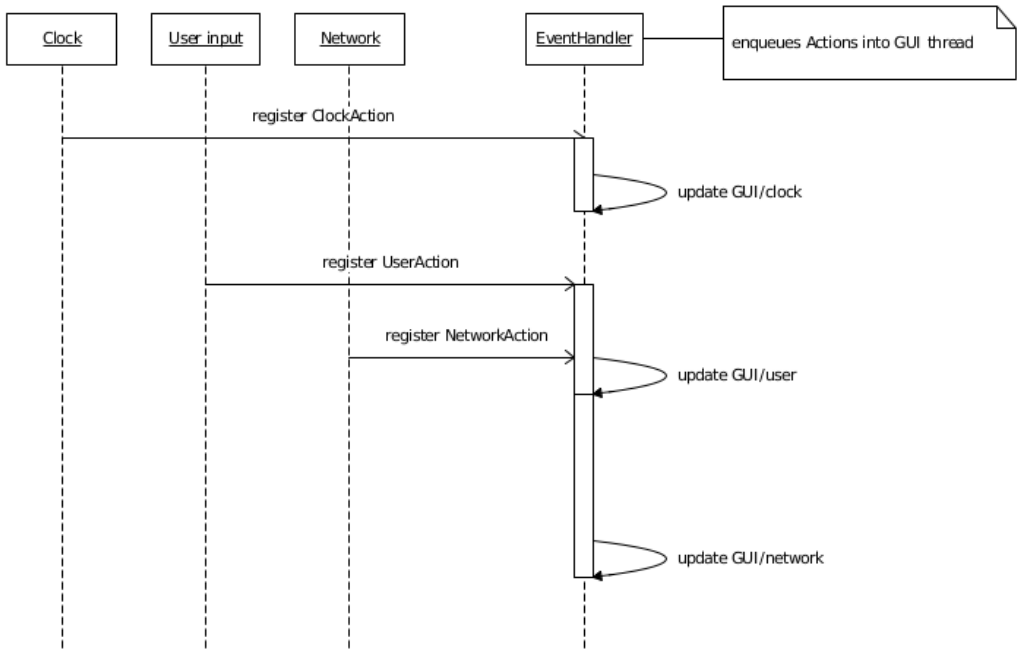


Figure 8.18: The sequences between some threads

The GameController communicates with the robots using a C data structure called `RoboCupGameControlData` as mentioned in the RoboCup SPL rules. It contains information about the current game and penalty state of each robot. It is broadcasted via UDP several times per seconds. Robots may answer using the `RoboCupGameControlReturnData` C data structure. However, this feature is optional and most teams do not implement it.

Both C data structures were translated to Java for the GameController.[3] They only hold the information and provide conversion methods from and to a byte stream. Unfortunately, the `GameControlData` does not contain all the information needed to fully describe the current state of the game. For example, it lacks information about the number of timeouts or penalties that a team has taken and the game time is only precise up to one second. Therefore, the class `GameControlData` is extended by a class called `AdvancedData`. This class holds the complete game state. From the classical MVC point of view, the `AdvancedData` is the model.

The view component is represented by the package `GUI`. All GUI functionality is controlled via the interface `GCGUI`. `GCGUI` only provides an update method with an instance of the class `AdvancedData` as a parameter. This update method is called frequently. Therefore, the GUI can only display the same game state as the one that is transmitted.

The controller part of the model-view-controller architecture is kind of tricky, because it has to deal with parallel inputs from the user, a ticking clock, and robots via the network. To simplify the access to the model, we only allow access to it from a single thread. Everything that modifies the model is encapsulated in action classes, which are defined by extending the abstract class `GCAction`. All threads (GUI, timing, network) register actions at the `EventHandler`. The `EventHandler` executes the actions on the GUI thread (cf. Fig. 8.18).

Each action knows, based on the current game state, whether it can be legally executed according to the rules. For example, switching directly from the *initial* to the *playing* state, penalizing a robot for holding the ball in the *ready* state or decreasing the goal count is illegal.

[3]Their names leave out the prefix “RoboCup”

Figure 8.19: Start screen of the *GameController*

8.3.2 UI Design

After launching the GameController, a small window will appear to select the basic settings of the game (cf. Fig. 8.19). The most basic decision is the league. You can choose between SPL and three Humanoid leagues: Kid-, Teen- and Adult-Size. You can select, which teams will play and whether it is a play-off game, as well as you can choose between a fullscreen and a fully scalable windowed mode. You can also decide, whether the teams will change colors during halftime. After pressing the start button, the main GUI will appear.

The look of the GUI (cf. Fig. 8.20) is completely symmetric and all buttons are as big as possible. In addition, keyboard shortcuts are provided for most buttons. Thus making it possible to operate the GameController in a more efficient way. Buttons are only enabled if the corresponding actions are legal. This should decrease the chance that the operator of the *GameController* presses a wrong button. Since mistakes such as penalizing the wrong robot can still occur, the GUI provides an undo functionality that allows to revert actions. This clearly distinguishes normal actions that must follow the rules from corrective actions that are only legal because they heal a mistake that was made before.

All undoable actions are displayed in a timeline at the bottom of the *GameController*. By double clicking on one of the actions in the timeline, the state will be reverted to the state right before that action has been executed. However the game time will only be reverted if a transition between different game states is reverted as well.

When testing their robots, most teams want to be able to do arbitrary state transitions with the *GameController*. Therefore, it has a functionality to switch in and out of a test mode. While being in test mode, all actions are allowed at any time.

8.3.3 Game State Visualizer

The *GameController* comes with two additional tools within the same Java project. One of them is the *GameStateVisualizer* (cf. Fig. 8.21). Its purpose is to show the state of the game to the audience. This includes the teams playing, the current score, the time remaining, and some

Figure 8.20: The main screen of the *GameController*

other information. The *GameStateVisualizer* listens through the network to the *GameController* and displays everything of interest.

For the purpose of testing the network or packages the robots should receive, there is a test mode which can be switched on or off by pressing F11. In the test mode, the *GameStateVisualizer* shows everything that is contained in the packages and how it should be interpreted.

8.3.4 Log Analyzer

The *GameController* writes some status information and all decisions entered to a log file with a unique name. So after a competition, hundreds of log files exist, not only the ones of official games, but also of practice matches, test kick-offs, etc. The `LogAnalyzer` allows to interactively select the log files that resulted from official games and then convert from log files to files that can be imported by a spreadsheet application. The `LogAnalyzer` also cleans the data, e. g. by removing decisions that were reverted later, so they do not impede statistics made later.

Right after you launch the LogAnalyzer, it quickly parses each log file and then analyzes some meta information to make a guess, whether and why a log file may not belong to a real game. It then lists all logs in a GUI with that guess (cf. Fig. 8.22). While dealing with hundreds of log-files, you can select the right ones within a few minutes by comparing them to the timetable of the event. Afterwards a file containing the consolidated information of all the selected logs in the form of comma separated values can be created.

