B-Human

Team Report and Code Release 2017

Thomas Röfer[1,2], Tim Laue[2],
Yannick Bülter[2], Daniel Krause[2], Jonas Kuball[2], Andre Mühlenbrock[2], Bernd Poppinga[2],
Markus Prinzler[2], Lukas Post[2], Enno Roehrig[2], René Schröder[2], Felix Thielke[2]

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

Revision: October 6, 2017

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 has won eight RoboCup German Open competitions, the RoboCup European Open 2016 competition, and has become RoboCup world champion six times.

After we regained the world champion title in 2016, we were able to defend this title at RoboCup 2017 in Nagoya, Japan. We won all seven matches in the *Champions Cup*, achieving a total goal difference of 34 : 1. Furthermore, we also won the new *Mixed Team Competition*, in which we teamed up with the *HULKs* from TU Hamburg-Harburg and formed the joint team *B-HULKs*. Our success at RoboCup 2017 was completed by winning this year's *Technical Challenge*, too.

The 2017 team consisted of the following persons (most of them are shown in Fig. 1.1):

Students: Yannick Bülter, Daniel Krause, Jonas Kuball, Lam Duy Le, Florian Maaß, Andre Mühlenbrock, Alicia Pagel, Bernd Poppinga, Lukas Post, Markus Prinzler, Jesse Richter-Klug, Enno Röhrig, Jurij Schmidt, René Schröder, Alexander Stöwing, Markus Strehling, Felix Thielke.

Active Alumni: Judith Müller, Andreas Stolpmann, Alexis Tsogias.

Leaders: Tim Laue, Thomas Röfer.

Associated Researcher: Udo Frese.

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. A short description of the changes to our system compared to last year has already been given in our Team Description Paper for RoboCup 2017 [23]. This document is based on the code release of the previous year [22] and aims to provide a complete description of the system that we used at RoboCup 2017. The major changes made to the system since last year are shortly enumerated in Section 1.3.

The remainder of this document is organized as follows: Chapter 2 gives a short introduction on how to build the code including the required software and how to run the NAO with our

Figure 1.1: The majority of the team members, celebrating after having won the RoboCup German Open 2017

software. Chapter 3 describes our framework's architecture. Our image processing approaches are presented in Chapter 4, followed by the state estimation approaches in Chapter 5. Chapter 6 explains the use of our behavior description language and gives an overview of the behavior used at RoboCup 2017. Surveys of the sensor reading and motion control parts of the system are given in Chapter 7 and Chapter 8 respectively. A brief description of our participation in the *Penalty Shot Challenge* and the *Mixed Team Competition* is given in Chapter 9. Finally, in Chapter 10, our simulation and remote control environment *SimRobot*, the *B-Human User Shell*, and some other tools are presented.

1.3 Major Changes Since 2016

The major changes made since RoboCup 2016 are described in the following sections:

2.3.2.1 Required Software
On Windows, it is now possible to use the *Windows Subsystem for Linux* as an alternative to *Cygwin*.

2.5 Copying the Compiled Code
Many parameters of the script *copyfiles* were renamed to improve consistency.

2.9 Configuration Files
The directory tree for configuration files now distinguishes between *robot*, *location*, and *scenario*.

3.4.7 Functions
Representations can now provide functionality. This has been widely used in the behavior

control, in which the so-called libraries are now (potentially switchable) modules that provide their functionality through representations.

3.5.4 Team Communication
The implementation of the communication between robots was majorly overhauled, not only for the cooperation with the team HULKs in the *Mixed Team Competition*, but also to simplify its general use.

4.3.5 Penalty Mark Perception
The penalty mark is now detected using a shape check based on a contrast-normalized Sobel edge image.

7.1 Ground Contact Recognition
The ground contact is now detected using the FSR sensors under the NAO's feet.

7.5 Detecting a Fall
Whether the robot is upright or falls is now primarily determined from the orientation of the support foot relative to the ground plane in a finite state machine.

8.3 Walking Engine
The Walk2014Generator developed by the team UNSW Australia [7] was integrated in our system.

8.4 Fall Engine
The FallEngine, which is executed when a fall is detected, controls the motion to prevent damage.

8.7 ZMP Balancing
The ZMPBalancer is a collection of parameterized balancing approaches used by our GetUpEngine.

9 *Technical Challenge* and *Mixed Team Competition*
In addition to the main soccer competition, we also participated in the *Penalty Shot Challenge* and – together with the HULKs – in the *Mixed Team Competition*.

10.4 Team Communication Monitor
The *GameStateVisualizer* was integrated into the *TeamCommunicationMonitor*.

10.5 Event Recorder
The *EventRecorder* is a semi-automatic logging software for matches, which was developed and integrated into the *GameController* project.

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 make on Linux, 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. In addition there are *generate* scripts that create the project files for Xcode, Visual Studio and CodeLite.

Src contains the source code of the B-Human software including the B-Human User Shell (cf. Sect. 10.2).

Util contains auxiliary and third party libraries (cf. Sect. 11) as well as our simulator SimRobot (cf. Sect. 10.1).

2.2 Components and Configurations

The B-Human software is usable on Windows, Linux, and macOS. It consists of two shared libraries for NAOqi running on the real robot, an additional executable for the robot, the same

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

bush is a tool to deploy and manage multiple robots at the same time (cf. Sect. 10.2).

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 macOS, the same source files are simply part of the library *Controller*.