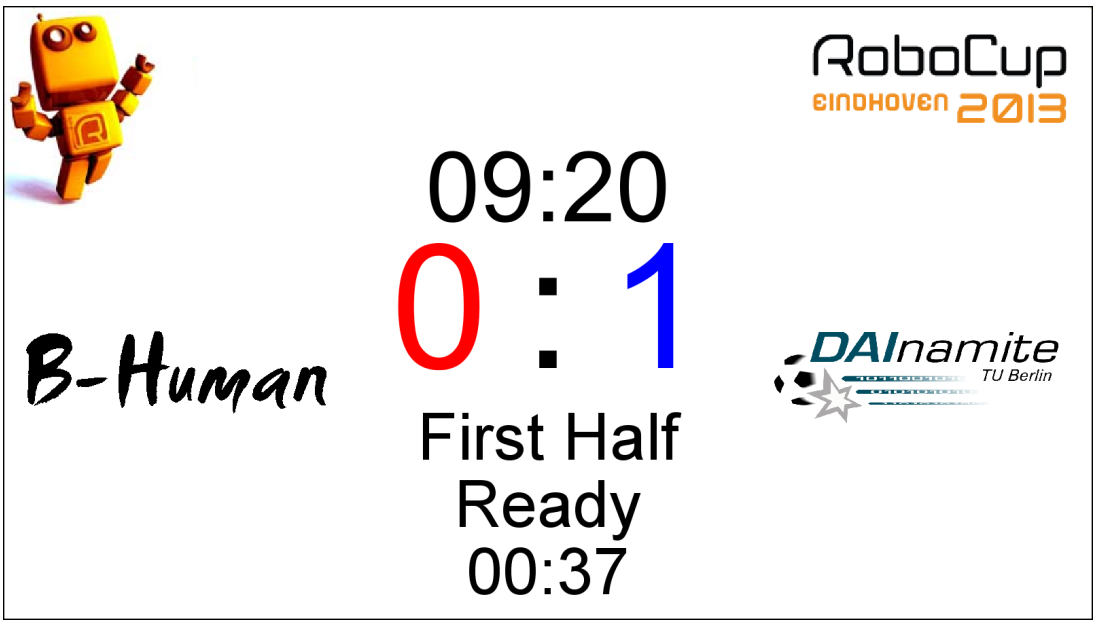


Figure 8.21: The *GameStateVisualizer*

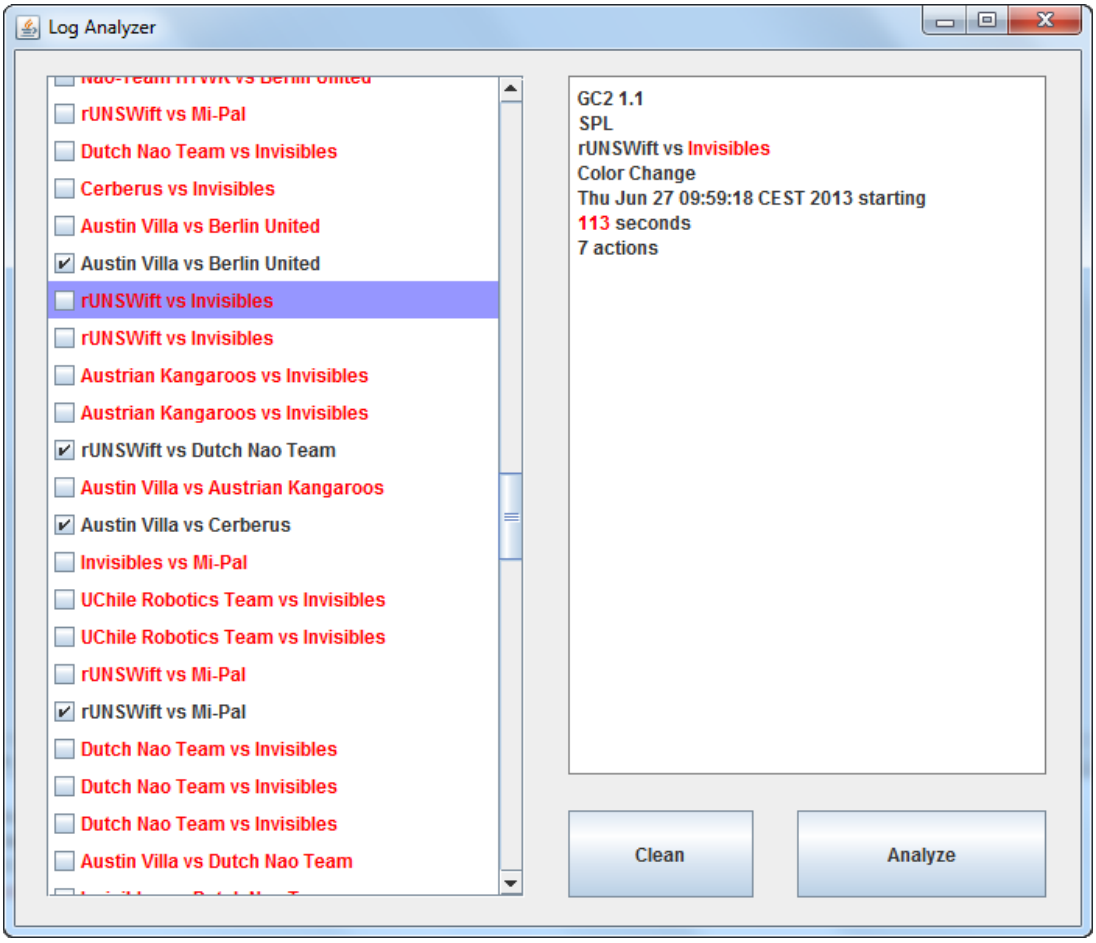


Figure 8.22: The *LogAnalyzer*

Chapter 9

Acknowledgements

We gratefully acknowledge the support given by Aldebaran Robotics. We thank our current sponsors IGUS and TME as well as our previous sponsors Wirtschaftsförderung Bremen (WFB), Sparkasse Bremen, Deutsche Forschungsgemeinschaft (DFG), and German Academic Exchange Service (DAAD) for funding parts of our project. Since B-Human 2013 did not start its software from scratch, we also want to thank the previous team members as well as 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 generating the graphs shown in the options view and the module view of the simulator.
(`http://www.graphviz.org`)

ccache: A fast C/C++ compiler cache.
(`http://ccache.samba.org`)

clang: A compiler front end for the C, C++, Objective-C, and Objective-C++ programming languages.
(`http://clang.llvm.org`)

Eigen: A C++ template library for linear algebra: matrices, vectors, numerical solvers, and related algorithms.
(`http://eigen.tuxfamily.org`)

getModKey: For checking whether the shift key is pressed in the Deploy target on OS X.
(`http://allancraig.net/index.php?option=com_docman&Itemid=100`, not available anymore)

ld: The GNU linker is used for cross linking on Windows and OS X.
(`http://sourceware.org/binutils/docs-2.21/ld`)

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

libjpeg-turbo: For the NAO we use an optimized version of the libjpeg library.
(`http://libjpeg-turbo.virtualgl.org`)

libqxt: For showing the sliders in the camera calibration view of the simulator.
(`http://dev.libqxt.org/libqxt/wiki/Home`)

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

mare: Build automation tool and project file generator.
(`http://github.com/craflin/mare`)

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

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

Qt: The GUI framework of the simulator.
(`http://qt-project.org`)

qtpropertybrowser: Extends the Qt framework with a property browser.
(`https://qt.gitorious.org/qt-solutions/qt-solutions/source/80592b0e7145fb876ea0e84a6e3dadfd5f7481b6:qtpropertybrowser`)

sshpass: Non-interactive ssh password provider used in the *installRobot* scripts.
(`http://sourceforge.net/projects/sshpass`)

snappy: Used for the compression of log files.
(`http://code.google.com/p/snappy`)

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] Google. Snappy – a fast compressor/decompressor. Online: `http://code.google.com/p/snappy`, September 2013.

[5] Colin Graf. Entwicklung eines geregelten Laufs für den NAO. Diploma thesis, University of Bremen, 2013.