Nao is the B-Human executable for the NAO. It depends on *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*, *SimulatedNao*, and some third-party libraries. 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 and macOS). *Develop* produces debuggable robot code while linking against non-debuggable but faster *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.

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

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 in Tab. 2.1.

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 that are needed to compile the components. The script

	without assertions (NDEBUG)	debug symbols (compiler flags)	debug libs[1] (_DEBUG, compiler flags)	optimizations (compiler flags)	debugging support[2]
Release					
Nao	✓	×	×	✓	×
SimulatedNao	✓	×	×	✓	✓
Develop					
Nao	×	×	×	✓	✓
SimulatedNao	×	✓	×	×	✓
Debug					
Nao	×	✓	✓	×	✓
SimulatedNao	×	✓	✓	×	✓

[1] - on Windows - `https://docs.microsoft.com/en-us/cpp/c-runtime-library/debug`
[2] - See Sect. 3.6

Table 2.1: Effects of the different build configurations.

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 an Xcode project for macOS). 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 needed when working with CodeLite. Building the code from the command line, via the provided Makefile, works without calling *generate* on Linux.

2.3.2 Visual Studio on Windows

2.3.2.1 Required Software

- Windows 10 64 bit or later
- Visual Studio 2017[1] or later
- A Unix base system. There are two alternatives:
  1. Windows Subsystem for Linux. Execute *Make/VS2017/installWSL.cmd*. The script will guide through the installation. It will first open a dialog to install the Windows Subsystem for Linux (unless it is already installed), then the Windows installer will ask for a reboot of the computer, and then the script has to be executed again to install the packages required.
  2. Cygwin x86 / x64 (available at `http://www.cygwin.com`) with the additional packages *rsync*, *openssh*, *ccache*, and *clang*. Let the installer add an icon to the start menu (the *Cygwin Terminal*). Add the *...\cygwin64\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 *C++ SDK 2.1.4 Linux 32* (*naoqi-sdk-2.1.4.13-linux32.tar.gz*) the script

[1] Visual Studio 2017 Community Edition Version 15.3 with only the "VC++ 2017 v141 Toolset (x86, x64)" and "Windows 8.1 SDK and UCRT SDK" installed is sufficient.

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.ald.softbankrobotics.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/VS2017/generate.cmd* and open the solution *Make/VS2017/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. 10.2). You may select *SimRobot* or *Utils/bush* as "StartUp Project".

2.3.3 Xcode on macOS

2.3.3.1 Required Software

The following components are required:

- macOS 10.12 or later
- Xcode 8.3 or later
- alcommon – For the extraction of the required alcommon library and compatible boost headers from the *C++ SDK 2.1.4 Linux 32* (*naoqi-sdk-2.1.4.13-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.ald.softbankrobotics.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*.[2] 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[3] 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.

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).[4] If the *login* script was used before to login to a NAO, the IP address used will be provided as default. In addition, the option `-b` is provided by default, which will restart the B-Human software on the NAO after it was deployed. Both the IP

[2] Xcode must have been executed at least once before to accept its license and to install its components.
[3] Note that the cross compiler actually builds code for Linux, although the scheme says "My Mac".
[4] Before you can do that, you have to setup the NAO first (cf. Sect. 2.4).

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/Includes/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.

2.3.3.3 Support for Xcode

Calling the script *Make/macOS/generate* also installs a lot of development support for Xcode:

Data formatters. If the respective file does not already exist, a symbolic link is created to formatters that let Xcode's debugger display summaries of several *Eigen* datatypes.

Source file templates. Xcode's context menu entry *New File...* contains a category *B-Human* that allows to create some B-Human-specific source files.

Code snippets. Many code snippets are available that allow adding standard constructs following B-Human's coding style as well as some of B-Human's macros.

Source code formatter. A system text service for formatting B-Human code is available to be used from Xcode's menu *Xcode→Services*.

2.3.4 Linux

The following has been tested and works on Ubuntu 17.04 64-bit. It should also work on other Linux distributions (as long as they are 64-bit); however, different or additional packages may be needed.

2.3.4.1 Required Software

The build has been tested using the software versions provided by the current Ubuntu distribution repositories. Earlier versions of, e. g., clang may work, but are untested.

Requirements (listed by common package names) for Ubuntu 17.04:

- clang
- qtbase5-dev
- libqt5svg5-dev
- libglew-dev
- libxml2-dev
- graphviz – Optional, for generating module graphs and the behavior graph.
- xterm – Optional, for opening an ssh session from the B-Human user shell *bush*.

- alcommon – For the extraction of the required alcommon library and compatible boost headers from the *C++ SDK 2.1.4 Linux 32* (*naoqi-sdk-2.1.4.13-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.ald.softbankrobotics.com` (account required). Please note that this package is only required to compile the code for the actual NAO robot.