[6] Colin Graf and Thomas Röfer. A center of mass observing 3D-LIPM gait for the RoboCup Standard Platform League humanoid. In Thomas Röfer, Norbert Michael Mayer, Jesus Savage, and Uluc Saranli, editors, *RoboCup 2011: Robot Soccer World Cup XV*, Lecture Notes in Artificial Intelligence. Springer, 2011.

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

[8] Jan Hoffmann, Matthias Jüngel, and Martin Lötzsch. A vision based system for goal-directed obstacle avoidance used in the RC'03 obstacle avoidance challenge. In *In 8th International Workshop on RoboCup 2004 (Robot World Cup Soccer Games and Conferences), Lecture Notes in Artificial Intelligence*, pages 418–425. Springer, 2004.

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

[10] 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.

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

[12] James J. Kuffner and Steven M. LaValle. RRT-connect: An efficient approach to single-query path planning. In *Proceedings of the 2000 IEEE International Conference on Robotics and Automation (ICRA 2000)*, volume 2, pages 995–1001, San Francisco, CA, USA, 2000.

[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, 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.

[16] Steven M. LaValle. Rapidly-exploring random trees: A new tool for path planning. Technical Report TR 98-11, Computer Science Dept., Iowa State University, 1998.

[17] Steven M. LaValle and James J. Kuffner. Randomized kinodynamic planning. In *Proceedings of the 1999 IEEE International Conference on Robotics and Automation (ICRA 1999)*, volume 1, pages 473–479, Detroit, MI, USA, 1999.

[18] Judith Müller, Tim Laue, and Thomas Röfer. Kicking a Ball – Modeling Complex Dynamic Motions for Humanoid Robots. In Javier Ruiz del Solar, Eric Chown, and Paul G. Ploeger, editors, *RoboCup 2010: Robot Soccer World Cup XIV*, volume 6556 of *Lecture Notes in Artificial Intelligence*, pages 109–120. Springer, 2011.

[19] Nobuyuki Otsu. A threshold selection method from grey level histograms. *IEEE Transactions on Systems, Man, and Cybernetics*, 9(1):62–66, January 1979.

[20] Thomas Reinhardt. Kalibrierungsfreie Bildverarbeitungsalgorithmen zur echtzeitfähigen Objekterkennung im Roboterfußball. Master's thesis, HTWK Leipzig, 2011.

[21] Thomas Röfer, Tim Laue, Judith Müller, Michel Bartsch, Arne Böckmann, Florian Maaß, Thomas Münder, Marcel Steinbeck, Simon Taddiken, Alexis Tsogias, and Felix Wenk. B-Human team description for RoboCup 2013. In *RoboCup 2013: Robot Soccer World Cup XVII Preproceedings*, RoboCup Federation, 2013.

[22] 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.

[23] 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.

[24] Thomas Röfer and Tim Laue. On B-Human's code releases in the standard platform league – software architecture and impact. In *RoboCup 2013: Robot Soccer World Cup XVII*, Lecture Notes in Artificial Intelligence. Springer, 2013. to appear.

[25] Thomas Röfer, Tim Laue, Arne Böckmann, Judith Müller, and Alexis Tsogias. B-Human 2013: Ensuring stable game performance. In *RoboCup 2013: Robot Soccer World Cup XVII*, Lecture Notes in Artificial Intelligence. Springer, 2013. to appear.

[26] Thomas Röfer, Tim Laue, Judith Müller, Michel Bartsch, Malte Jonas Batram, Arne Böckmann, Nico Lehmann, Florian Maaß, Thomas Münder, Marcel Steinbeck, Andreas Stolpmann, Simon Taddiken, Robin Wieschendorf, and Danny Zitzmann. B-Human team report and code release 2012, 2012. Only available online: `http://www.b-human.de/downloads/coderelease2012.pdf`.

[27] Thomas Röfer, Tim Laue, Judith Müller, Colin Graf, Arne Böckmann, and Thomas Münder. B-Human Team Description for RoboCup 2012. In Xiaoping Chen, Peter Stone, Luis Enrique Sucar, and Tijn Van der Zant, editors, *RoboCup 2012: Robot Soccer World Cup XV Preproceedings*. RoboCup Federation, 2012.

[28] 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`.

[29] Sebastian Thrun, Wolfram Burgard, and Dieter Fox. *Probabilistic Robotics*. MIT Press, Cambridge, 2005.

[30] Max Trocha. Werkzeug zur taktischen Auswertung von Spielsituationen. Bachelor's thesis, University of Bremen, 2010.

[31] Felix Wenk and Thomas Röfer. Online Generated Kick Motions for the NAO Balanced Using Inverse Dynamics. In *RoboCup 2013: Robot Soccer World Cup XVII*, Lecture Notes in Artificial Intelligence. Springer, 2013.

Appendix A

The Scene Description Language

A.1 EBNF

In the next section the structure of a scene description file is explained by means of an EBNF representation of the language. In the following, you can find an explanation of the symbols used.

Symbols surrounded by ?(...)?
: Parentheses with question marks mean, that the order of all elements between them is irrelevant. That means every permutation of elements within those brackets is allowed. For example: *something* =?(*firstEle secondEle thirdEle*)?; can also be written as *something* =?(*secondEle firstEle thirdEle*)? or as *something* = ?(*thirdEle firstEle sencondEle*)? and so on.

Symbols surrounded by !
: $!x, y, \ldots!$ means, that each rule is required. In fact the exclamation marks should only underline that all elements between them are absolutely required. In normal EBNF-Notation a rule like $Hinge = \ldots !bodyClass,\ axisClass!\ldots$ can be written as $Hinge = \ldots bodyClass\ axisClass\ldots$.

+[...]+
: $+[x, y]+$ means, that x and y are optional. You could also write $somewhat = +[x, y, z]+$ as $somewhat = [x]\ [y]\ [z]$.

{...}
: Elements within curly braces are repeatable optional elements. These brackets have the normal EBNF meaning.

"..."
: Terminal symbols are marked with quotation marks.

A.2 Grammar

```
appearanceClass      = Appearance | BoxAppearance | SphereAppearance
                     | CylinderAppearance | CapsuleAppearance | ComplexAppearance;
axisClass            = Axis;
bodyClass            = Body;
compoundClass        = Compound;
```

```
deflectionClass      = Deflection;
extSensorClass       = Camera | DepthImageSensor | SingleDistanceSensor
                     | ApproxDistanceSensor;
frictionClass        = Friction | RollingFriction;
geometryClass        = Geometry | BoxGeometry | CylinderGeometry
                     | CapsuleGeometry | SphereGeometry;
infrastructureClass  = Simulation | Include;
intSensorClass       = Accelerometer | Gyroscope | CollisionSensor;
jointClass           = Hinge | Slider;
lightClass           = Light;
massClass            = Mass | BoxMass | InertiaMatrixMass | SphereMass;
materialClass        = Material;
motorClass           = ServoMotor | VelocityMotor;
primitiveGroupClass  = Quads | Triangles;
rotationClass        = Rotation;
sceneClass           = Scene;
setClass             = Set;
solverClass          = Quicksolver;
texCoordsClass       = TexCoords;
translationClass     = Translation;
verticesClass        = Vertices;


Accelerometer        = "<Accelerometer></Accelerometer>" | "<Accelerometer/>";
Gyroscope            = "<Gyroscope></Gyroscope>" | "<Gyroscope/>";
CollisionSensor      = "<CollisionSensor>"  ?( +[translationClass, rotationClass]+
                       {geometryClass} )? "</CollisionSensor>";

Appearance           = "<Appearance>" ?( +[translationClass, rotationClass]+
                       {setClass | appearanceClass} )? "</Appearance>";
BoxAppearance        = "<BoxAppearance>" ?( !surfaceClass! +[translationClass,
                       rotationClass]+ {setClass | appearanceClass} )?
                       "</BoxAppearance>";
ComplexAppearance    = "<ComplexAppearance>" ?( !surfaceClass, verticesClass,
                       primitiveGroupClass! +[translationClass, rotationClass
                       | texCoordsClass]+ {setClass | appearanceClass
                       | primitiveGroupClass} )? "</ComplexAppearance>";
CapsuleAppearance    = "<CapsuleAppearance>" ?( !surfaceClass!
                       +[translationClass, rotationClass]+
                       {setClass | appearanceClass} )? "</CapsuleAppearance>";
CylinderAppearance   = "<CylinderAppearance>" ?( !surfaceClass!
                       +[translationClass, rotationClass]+
                       {setClass | appearanceClass} )? "</CylinderAppearance>";
SphereAppearance     = "<SphereAppearance>" ?( !surfaceClass!
                       +[translationClass, rotationClass]+
                       {setClass | appearanceClass} )? "</SphereAppearance>";

ApproxDistanceSensor = "<ApproxDistanceSensor>" ?( +[translationClass,
                        rotationClass]+ )? "</ApproxDistanceSensor>";
Camera               = "<Camera>" ?( +[translationClass, rotationClass]+ )?
                       "</Camera>";
DepthImageSensor     = "<DepthImageSensor>" ?( +[translationClass,
                       rotationClass]+ )? "</DepthImageSensor>";
SingleDistanceSensor = "<SingleDistanceSensor>" ?( +[translationClass,
                       rotationClass]+ )? "</SingleDistanceSensor>";
```

```
Mass                = "<Mass>" ?( +[translationClass, rotationClass]+
                      {setClass | massClass} )? "</Mass>";
BoxMass             = "<BoxMass>" ?( +[translationClass | rotationClass]+
                      {setClass | massClass} )? "</BoxMass>";
InertiaMatrixMass   = "<InertiaMatrixMass>"
                      ?( +[translationClass, rotationClass]+
                      {setClass | massClass} )? "</InertiaMatrixMass>";
SphereMass          = "<SphereMass>" ?( +[translationClass | rotationClass]+
                      {setClass | massClass} )? "</SphereMass>"

Geometry            = "<Geometry>" ?( +[translationClass, rotationClass,
                      materialClass]+ {setClass | geometryClass} )?
                      "</Geometry>";
BoxGeometry         = "<BoxGeometry>" ?( +[translationClass, rotationClass,
                      materialClass]+ {setClass | geometryClass} )?
                      "</BoxGeometry>";
CylinderGeometry    = "<CylinderGeometry>"
                      ?( +[translationClass, rotationClass, materialClass]+
                      {setClass | geometryClass} )? "</CylinderGeometry>";
CapsuleGeometry     = "<CapsuleGeometry>"
                      ?( +[translationClass, rotationClass, materialClass]+
                      {setClass | geometryClass} )? "</CapsuleGeometry>";
SphereGeometry      = "<SphereGeometry>"
                      ?( +[translationClass, rotationClass, materialClass]+
                      {setClass | geometryClass} )? "</SphereGeometry>";

Axis                = "<Axis>" ?( +[motorClass, deflectionClass]+
                      {setClass} )? "</Axis>";
Hinge               = "<Hinge>" ?( !bodyClass, axisClass! +[translationClass,
                      rotationClass]+ {setClass} )? "</Hinge>";
Slider              = "<Slider>" ?( !bodyClass, axisClass! +[translationClass,
                      rotationClass]+ {setClass} )? "</Slider>";

Body                = "<Body>" ?( !massClass! +[translationClass,
                      rotationClass]+ {setClass | jointClass | appearanceClass
                      | geometryClass | massClass | intSensorClass |
                      extSensorClass} )? "</Body>";

Material            = "<Material>" ?( {setClass | frictionClass} )?
                      "</Material>";
Friction            = "<Friction></Friction>" | "<Friction/>";
RollingFriction     = "<RollingFriction></RollingFriction>"
                    | "<RollingFriction/>";

ServoMotor          = "<ServoMotor></ServoMotor>" | "</ServoMotor>";
VelocityMotor       = "<VelocityMotor></VelocityMotor>" | "</VelocityMotor>";

Simulation          = "<Simulation>" !sceneClass! "</Simulation>";
Scene               = "<Scene>" ?( +[solverClass]+ {setClass | bodyClass
                      | compoundClass | lightClass} )? "</Scene>";

Compound            = "<Compound>" ?( +[translationClass, rotationClass]+
                      {setClass | compoundClass | bodyClass | appearanceClass
                      | geometryClass | extSensorClass} )? "</Compound>";
Deflection          = "<Deflection></Deflection>" | "<Deflection/>";
Include             = "<Include></Include>" | "<Include/>";
```

```
Light                = "<Light></Light>" | "<Light/>";
Set                  = "<Set></Set>" | "<Set/>";

Rotation             = "<Rotation></Rotation>" | "<Rotation>";
Translation          = "<Translation></Translation>" | "<Translation/>";

Quads                = "<Quads>" Quads Definition "</Quads>";
TexCoords            = "<TexCoords>" TexCoords Definition "</TexCoords>";
Triangles            = "<Triangles>" Triangles Definition "</Triangles>";
Vertices             = "<Vertices> Vertices Definition "</Vertices>";
```

A.3 Structure of a Scene Description File

A.3.1 The Beginning of a Scene File

Every scene file has to start with a *<Simulation>* tag. Within a *Simulation* block a *Scene* element is required, but there is one exception: files included via *<Include href=...>* must start with *<Simulation>*, but there is no *Scene* element required. A *Scene* element specifies which controller is loaded for this scene via the *controller* attribute (in our case all scenes set the *controller* attribute to *SimulatedNao*, so that the libSimulatedNao is loaded by SimRobot). It is recommended to include other specifications per *Include* before the scene description starts (compare with BH2013.ros2), but it is not necessary.

A.3.2 The ref Attribute

An element with a name attribute can be referenced by the *ref*-attribute using its name, i. e. elements that are needed repeatedly in a scene need to be defined only once. For example there is only one description of the structure of a goal in the field description file (Field2013SPL.rsi2), but both a yellow and a blue goal are needed on a field. So there are two references to the goal definition. The positioning of the two goals is done by *Translation* and *Rotation* elements. The color is set by a *Set* element, which is described below.

```
⋮
<Compound ref="fieldGoal" name="yellowGoal">
  <Translation x="-3m"/>
  <Set name="color" value="fieldYellow"/>
</Compound>
<Compound ref="fieldGoal" name="blueGoal">
  <Translation x="3m"/>
  <Rotation z="180degree"/>
  <Set name="color" value="fieldBlue"/>
</Compound>
⋮
<Compound name="fieldGoal">
  <CylinderGeometry height="800mm" radius="50mm" name="post1">
    <Translation y="-700mm" z="400mm"/>
  </CylinderGeometry>
⋮
```

A.3.3 Placeholders and Set Element

A placeholder has to start with a $ followed by a arbitrary string. A placeholder is replaced by the definition specified within the corresponding *Set* element. The attribute *name* of a *Set* elements specifies the placeholder, which is replaced by the value specified by the attribute *value* of the *Set* element.