On Ubuntu 17.04, you can execute the following command to install all requirements except for *alcommon*:

```
sudo apt install clang qtbase5-dev libqt5svg5-dev libglew-dev libxml2-dev
    graphviz xterm
```

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
```

to build the whole solution or

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

to build single components.

To clean up the whole solution, use:

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

As an alternative, there is also support for the integrated development environment CodeLite that works 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 or later is required to open the workspace generated. Older versions might crash. The latest reported compatible version of CodeLite is 10.0.0.

2.4 Setting Up the NAO

2.4.1 Requirements

First of all, download the atom system image, e. g. version 2.1.4 (*opennao-atom-system-image-2.1.4.13_2015-08-27.opn*), and the *Flasher*, e. g. version 2.1.0, for your operating system from the download area of `https://community.ald.softbankrobotics.com` (account required). In order to flash the robot, you 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[5]:

sed, *rsync*.

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

[5] In the unlikely case that they are missing in a Linux distribution, execute *sudo apt-get install sed openssh-clients*. On Windows and macOS, they are already installed at this point.

The commands in this chapter are shell commands. They must be executed inside a Unix shell, i. e. on Windows, you have to start *bash* first. All shell commands should be executed from the *Install* directory.

2.4.2 Installing the Operating System

After the robot specific configuration files were created (cf. Sect. 2.4.3 and Sect. 2.4.4), plug in your USB flash drive and start the *NAO flasher tool*[6]. Select the *opennao-atom-system-image-2.1.4.13.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 that is switched off 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 network for DHCP. Once the reboot is finished, the NAO will do its usual wake-up procedure. Now the NAO will say its current IP address by pressing the chest button.

2.4.3 Creating Robot Configuration Files for a NAO

Before you start to set up the NAO, you need to create configuration files for each robot you want to set up. To create the configuration files, run *createRobot* followed by *addRobotIds* in the *Install* directory. The first 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 identifies the robot and is used in the system to load robot specific configurations. Furthermore, it is used as the host name of the NAO operating system. The second file creates a table associating the *headId* and *bodyId* of each NAO to the name used by *createRobot*. These ids are the serial-numbers SoftBank Robotics uses for the NAO. Apart from the name this script expects either those ids, typed in manually, or the current ip-address of the NAO, in which case the ids will be loaded from the robot.

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. It is assumed that the robot is already connected via an ethernet connection and has reported its IP address to be 169.254.54.28 (via pressing the chest button):

```
cd Install
./createRobot -t 3 -r 25 Penny
./addRobotIds -ip 169.254.54.28 Penny
```

If the NAO is not available, the serial numbers can also be specified manually:

```
./addRobotIds -ids ALDxxxxxxxxxxx ALDxxxxxxxxxxx Penny
```

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

Help for both scripts is available using the option *-h*. Running *createRobot* creates all needed files to install the robot. This script also creates a directory with the robot's name in *Config/Robots*. *addRobotIds* will store the table in *Config/Robots/robots.cfg*.

Note: When upgrading from an older B-Human code release running *createRobot* is not necessary. Nevertheless, the script *addRobotIds* has to be executed for robots that were installed with code releases before 2016.

2.4.4 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
```

Another way to switch between different configurations is by using the tools *copyfiles* (cf. Sect. 2.5) or *bush* (cf. Sect. 10.2).

2.4.5 Installing the Robot

Finally, the script *installRobot* has to be executed in order to prepare the robot for the B-Human software. This script only expects the current IP address of the robot. For example run:

```
./installRobot 169.254.54.28
```

Follow the instructions on the screen until the robot reboots.[7]

Now you can use *copyfiles* (cf. Sect. 2.5) or *bush* (cf. Sect. 10.2) to copy compiled code and configuration files to the NAO.

2.5 Copying the Compiled Code

The script *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 macOS, you can use your IDE to use *copyfiles*. In Visual Studio, you can run the script by "building" the project *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>*.

[7] You only will be asked for the password *nao* if the ssh key has not been copied yet, i. e. if neither *addRobotIds -ip* nor *installRobot* ran before for this robot.

copyfiles requires two mandatory parameters. First, the configuration the code was compiled with (*Debug*, *Develop*, or *Release*)[8], and second, the IP address of the robot. To adjust the desired settings, it is possible to set the following optional parameters:

Option	Description
-b	Restarts *bhuman* (and *naoqi* if necessary) after copying.
-c <color>	Sets the team color to *blue*, *red*, *yellow*, *black*, *white*, *green*, *orange*, *purple*, *brown*, or *gray* replacing the value in the *settings.cfg*.
-d	Removes all log files from the robot's */home/nao/logs* directory before copying files.
-h | --help	Prints the help.
-l <location>	Sets the location, replacing the value in the *settings.cfg*.
-m <number>	Sets the magic number. Robots with different magic numbers will ignore each other when communicating.
-n	Stops *naoqi*.
-nc	Never compiles, even if binaries are outdated.
-nr	Does not check whether the robot to deploy to is reachable.
-o <port>	Overwrite team port (default is 10000 + team number).
-p <number>	Sets the player number, replacing the value in the *settings.cfg*.
-r <n> <ip>	Copies to IP address <ip> and sets the player number to *n*. This option can be specified more than once to deploy to multiple robots.
-s <scenario>	Sets the scenario, replacing the value in the *settings.cfg*.
-t <number>	Sets team number, replacing the value in the *settings.cfg*.
-v <percent>	Set NAO's sound volume.
-w <profile>	Set wireless profile.

Possible calls could be:

```
./copyfiles Develop 134.102.204.229 -t 5 -c blue -p 3 -b
./copyfiles Release -r 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. 10.2) can be used to copy the compiled code to several robots at once.

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 one executable (*bhuman*), which is 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/Includes/connect.con*. Thus a later use of the SimRobot scene *RemoteRobot.ros2* will automatically connect to the same robot. On macOS, 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.

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

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. In case *libbhuman* or *libgamectrl* were updated, *copyfiles* restarts NAOqi automatically.

bhuman executes the *bhuman* executable 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.

bhumand start|stop|restart starts, stops or restarts the *bhuman* executable. *Copyfiles* always stops *bhuman* before deploying. If *copyfiles* is started with option *-r*, it will restart *bhuman* after all files were copied.

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.

2.7 Starting SimRobot

On Windows and macOS, SimRobot can either be started from the development environment or by starting a scene description file in *Config/Scenes*[9]. 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 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.

[9] 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 macOS, starting a scene description file bears the risk of executing a different version of SimRobot than the one that was just compiled.

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.[10] 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 coordinates to world coordinates (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 four 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 up the robot (cf. Sect. 2.8.2). In some cases it may be necessary to repeat these steps, because big changes in the joint calibration may invalidate the camera calibration.

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. 8.3.4). 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
```

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

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

When the calibration is finished it should be saved:

```
save representation:JointCalibration
```

Manually Adjusting Joint Offsets

First of all, the robot has to be switched to a stationary stand, otherwise the balancing mechanism of the motion engine might move the legs, messing up the joint calibration:

```
mr StandArmRequest CalibrationStand
mr StandLegRequest CalibrationStand
```

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. 10.1.4.5)

```
vd representation:JointCalibration
```

which is more comfortable.

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* and switch to the CalibrationStand:

```
call Calibrators/Joint
```

When a completely new calibration is desired, 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 degrees.

Straightening Up the NAO

The camera calibration (cf. Sect. 2.8.3) also calculates a rotation for the body rotation. These values can be passed to the JointCalibrator that will then set the NAO in an upright position. Call:

```
get representation:CameraCalibration
call Calibrators/Joint
```

(a) (b)

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

Copy the values of *bodyRotationCorrection* (representation *CameraCalibration*) into *bodyRotation* (representation *JointCalibration*). Afterwards, set *bodyRotationCorrection* (representation *CameraCalibration*) to zero. Another way to make these actions more or less automatically is possible by using the AutomaticCameraCalibrator with the automation flag (cf. Sect. 2.8.3).

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

For calibrating the cameras (cf. Sect. 4.1.2.1) using the module AutomaticCameraCalibrator, 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 *Calibrators/Camera.con* (in the console type *call Calibrators/Camera*). This will initialize the calibration process and furthermore print commands or help 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 `AutomaticCameraCalibrator` module (e. g. for setting the robot's position to the penalty mark of a field, type *set module:AutomaticCameraCalibrator:robotPose rotation = 0; translation = {x = -3200; y = 0;};* in the console).

4. To automatically generate the commands for the following joint calibration to correct the body rotation, you can set a flag via *set module:AutomaticCameraCalibrator:setJointOffsets true*. After you finished the optimization you can just enter the generated commands and thereby correct the rotation.

(a) (b) (c)

Figure 2.2: The three interesting camera calibration stages. a) is the start of the calibrator. b) is the view after the control start with gathered samples. c) is the stage after optimization.

5. To start the point collection use the command *dr module:AutomaticCameraCalibrator:start* and wait for the output "Accumulation finished. Waiting to optimize...". The process includes both cameras and will collect samples for the calibration and make the head motions to cover the whole field. The samples for the upper camera are drawn blue and the samples for the lower camera red. A drawing above the images signalizes if the sample amount is sufficient for optimization (green) or not (red).

6. If you are unhappy with the collection of some specific samples you are now able to delete samples by left-clicking onto the sample in the image in which it has been found. If there are some samples missing you can manually add them by *Ctrl* + left-clicking into the corresponding image.

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

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:

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

exposure: The exposure used. The available interval is [0, 1000]. Usually, an exposure of 140 is used, which equals 14 ms. Be aware that high exposures lead to blurred images.

gain: The gain used. The available interval is [0, 255]. Usually, the gain is set to 50 - 70. Be aware that high gain values lead to noisy images.

autoWhiteBalance: Enable(1) / disable(0) the automatism for white balance. This parameter should always be disabled since a change in the white balance can change the color and mess up the color calibration. On the other hand, a real change in the color temperature of the environment will have the same result.

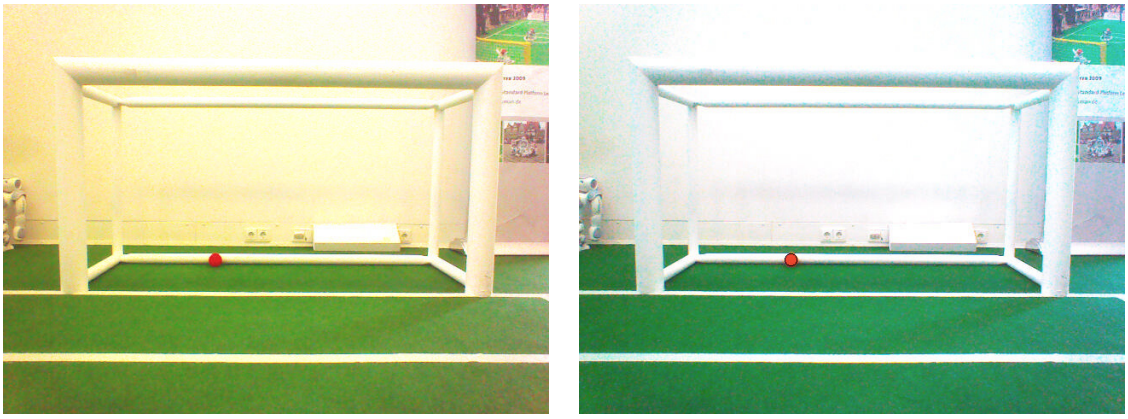


Figure 2.3: The left figure shows an image with improper white balance. The right figure shows the same image with better settings for white balance.

autoExposure: Enable (1) / disable (0) the automatism for exposure. When playing under static light conditions such as in the standard indoor tournament, this parameter should always be disabled, since the automation will often choose higher values than necessary, which will result in blurry images. However, for dynamic light conditions as were present in the Outdoor Competition at RoboCup 2016, using the automatism of the camera driver may be a necessity. In this case, its behavior can be altered using the parameters autoExposureAlgorithm and brightness.