In the following code example, the color of a post of a goal is set by a *Set* element. Within the definition of the compound fieldGoal named yellowGoal, the *Set* element sets the placeholder color to the value fieldYellow. The placeholder named color of post1, which is defined in the general definition of a field goal, is replaced by fieldYellow. So the *Surface* elements reference a *Surface* named fieldYellow.

```
⋮
<CylinderAppearance height="800mm" radius="50mm" name="post1">
  <Translation y="-700mm" z="400mm"/>
  <Surface ref="$color"/>
</CylinderAppearance>
⋮
<Compound ref="fieldGoal" name="yellowGoal">
  <Translation x="-3m"/>
  <Set name="color" value="fieldYellow"/>
</Compound>
⋮
<Surface name="fieldYellow" diffuseColor="rgb(60%, 60%, 0%)" ambientColor="rgb
    (45%, 45%, 0%)" specularColor="rgb(40%, 40%, 40%)" shininess="30"/>
⋮
```

A.4 Attributes

A.4.1 infrastructureClass

- **Include** This tag includes a file specified by href. The included file has to start with $< Simulation >$.
  - href
- **Simulation**
  This element does not have any attributes.

A.4.2 setClass

- **Set** This element sets a placeholder referenced by the attribute *name* to the value specified by the attribute *value*
  - name The name of a placeholder.
    - **Use:** required
    - **Range:** String
  - value The value the placeholder is set.
    - **Use:** required
    - **Range:** String

A.4.3 sceneClass

- **Scene** Describes a scene and specifies the controller of the simulation.
  - name The identifier of the scene object (must always be *RoboCup*).
    - **Use:** optional
    - **Range:** String
  - controller The name of the controller library (without prefix *lib*; in our case it is *SimulatedNao*).
    - **Use:** optional
    - **Range:** String
  - color The background color of the scene, see A.4.22.
  - stepLength
    - **Units:** s
    - **Default:** 0.01s
    - **Use:** optional
    - **Range:** $(0, MAXFLOAT]$
  - gravity Sets the gravity in this scene.
    - **Units:** $\frac{mm}{s^2}$, $\frac{m}{s^2}$
    - **Default:** $-9.80665\frac{m}{s^2}$
    - **Use:** optional
  - CFM Sets ODE cfm (constraint force mixing) value.
    - **Default:** -1
    - **Use:** optional
    - **Range:** $[0, 1]$
  - ERP Set ODE erp (error reducing parameter) value.
    - **Default:** -1
    - **Use:** optional
    - **Range:** $[0, 1]$
  - contactSoftERP Sets another erp value for colliding surfaces.
    - **Default:** -1
    - **Use:** optional
    - **Range:** $[0, 1]$
  - contactSoftCFM Sets another cfm value for colliding surfaces.
    - **Default:** -1
    - **Use:** optional
    - **Range:** $[0, 1]$

A.4.4 solverClass

- **Quicksolver**
  - iterations
    - **Default:** -1

    - ∗ **Use:** optional
    - ∗ **Range:** $(0, MAXINTEGER]$
  - skip
    - ∗ **Default:** 1
    - ∗ **Use:** optional
    - ∗ **Range:** $(0, MAXINTEGER]$

A.4.5 bodyClass

- **Body** Specifies an object that has a mass and can move.
  - name The name of the body.
    - ∗ **Use:** optional
    - ∗ **Range:** String

A.4.6 compoundClass

- **Compound** A Compound is a non-moving object. In contrast to the *Body* element a compound does not require a *Mass* element as child.
  - name The name of the compound.
    - ∗ **Use:** optional
    - ∗ **Range:** String

A.4.7 jointClass

- **Hinge** Defines a hinge. To define the axis of the hinge, this element requires an axis element as child element. Furthermore, a body element is required to which the hinge is connected.
  - name The name of the hinge.
    - ∗ **Use:** optional
    - ∗ **Range:** String
- **Slider** Defines a slider. Requires an axis element to specify the axis and a body element, which defines the body this slider is connected to.
  - name The name of the slider.
    - ∗ **Use:** optional
    - ∗ **Range:** String

A.4.8 massClass

- **Mass** All this mass classes define the mass of an object.
  - name The name of the mass declaration.
    - ∗ **Use:** optional
    - ∗ **Range:** String

- **BoxMass**
  - name The name of the boxMass declaration.
    - **Use:** optional
    - **Range:** String
  - value The mass of the box.
    - **Units:** g, kg
    - **Default:** 0
    - **Use:** required
    - **Range:** $[0, MAXFLOAT]$
  - width The width of the box.
    - **Units:** mm, cm, dm, m, km
    - **Default:** 0
    - **Use:** required
    - **Range:** $[-MAXFLOAT, MAXFLOAT]$
  - height The height of the box.
    - **Units:** mm, cm, dm, m, km
    - **Default:** 0
    - **Use:** required
    - **Range:** $[-MAXFLOAT, MAXFLOAT]$
  - depth The depth of the box.
    - **Units:** mm, cm, dm, m, km
    - **Default:** 0
    - **Use:** required
    - **Range:** $[-MAXFLOAT, MAXFLOAT]$
- **SphereMass**
  - name The name of the sphereMass declaration.
    - **Use:** optional
    - **Range:** String
  - value The mass of the sphere.
    - **Units:** g, kg
    - **Default:** 0
    - **Use:** required
    - **Range:** $[0, MAXFLOAT]$
  - radius The radius of the sphere.
    - **Units:** mm, cm, dm, m, km
    - **Default:** 0
    - **Use:** required
    - **Range:** $[0, MAXFLOAT]$
- **InertiaMatrixMass** The matrix of the mass moment of inertia. Note that this matrix is a symmetric matrix.

- name The name of the InertiaMatrixMass declaration.
  - **Use:** optional
  - **Range:** String
- value The total mass.
  - **Units:** g, kg
  - **Default:** 0
  - **Use:** required
  - **Range:** $[0, MAXFLOAT]$
- x The center of mass in x direction.
  - **Units:** mm,cm,dm,m,km
  - **Default:** 0
  - **Use:** optional
  - **Range:** $[-MAXFLOAT, MAXFLOAT]$
- y The center of mass in y direction.
  - **Units:** mm,cm,dm,m,km
  - **Default:** 0
  - **Use:** optional
  - **Range:** $[-MAXFLOAT, MAXFLOAT]$
- z The center of mass in z direction.
  - **Units:** mm,cm,dm,m,km
  - **Default:** 0
  - **Use:** optional
  - **Range:** $[-MAXFLOAT, MAXFLOAT]$
- ixx Moment of inertia around the x-axis when the object is rotated around the x-axis.
  - **Units:** $g * mm^2$, $kg * m^2$
  - **Default:** 0
  - **Use:** required
  - **Range:** $[-MAXFLOAT, MAXFLOAT]$
- ixy Moment of inertia around the y-axis when the object is rotated around the x-axis.
  - **Units:** $g * mm^2$, $kg * m^2$
  - **Default:** 0
  - **Use:** optional
  - **Range:** $[-MAXFLOAT, MAXFLOAT]$
- ixz Moment of inertia around the z-axis when the object is rotated around the x-axis.
  - **Units:** $g * mm^2$, $kg * m^2$
  - **Default:** 0
  - **Use:** optional
  - **Range:** $[-MAXFLOAT, MAXFLOAT]$
- iyy Moment of inertia around the y-axis when the object is rotated around the y-axis.
  - **Units:** $g * mm^2$, $kg * m^2$
  - **Default:** 0
  - **Use:** required

    - **Range:** $[-MAXFLOAT, MAXFLOAT]$
  - iyz Moment of inertia around the z-axis when the object is rotated around the y-axis
    - **Units:** $g * mm^2$, $kg * m^2$
    - **Default:** 0
    - **Use:** optional
    - **Range:** $[-MAXFLOAT, MAXFLOAT]$
  - izz Moment of inertia around the z-axis when the object is rotated around the z-axis.
    - **Units:** $g * mm^2$, $kg * m^2$
    - **Default:** 0
    - **Use:** required
    - **Range:** $[-MAXFLOAT, MAXFLOAT]$