The camera driver can do a one-time auto white balance. This feature can be triggered with the commands:

```
dr module:CameraProvider:doWhiteBalanceUpper
dr module:CameraProvider:doWhiteBalanceLower
```

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 views with the segmented upper and lower camera images and the color calibration view. See (cf. Sect. 10.1.4.1 and Sect. 10.1.4.1). 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 *cameraSettingsV5.cfg* (or *cameraSettingsV4.cfg* if the NAO is a V4 model) and *fieldColorsCalibrationV5.cfg* (or *fieldColorsCalibrationV4.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[11] are located below the directory *Config*.

Scenarios 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/Scenarios* and placing configuration files in it. Those configuration files are only taken into account if the scenario is activated in the file *Config/settings.cfg*.

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

Locations can be used to configure the software for use in different locations, e. g. in the lab at home, and at different competitions. For instance, the field dimensions and the color calibration can depend on the location the robots are used in.

Robots. Besides the global configuration files, there are some files which depend on the robot's head, body, or both. To differentiate the locations of these files, the names of the head and the body of each robot are used. They are defined in the file *Config/Robots/robots.cfg* that maps the serial numbers of the heads and the bodies of the robots to their actual names. In the Simulator, both names are always "Nao".

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/<head name>/Head/<filename>*
   - Used for files that only depend on the robot's **head**
   - e.g.: *Robots/Amy/Head/cameraIntrinsics.cfg*
2. *Config/Robots/<body name>/Body/<filename>*
   - Used for files that only depend on the robot's **body**
   - e.g.: *Robots/Alex/Body/walkingEngine.cfg*
3. *Config/Robots/<head name>/<body name>/<filename>*
   - Used for files that depend on both, the robot's head **and** body.
   - e.g.: *Robots/Amy/Alex/cameraCalibration.cfg*
4. *Config/Locations/<current location>/<filename>*
5. *Config/Scenarios/<current scenario>/<filename>*
6. *Config/Robots/Default/<filename>*
7. *Config/Locations/Default/<filename>*
8. *Config/Scenarios/Default/<filename>*
9. *Config/<filename>*

So, whether a configuration file is robot-dependent, location-dependent, scenario-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 [21] is based on the framework of the GermanTeam 2007 [20], 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 actuators and sensors (except the camera) of the NAO are accessed using 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 GameController (cf. Sect. 10.3) 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

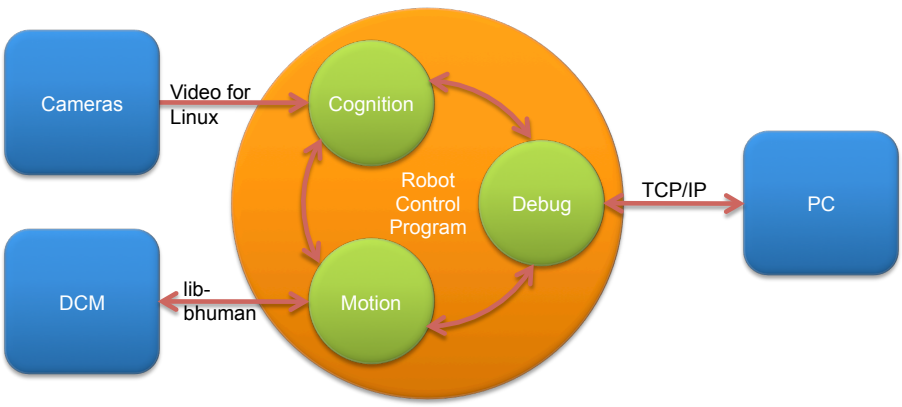


Figure 3.1: The processes used on the NAO

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 25 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 [20] 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. 10.1.4.5).

3.3.1 Blackboard

The blackboard [8] 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 is a map that associates names of representations to reference-counted instances of these representations. It only contains entries for the representations that at least one of the modules in associated process actually requires or provides.

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_PARAMETERS(
  {,
    (Vector2f)(5.f, 0.f) offset,
    (float)(1.1f) scale,
  }),
});
```

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

MAKE_MODULE(SimpleBallLocator, modeling);
```

The module interface defines the name of the module (e. g. `SimpleBallLocator`), the representations that are required to perform its task, the representations provided by the module, and the parameters of the module, the values of which can either be defined in place or loaded from a file. 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. It also inherits all parameters. 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. 10.1.4.5). 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 `cognitionInfrastructure`, `communication`, `perception`, `modeling`, and `behaviorControl` are available and in process *Motion* the categories `motionInfrastructure`, `motionControl`, 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.

`MODULE` is a macro that gets all the information about the module as parameters, i. e. they are all separated by commas. The macro ignores its second and its last parameter, because by convention, these are used for opening and closing curly brackets. These let some source code formatting tools to indent the definitions as a block. Currently, `MODULE` is limited to up to 80 definitions between the curly brackets. When the macro is expanded, it creates a lot of hidden functionality. Each entries that references a representation makes sure that it is created in the blackboard when the module is constructed and freed when the module is deconstructed. 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. On a host PC the information can be used to change the configuration and for visualization (cf. Sect. 10.1.4.5).

For each representation provided with `PROVIDES` the execution time can be determined (cf. Sect. 3.6.7) and it can be sent to a host PC or even altered by it. If the latter is not desired, `PROVIDES_WITHOUT_MODIFY` can be used instead. If a `MesssageID id`<`representation`> exists, the representation can also be logged.

If a representation provided defines a parameterless method `draw`, that method will be called after the representation was updated. The method is intended to visualize the representation using the techniques described in Sect. 3.6.3. If the representation defines a parameterless

method `verify`, that method will be called in *Debug* and *Develop* builds after the representation was updated as well. A `verify` method should contain `ASSERT`s that check whether the contents of the representation are plausible. Both methods are only called if they are defined in the representation itself and not if they are inherited from its base class.

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/Scenarios/<scenario>/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. 10.1.6.3).

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. 10.1.6.3).

There are two different parameter initialization methods. In the hard-coded approach, the initialization values are specified as part of the C++ source file. They are defined using the

`DEFINES_PARAMETERS` macro. This macro is intended for parameters that may change during development but will never change again afterwards. In contrast, loadable parameters are initialized to values that are loaded from a configuration file upon module creation, i.e. the initialization values are not specified in the source file. These parameters are defined using the `LOADS_PARAMETERS` macro. 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.

Only either `DEFINES_PARAMETERS` or `LOADS_PARAMETERS` can be used in a module definition. They can both only be used once. Their syntax follows the definition of generated streamable classes (cf. Sect. 3.4.4). 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 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.

[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 `LEDHandler` would read the file *ledHandler.cfg* if it would read its parameters from a file.

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 Range<T>& range)
{
  STREAM_REGISTER_BEGIN_EXT(range);
  STREAM_EXT(out, range.min);
  STREAM_EXT(out, range.max);
  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 the past. 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. Two macros automatically generate all this code for a streamable `struct`[3] and optionally also initialize its member variables. The first is:

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

The second is very similar:

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

The parameters have the following meaning:

class: The name of the struct to be declared.

[3]Meaning, the default access for members is `public`.

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. A workaround is to use the macro `COMMA` instead of an actual comma. However, the use of that macro should be avoided if possible, e. g. by defining constructors with comma-separated initializer lists outside of the `struct`'s body.

comma-separated-declarations: Declarations of the streamable attributes[4] in four possible forms[5]:

```
(<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 `std::array` or `std::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, e. g., generate `C::std::vector<E>` instead of `std::vector<C::E>`, which does not compile.

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,
{
  ENUM(ABC,
  {,
    a,
    b,
    c,
  });

  Example()
  {
    std::memset(array, 0, sizeof(array));
  },

  (int) anInt,
  (float)(3.14f) pi,
  (int[4]) array,
  (Vector2f)(1.f, 2.f) aVector,
```

[4] Currently, the macros support up to 60 entries.

[5] The spaces are actually important when compiling on Windows.

```
    (ABC) aLetter,
    ((MotionRequest) Motion)(stand) motionId,
  });
```

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

The macros can even be used if the `serialize` method should do more than just streaming the member variables. If a method `onRead()` is defined within the `struct`, it will be called at the end of `serialize` if data was read allowing to implement some post-processing, e. g. to compute the values of member variables that are not streamed from the values of other member variables that were.

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,
{,
  (Pose2f) defensiveGoaliePose,
  (std::vector<Vector2f>) 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 *Src/Tools/Streams/Enum.h* rather than using the C++ `enum` keyword directly. The macro's first parameter is the name of the enumeration type. The second and the last parameter are reserved for curly brackets and are ignored. All other 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. `static const unsigned char 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 `nullptr`.

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

3.4.6.1 Iterating over Enumerations

It is often necessary to enumerate all constants defined in an enumeration type. This can easily be done using the `FOREACH_ENUM` macro:

```
FOREACH_ENUM(Letter, letter)
{
  // do something with "letter", which is of type "Letter"
}
```

It is also possible to specify a different upper limit to only enumerate a part of the constants. The upper limit must be one of the constants defined:

```
FOREACH_ENUM(Letter, letter, numOfLettersBeforeC)
{
  // do something with "letter", which is of type "Letter"
}
```

If the enumeration type is defined in another class, that class needs to be specified separately:

```
FOREACH_ENUM((MotionRequest) Motion, motion)
{
  // do something with "motion", which is of type "MotionRequest::Motion"
}
```

In that case, an optional upper limit would automatically be assumed to be defined in the other class as well, i.e. it must not be prefixed by the classes' name.