A.4.9 geometryClass

- **Geometry** Elements of geometryClass specify the physical structure of an object.
  - name
    - **Use:** optional
    - **Range:** String
- **BoxGeometry**
  - color A color definition, see A.4.22
  - name
    - **Use:** optional
    - **Range:** String
  - width The width of the box.
    - **Units:** mm, cm, dm, m, km
    - **Default:** 0
    - **Use:** required
    - **Range:** $[-MAXFLOAT, MAXFLOAT]$
  - height The height of the box.
    - **Units:** mm, cm, dm, m, km
    - **Default:** 0
    - **Use:** required
    - **Range:** $[-MAXFLOAT, MAXFLOAT]$
  - depth The depths of the box.
    - **Units:** mm, cm, dm, m, km
    - **Default:** 0
    - **Use:** required
    - **Range:** $[-MAXFLOAT, MAXFLOAT]$
- **SphereGeometry**
  - color A color definition, see A.4.22

  - name
    - **Use:** optional
    - **Range:** String
  - radius The radius of the sphere.
    - **Units:** mm, cm, dm, m, km
    - **Default:** 0
    - **Use:** required
    - **Range:** $[-MAXFLOAT, MAXFLOAT]$

- **CylinderGeometry**
  - color A color definition, see A.4.22
  - name
    - **Use:** optional
    - **Range:** String
  - radius The radius of the cylinder.
    - **Units:** mm, cm, dm, m, km
    - **Default:** 0
    - **Use:** required
    - **Range:** $[-MAXFLOAT, MAXFLOAT]$
  - height The height of the cylinder.
    - **Units:** mm, cm, dm, m, km
    - **Default:** 0
    - **Use:** required
    - **Range:** $[-MAXFLOAT, MAXFLOAT]$

- **CapsuleGeometry**
  - color A color definition, see A.4.22
  - name
    - **Use:** optional
    - **Range:** String
  - radius The radius of the capsule.
    - **Units:** mm, cm, dm, m, km
    - **Default:** 0
    - **Use:** required
    - **Range:** $[-MAXFLOAT, MAXFLOAT]$
  - height The height of the capsule.
    - **Units:** mm, cm, dm, m, km
    - **Default:** 0
    - **Use:** required
    - **Range:** $[-MAXFLOAT, MAXFLOAT]$

A.4.10 materialClass

- **Material** Specifies a material.
  - **Use:** required
  - **Range:** String
  - name The name of the material.
    - **Use:** optional
    - **Range:** String

A.4.11 frictionClass

- **Friction** Specifies the friction between this material and an other material.
  - material The other material the friction belongs to.
    - **Use:** required
    - **Range:** String
  - value The value of the friction.
    - **Default:** 1
    - **Use:** required
    - **Range:** $[0, MAXFLOAT]$
- **RollingFriction** Specifies the rolling friction of an material.
  - material The other material the rolling friction belongs to.
    - **Use:** required
    - **Range:** String
  - value The value of the rolling friction.
    - **Default:** 1
    - **Use:** required
    - **Range:** $[0, MAXFLOAT]$

A.4.12 appearanceClass

- **Appearance** The appearance elements specify only shapes for the surfaces, so all appearance elements require a *Surface* specification. Appearance elements do not have a physical structure. Therefore a geometry has to be defined.
  - name The name of this appearance.
    - **Use:** optional
    - **Range:** String
- **BoxAppearance**
  - name The name of this appearance. To specify how it should look like an element of the type surfaceClass is needed.
    - **Use:** optional

    - ∗ **Range:** String
  - width The width of the box.
    - ∗ **Units:** mm, cm, dm, m, km
    - ∗ **Default:** 0
    - ∗ **Use:** required
    - ∗ **Range:** $[-MAXFLOAT, MAXFLOAT]$
  - height The height of the box.
    - ∗ **Units:** mm, cm, dm, m, km
    - ∗ **Default:** 0
    - ∗ **Use:** required
    - ∗ **Range:** $[-MAXFLOAT, MAXFLOAT]$
  - depth The depth of the box.
    - ∗ **Units:** mm, cm, dm, m, km
    - ∗ **Default:** 0
    - ∗ **Use:** required
    - ∗ **Range:** $[-MAXFLOAT, MAXFLOAT]$

- **SphereAppearance**
  - name
    - ∗ **Use:** optional
    - ∗ **Range:** String
  - radius The radius of the sphere.
    - ∗ **Units:** mm, cm, dm, m, km
    - ∗ **Default:** 0
    - ∗ **Use:** required
    - ∗ **Range:** $[-MAXFLOAT, MAXFLOAT]$

- **CylinderAppearance**
  - name
    - ∗ **Use:** optional
    - ∗ **Range:** String
  - height The height of the cylinder.
    - ∗ **Units:** mm, cm, dm, m, km
    - ∗ **Default:** 0
    - ∗ **Use:** required
    - ∗ **Range:** $[-MAXFLOAT, MAXFLOAT]$
  - radius The radius of the cylinder.
    - ∗ **Units:** mm, cm, dm, m, km
    - ∗ **Default:** 0
    - ∗ **Use:** required
    - ∗ **Range:** $[-MAXFLOAT, MAXFLOAT]$

- **CapsuleAppearance**

  - name
    - **Use:** optional
    - **Range:** String
  - height The height of the capsule.
    - **Units:** mm, cm, dm, m, km
    - **Default:** 0
    - **Use:** required
    - **Range:** $[-MAXFLOAT, MAXFLOAT]$
  - radius The radius of the capsule.
    - **Units:** mm, cm, dm, m, km
    - **Default:** 0
    - **Use:** required
    - **Range:** $[-MAXFLOAT, MAXFLOAT]$

- **ComplexAppearance**
  - name
    - **Use:** optional
    - **Range:** String

A.4.13 translationClass

- **Translation** Specifies a translation of an object.
  - x Translation in x direction.
    - **Units:** mm, cm, dm, m, km
    - **Default:** 0m
    - **Use:** optional
    - **Range:** $[-MAXFLOAT, MAXFLOAT]$
  - y Translation in y direction.
    - **Units:** mm, cm, dm, m, km
    - **Default:** 0m
    - **Use:** optional
    - **Range:** $[-MAXFLOAT, MAXFLOAT]$
  - z Translation in z direction.
    - **Units:** mm, cm, dm, m, km
    - **Default:** 0m
    - **Use:** optional
    - **Range:** $[-MAXFLOAT, MAXFLOAT]$

A.4.14 rotationClass

- **Rotation** Specifies the rotation of an object.
  - x Rotation around the x-axis.