3.4.6.2 Enumerations as Array Indices

In the B-Human code, enumerations are often used as indices for arrays, because this gives entries a name, but still allows to iterate over these entries. However, in configuration files and in the UI, it is hard to find specific entries in arrays, in particular if they have a larger number of elements. For instance, the arrays of all joint angles has 26 elements, making it hard to identify the angle of a specific joint. Therefore, a special macro (defined in *Src/Tools/Streams/EnumIndexedArray.h*) that is backed by a template class allows to define an array indexed by an enumeration type that solves this problem by streaming such an array as if it were a structure with a member variable for each of its elements named after the respective enumeration constant. Technically, such an array is derived from `std::array` and simply defines the method `serialize` (cf. Sect. 3.4.3). For example, an array of all joint angles could be defined by using the enumeration `Joints::Joint` as index:

```
ENUM_INDEXED_ARRAY(Angle, (Joints) Joint) jointAngles;
```

`jointAngles` still behaves like an array, i. e. it is derived from `std::array<Angle, Joints::numOfJoints>`, but when, e. g., written to a file using `OutMapFile` (cf. Sect. 3.4.5), it would appear differently:

```
headYaw = 0deg;
headPitch = 0deg;
lShoulderPitch = 0deg;
...
```

The class prefix of the enumeration type used as an index only needs to be specified if that type was defined in another class. However, if the elements of the array are also of an enumeration type, that type must be fully specified, i. e. including the class prefix, even if that type is defined in the same class as the array:

```
ENUM_INDEXED_ARRAY((MotionRequest) Motion, Letter) someArray;
```

3.4.7 Functions

Until last year, the representations in the B-Human module framework have only forwarded data from a module to other modules. Now, representations can also provide *functionality*, i. e. they can contain functions, which have an implementation that is provided by a module.

So far, all data in a representation had to be computed before it was used. This lead to a certain overhead, because the data has to be computed in advance without knowing whether it will actually be required in the current situation. In addition, there is a lot of data that cannot be computed in advance, because its computation depends on external values. For instance, a path planner must know the target to which it has to plan a path before it can be executed. However, in many situations a path planner is not needed at all, because the motion is generated reactively.

Functions in representations allow modules that require the representation to execute code of the module that provided that representation at any time and they allow to pass parameters to that implementation. For instance, while the behavior control of the 2016 B-Human system still contained many so-called libraries that could compute all kinds of information on demand as a fixed part, they are now simply representations that are provided by regular B-Human modules that implement the logic behind the interface. These modules can be switched to other implementations as all other B-Human modules can, giving a greater flexibility when improving the functionality of the system.

These functions are based on `std::function` from the C++ runtime library and the assigned implementations are usually lambda expressions.

However, to make them more compliant with the general B-Human architecture, they differ from the normal standard implementation in their behavior if no value was assigned to them (their *default* behavior). Instead of throwing an exception as `std::function` would do when called, they simply do nothing. If they have a return value, they return the default value of the type returned or `Zero()` in case of Eigen types.

However, there are some specialties to functions in representations:

- Functions cannot be streamed in any way, although they must be part of a class that is derived from the class `Streamable`. This also means that a module cannot call a function of a representation the provider of which is executed in another framework process. However, representations containing functions can be streamed, but the functions cannot be part of the data that is streamed, i. e. they must be ignored during streaming.
- Representations containing functions are treated in a special way. They are always reset when their provider is switched. The reason is that the function object would otherwise still contain a reference to the module that originally provided the implementation, but the module does not exist anymore.

For instance, the path planner provides its functionality through the representation *PathPlanner*:

```
STREAMABLE(PathPlanner,
{
  FUNCTION(MotionRequest(const Pose2f& target, const Pose2f& speed,
           bool excludePenaltyArea)) plan,
});
```

It assigns the function in its method `update`:

```
pathPlanner.plan = [this](const Pose2f& target, const Pose2f& speed,
```

```
                                bool excludePenaltyArea) -> MotionRequest
{ // ...
};
```

In the behavior control, the path planner is executed by this call:

```
theMotionRequest = thePathPlanner.plan(target, Pose2f(speed, speed, speed),
    avoidOwnPenaltyArea);
```

3.5 Communication

Three kinds of communication are implemented in the B-Human framework: *inter-process communication*, *debug communication*, and *team communication*.

3.5.1 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.

3.5.2 Message Queues

The *debug communication*, the *team communication*, and *logging* are all based on the same technology: *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.

3.5.3 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 10.1.6.3.

The debug communication and the process *Debug* are not available on the actual NAO when the software deployed was compiled in the configuration *Release*.

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 via UDP, so all teammates can receive them. Sending and receiving team messages is done in the process *Cognition* using the representations *TeamData* for handling received messages and *BHumanMessageOutputGenerator* for generating messages to send. These representations are both generated and filled with their functionality by the module TeamMessageHandler.

The format of team messages is given by the `SPLStandardMessage`, which consists of a standardized and a non-standard part. For our communication, the TeamMessageHandler fills in the standardized part with appropriate values from representations and writes additional data in form of a message queue in the non-standardized part of the packet. Because of our cooperation with the HULKs in the Mixed Team Competition, the message queue in the non-standardized part is preceded by a defined structure called `BHULKsStandardMessage`, which contains data that we communicate with the HULKs.

In order to be transmitted by the TeamMessageHandler, a representation has to extend the struct `BHumanMessageParticle` and provide implementations for reading and writing its data to or from a message. If a representation is only supposed to be included in the message queue and does not fill in any standardized fields of the `SPLStandardMessage` or `BHULKsStandardMessage`, it can also just extend the struct `PureBHumanArbitraryMessageParticle`.

According to the rules, team communication packets are only broadcasted every 200 ms. The *BHumanMessageOutputGenerator* contains a flag set by the TeamMessageHandler that states whether a team communication packet will be sent out in the current frame or not. The TeamMessageHandler also implements the network time protocol (*NTP*) and translates time stamps contained in the messages it receives into the local time of the robot.

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. These mechanisms are available in all project configurations except for *Release* on the actual NAO.