    * **Units:** radian, degree
    * **Default:** 0degree
    * **Use:** optional
  - y Rotation around the y-axis.
    * **Units:** radian, degree
    * **Default:** 0degree
    * **Use:** optional
  - z Rotation around the z-axis.
    * **Units:** radian, degree
    * **Default:** 0degree
    * **Use:** optional

A.4.15 axisClass

- **Axis** Specifies the axis of a joint.
  - x
    * **Default:** 0
    * **Use:** optional
    * **Range:** $[-MAXFLOAT, MAXFLOAT]$
  - y
    * **Default:** 0
    * **Use:** optional
    * **Range:** $[-MAXFLOAT, MAXFLOAT]$
  - z
    * **Default:** 0
    * **Use:** optional
    * **Range:** $[-MAXFLOAT, MAXFLOAT]$
  - cfm
    * **Default:** -1
    * **Use:** optional
    * **Range:** $[0, 1]$

A.4.16 deflectionClass

- **Deflection** Specifies the maximum and minimum deflection of a joint.
  - min The minimal deflection.
    * **Default:** 0
    * **Use:** required
    * **Range:** $[-MAXFLOAT, MAXFLOAT]$
  - max The maximal deflection..
    * **Default:** 0
    * **Use:** required

    - **Range:** $[-MAXFLOAT, MAXFLOAT]$
  - stopCFM
    - **Default:** -1
    - **Use:** optional
    - **Range:** $[0, 1]$
  - stopERP
    - **Default:** -1
    - **Use:** optional
    - **Range:** $[0, 1]$

A.4.17 motorClass

- **ServoMotor**
  - maxVelocity The maximum velocity of this motor.
    - **Units:** radian/s, degree/s
    - **Default:** 0
    - **Use:** required
    - **Range:** $[-MAXFLOAT, MAXFLOAT]$
  - maxForce The maximum force of this motor.
    - **Units:** N
    - **Default:** 0
    - **Use:** required
    - **Range:** $[-MAXFLOAT, MAXFLOAT]$
  - p The p value of the motor's pid interface.
    - **Default:** 0
    - **Use:** required
    - **Range:** $[-MAXFLOAT, MAXFLOAT]$
  - i The i value of the motor's pid interface.
    - **Default:** 0
    - **Use:** optional
    - **Range:** $[-MAXFLOAT, MAXFLOAT]$
  - d The d value of the motor's pid interface.
    - **Default:** 0
    - **Use:** optional
    - **Range:** $[-MAXFLOAT, MAXFLOAT]$
- **VelocityMotor**
  - maxVelocity The maximum velocity of this motor.
    - **Units:** radian/s, degree/s
    - **Default:** 0
    - **Use:** required
    - **Range:** $[-MAXFLOAT, MAXFLOAT]$

  - maxForce The maximum force of this motor.
    * **Units:** N
    * **Default:** 0
    * **Use:** required
    * **Range:** $[-MAXFLOAT, MAXFLOAT]$

A.4.18 surfaceClass

- **Surface** Defines the appearance of a surface.
  - diffuseColor The diffuse color, see A.4.22.
  - ambientColor The ambient color of the light, see A.4.22.
  - specularColor The specular color, see A.4.22.
  - emissionColor The emitted color of the light, see A.4.22.
  - shininess The shininess value.
    * **Use:** optional
    * **Range:** o
  - diffuseTexture A texture.
    * **Use:** optional
    * **Range:** o

A.4.19 intSensorClass

- **Gyroscope** Mounts a gyroscope on a body.
  - name The name of the gyroscope.
    * **Use:** optional
    * **Range:** String
- **Accelerometer** Mounts an accelerometer on a body.
  - name The name of the accelerometer.
    * **Use:** optional
    * **Range:** String
- **CollisionSensor** A collision sensor which uses geometries to detect collisions with other objects.
  - name The name of the collision sensor.
    * **Use:** optional
    * **Range:** String

A.4.20 extSensorClass

- **Camera** Mounts a camera on a body.
  - name Name of the camera.
    - **Use:** optional
    - **Range:** String
  - imageWidth The width of the camera image.
    - **Default:** 0
    - **Use:** required
    - **Range:** Integer > 0
  - imageHeight The height of the camera image.
    - **Default:** 0
    - **Use:** required
    - **Range:** Integer > 0
  - angleX Opening angle in x.
    - **Units:** degree, radian
    - **Default:** 0
    - **Use:** required
    - **Range:** Float > 0
  - angleY Opening angle in y.
    - **Units:** degree, radian
    - **Default:** 0
    - **Use:** required
    - **Range:** Float > 0
- **SingleDistanceSensor**
  - name The name of the sensor.
    - **Use:** optional
    - **Range:** String
  - min The minimum distance this sensor can measure.
    - **Default:** 0
    - **Use:** optional
    - **Range:** $[-MAXFLOAT, MAXFLOAT]$
  - max The maximum distance this sensor can measure.
    - **Default:** 999999.f
    - **Use:** optional
    - **Range:** $[-MAXFLOAT, MAXFLOAT]$
- **ApproxDistanceSensor**
  - name The name of the sensor.
    - **Use:** optional
    - **Range:** String

  - min The minimum distance this sensor can measure.
    - **Default:** 0
    - **Use:** optional
    - **Range:** $[-MAXFLOAT, MAXFLOAT]$
  - max The maximum distance this sensor can measure.
    - **Default:** 999999.f
    - **Use:** optional
    - **Range:** $[-MAXFLOAT, MAXFLOAT]$
  - angleX The maximum angle in x-direction the ray of the sensor can spread.
    - **Units:** degree, radian
    - **Default:** 0
    - **Use:** required
    - **Range:** Float > 0
  - angleY The maximum angle in y-direction the ray of the sensor can spread.
    - **Units:** degree, radian
    - **Default:** 0
    - **Use:** required
    - **Range:** Float > 0

- **DepthImageSensor**
  - name
    - **Use:** optional
    - **Range:** String
  - imageWidth The width of the image.
    - **Default:** 0
    - **Use:** required
    - **Range:** Integer > 0
  - imageHeight The height of the image.
    - **Default:** 1
    - **Use:** optional
    - **Range:** Integer > 0
  - angleX
    - **Units:** degree, radian
    - **Default:** 0
    - **Use:** required
    - **Range:** Float > 0
  - angleY
    - **Units:** degree, radian
    - **Default:** 0
    - **Use:** required
    - **Range:** Float > 0
  - min The minimum distance this sensor can measure.

    - **Default:** 0
    - **Use:** optional
    - **Range:** $[-MAXFLOAT, MAXFLOAT]$
  - max The maximum distance this sensor can measure.
    - **Default:** 999999.f
    - **Use:** optional
    - **Range:** $[-MAXFLOAT, MAXFLOAT]$
  - projection The kind of projection.
    - **Default:** 0
    - **Use:** optional
    - **Range:** perspective, spheric

A.4.21 lightClass

- **Light** Definition of a light source.
  - diffuseColor Diffuse color definition, see A.4.22
  - ambientColor Ambient color definition, see A.4.22
  - specularColor Specular color definition, see A.4.22
  - x The x position of the light source.
    - **Units:** mm,cm,dm,m,km
    - **Use:** optional
    - **Range:** o
  - y The y position of the light source.
    - **Units:** mm,cm,dm,m,km
    - **Use:** optional
    - **Range:** o
  - z The z position of the light source.
    - **Units:** mm,cm,dm,m,km
    - **Use:** optional
    - **Range:** o
  - constantAttenuation The constant attenuation of the light.
    - **Use:** optional
    - **Range:** $[0.f, MAXFLOAT]$
  - linearAttenuation The linear attenuation of the light.
    - **Use:** optional
    - **Range:** $[0.f, MAXFLOAT]$
  - quadraticAttenuation The quadratic attenuation of the light.
    - **Use:** optional
    - **Range:** $[0.f, MAXFLOAT]$
  - spotCutoff
    - **Units:** mm,cm,dm,m,km

    - **Use:** optional
    - **Range:** $[-MAXFLOAT, MAXFLOAT]$
  - spotDirectionX The x direction of the light spot.
    - **Units:** mm,cm,dm,m,km
    - **Use:** optional
    - **Range:** $[-MAXFLOAT, MAXFLOAT]$
  - spotDirectionY The y direction of the light spot.
    - **Units:** mm,cm,dm,m,km
    - **Use:** optional
    - **Range:** $[-MAXFLOAT, MAXFLOAT]$
  - spotDirectionZ The z direction of the light spot.
    - **Units:** mm,cm,dm,m,km
    - **Use:** optional
    - **Range:** $[-MAXFLOAT, MAXFLOAT]$
  - spotExponent
    - **Use:** optional
    - **Range:** $[0.f, 128.f]$

A.4.22 Color Specification

There two ways of specifying a color for a color-attribute.

- **HTML-Style** To specify a color in html-style the first sign of the color value has to be a # followed by hexadecimal values for red, blue, green (and maybe a fourth value for the alpha-channel. These values can be one-digit or two-digits, but not mixed.
  - #rgb e.g. #f00
  - #rgba e.g. #0f0a
  - #rrggbb e.g. #f80011
  - #rrggbbaa e.g. #1038bc
- **CSS-Style** A css color starts with rgb (or rgba) followed by the values for red, green, blue put into brackets and separated by commas. The values for r,g,b has to be between 0 and 255 or between 0% and 100%, the a-value has to be between 0 and 1.
  - rgb(r,g,b) e.g. rgb(255, 128, 0)
  - rgba(r,g,b,a) e.g. rgba(0%, 50%, 75%, 0.75)

Appendix B

Compiling the Linux Kernel

The default Linux kernel bundled by Aldebaran has some disadvantages listed below:

LAN driver: The LAN driver freezes the system from time to time when copying a lot of files via *scp*.

WLAN driver: The WLAN driver is not thread safe and therefore does not allow SMP.

Hyper-Threading: SMP is disabled by default.

To solve the listed disadvantages, B-Human bundles its own Linux kernel. This kernel will be installed automatically with *installRobot* or *installRobot.usb*. The following guide will explain how to compile, package, and install the current version of the kernel.

B.1 Requirements

This guide assumes a freshly installed Ubuntu 13.04 (32 or 64 bit) and is not applicable on Windows or OS X. Furthermore, the following packages are necessary:

tar to extract the packaged kernel.

build-essential contains common tools such as *gcc*, *make*...

git for checking out the current version of the kernel.

wget to download the source files of the *gcc* in version 4.5.2, since it is not available in the repositories anymore.

gcc-multilib, libmpfr-dev, libmpc-dev to compile the downloaded *gcc*.

To resolve the dependencies run the following shell command:

```
sudo apt-get install build-essential git wget gcc-multilib libmpfr-dev libmpc-
    dev
```

B.2 Installing the Cross Compiler

The original Aldebaran kernel was built using gcc 4.5.2, therefore the custom kernel has to be built using the exact same version. This version is not available in the Ubuntu repositories anymore and has to be compiled manually. To do so, open a new shell and change to a directory where you have read and write access, for example:

```
cd /home/$USER/Documents
```

Download the necessary source files, extract them, and change into the directory:

```
wget ftp://ftp.fu-berlin.de/unix/languages/gcc/releases/gcc-4.5.2/gcc-4.5.2.
    tar.bz2
tar xfj gcc-4.5.2.tar.bz2
cd gcc-4.5.2
```

Due to the updated EGLIBC 2_16 in Ubuntu 13.04, gcc 4.5.2 does not compile without patches[1]. The necessary patch is available in the directory *Util/gcc-patch* of our code release. Apply it by running the following command:

```
patch -p0 -i /path/to/B-Human/Util/gcc-patch/gcc-4.5.2.patch
```

Furthermore, since Ubuntu 12.10 there seems to be a bug in the include paths[2]. The result is that the object file *crti.o* cannot be found during compilation. To fix this problem, the environment variable $LIBRARY_PATH has to be created or extended. Please make sure that the variable exists if you extend it. You can check whether it exists or not by running:

```
echo $LIBRARY_PATH
```

B.2.1 32 bit Ubuntu

Run the following command if the output was empty:

```
export LIBRARY_PATH=/usr/lib/i386-linux-gnu
```

or this command if the output was not empty:

```
export LIBRARY_PATH=/usr/lib/i386-linux-gnu:$LIBRARY_PATH
```

B.2.2 64 bit Ubuntu

Run the following command, if the output was empty:

```
export LIBRARY_PATH=/usr/lib/x86_64-linux-gnu
```

or this command if the output was not empty:

```
export LIBRARY_PATH=/usr/lib/x86_64-linux-gnu:$LIBRARY_PATH
```

Additionally on 64 bit Ubuntu, it is necessary to add symbolic links to the object files. **Note:** After the installation of the *gcc* the symbolic links should be removed.

```
sudo ln -s /usr/lib/x86_64-linux-gnu/crt*.o /usr/lib/
```

The compilation procedure of the *gcc* assumes that the output is stored in its own directory outside the source directory. Leave the source directory, create a new one and change into it.

[1] cf. http://sourceware.org/ml/libc-alpha/2012-03/msg00414.html
[2] cf. http://askubuntu.com/questions/251978/cannot-find-crti-o-no-such-file-or-directory

```
cd ..
mkdir gcc-4.5.2-build
cd gcc-4.5.2-build
```

Configure the *gcc* using following parameters:

–**prefix** defines the output directory for the gcc binary.

–**program-suffix** is useful if *–prefix* contains multiple *gcc* installations. It will be added to the name of the binary.

–**enable-languages** adds language support for the given languages.

Run the following command and set the parameters to your needs:

```
../gcc-4.5.2/configure --prefix=/path/to/gcc_dir [--program-suffix=-4.5.2] --
    enable-languages=c,c++
```

Start the compilation procedure and install the *gcc*. Use as many cores as your system provides by setting the parameter *-j*. The following example assumes 4 cores. If *–prefix* is not writable by the current user, add *sudo* to the installation target

```
make -j 4
[sudo] make install
```

The *gcc* is now ready to use. If created above, remove the symbolic links (64 bit Ubuntu only):

```
sudo rm /usr/lib/crt*.o
```

B.3 Compile/Package the Kernel and Modules

Again change into a directory, where you have read and write access, for example:

```
cd /home/$USER/Documents
```

Afterwards clone the GIT repository containing the current kernel sources and change into the directory

```
git clone https://github.com/bhuman/KernelV4.git kernel
cd kernel
```

To compile and package the kernel together with all modules, run the following command.

```
make CC=/path/to/gcc_dir/bin/gcc -j number-of-cores tarbz2-pkg
```

Afterwards the archive *linux-2.6.33.9-rt31-aldebaran-rt.tar.bz2* should be created inside the main directory of the kernel sources.

B.4 Replacing the Kernel

You now need to replace all files inside the *Install/Files/kernel* directory. First of all extract the *.tar.bz2* archive into a sub directory.

```
mkdir bhuman-kernel
tar xfj linux-2.6.33.9-rt31-aldebaran-rt.tar.bz2 -C bhuman-kernel
```

Afterwards copy all necessary files.

```
cp bhuman-kernel/boot/* /path/to/B-Human/Install/Files/kernel
cp -R bhuman-kernel/lib/modules/2.6.33.9-rt31-aldebaran-rt /path/to/B-Human/
    Install/Files/kernel
```

Finally remove the two unnecessary symbolic links *build* and *source* from *Install/Files/kernel/2.6.33.9-rt31-aldebaran-rt*

```
rm /path/to/bhuman/dir/Install/Files/kernel/2.6.33.9-rt31-aldebaran-rt/build
rm /path/to/bhuman/dir/Install/Files/kernel/2.6.33.9-rt31-aldebaran-rt/source
```

Now rerun the NAO installation procedure (cf. Sect. 2.4).