3.6.1 Debug Requests

Debug requests are used to enable and disable parts of the source code. They can be seen as runtime switches for debugging.

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. 10.1.6.3). The following macros ease the use of the mechanism as well as hide the implementation details:

DEBUG_RESPONSE(<id>) executes the following statement or block if the debug request with the name `id` is enabled.

DEBUG_RESPONSE_ONCE(<id>) executes the following statement or block *once* when the debug request with the name `id` is enabled.

DEBUG_RESPONSE_NOT(<id>) executes the following statement or block if the debug request with the name `id` is *not* enabled.

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. 10.1.6.3).

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. 10.1.4.1) or in a color space view (cf. Sect. 10.1.4.1). Each debug image consists of pixels in one of the formats *RGB*, *BGRA*, *YUYV*, *YUV*, *Colored* and *Grayscale*, which are defined in *Src/Tools/ImageProcessing/PixelTypes.h*. Except for *Colored* and *Grayscale*, pixels of these formats are all made up of four unsigned byte values, each representing one of the color channels of the format. The *YUYV* format is special as one word of the debug image describes two image pixels by specifying two luminance values, but only one value per U and V channel. Thus, it resembles the *YUV422* format of the images supplied by the NAO's cameras. Debug images in the *Grayscale* format only contain a single channel of unsigned byte values describing the luminance of the associated pixel. The *Colored* format consists of only one channel, too. Values in this channel are entries of the `FieldColors::Color` enumeration which contains identifiers for the color classes used for image processing (cf. Sect. 4.1.4).

Debug images are supposed to be declared as instances of the template class `TImage`, instantiated with one of the pixel formats named above, or the `Image` class which is otherwise only used for the camera image. The following macros are used to transfer debug images to a connected PC.

SEND_DEBUG_IMAGE(<id>, <image>, [<method>]) sends the debug image to the PC. The identifier given to this macro is the name by which the image can be requested. The optional parameter is a pixel type that allows to apply a special drawing method for the image.

COMPLEX_IMAGE(<id>) only executes the following statement if the creation of a certain debug image is requested. For debug images that require complex instructions to paint, it can significantly improve the performance to encapsulate the drawing instructions in this macro (and maybe additionally in a separate method).

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

```
class Test
{
private:
  TImage<GrayscaledPixel> testImage;

public:

  void doSomething() {
    // [...]
    COMPLEX_IMAGE("test") draw();
    // [...]
  }

  void draw()
  {
    testImage.setResolution(640, 480);
    memset(testImage[0], 0x7F, testImage.width * testImage.height);
    SEND_DEBUG_IMAGE("test", testImage);
  }
};
```

The example calls the `draw()` method if the `"test"` image was requested, which then initializes a grayscale debug image, paints it gray and sends it 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. 10.1.4.1) and all drawings of type "drawingOnField" can be rendered into a field view (cf. Sect. 10.1.4.1).

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 (`solidPen`, `dashedPen`, `dottedPen`, and `noPen`) and fill styles (`solidBrush` and `noBrush`) are part of the namespace `Drawings`. Colors can be specified as `ColorRGBA`. The class also contains a number of predefined colors such as `ColorRGBA::red`. A few examples for drawing macros are:

DECLARE_DEBUG_DRAWING(<id>, <type>) declares a debug drawing with the specified *id* and *type*.

COMPLEX_DRAWING(<id>) only executes the following statement or block 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.

DEBUG_DRAWING(<id>, <type>) is a combination of DECLARE_DEBUG_DRAWING and COMPLEX_DRAWING. It declares a debug drawing with the specified *id* and *type*. It also executes the following statement or block if the debug drawing is requested.

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.

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::solidPen, ColorRGBA::blue,
                               Drawings::solidBrush, 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>). 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 SoftBank Robotics' 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.

COMPLEX_DRAWING3D(<id>) only executes the following statement or block if the creation of the debug drawing is requested (similar to `COMPLEX_DRAWING(<id>)` for 2-D drawings).

DEBUG_DRAWING3D(<id>, <type>) is a combination of `DECLARE_DEBUG_DRAWING3D` and `COMPLEX_DRAWING3D`. It declares a debug drawing with the specified *id* and *type*. It also executes the following statement or block if the debug drawing is requested.

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, ColorRGBA::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. 10.1.3) by sending a debug request (cf. Sect. 10.1.6.3) 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. The rendering of debug drawings can be configured for individual scene views by right-clicking on the view and selecting the desired type of visualization in the "Drawings Rendering" submenu.

3.6.5 Plots

The macro `PLOT(<id>, <number>)` allows plotting data over time. The plot view (cf. Sect. 10.1.4.5) 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. 10.1.6.3).

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", theInertialSensorData.gyro.y());
```

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. 10.1.6.3). 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);
MotionRequest m;
MODIFY("representation:MotionRequest", m);
```

The macro `PROVIDES` of the module framework (cf. Sect. 3.3) includes the `MODIFY` macro for the representation provided. For instance, if a representation *Foo* is provided by `PROVIDES(Foo)`, it is modifiable under the name `representation:Foo`. If a representation provided should not be modifiable, e. g., because its serialization does not register all member variables, it must be provided using `PROVIDES_WITHOUT_MODIFY`.

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 macro `STOPWATCH(<id>)` (declared in *Src/Tools/Debugging/Stopwatch.h*) measures the runtime of the statement or block that follows. *id* 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. 10.1.4.5). By default, a stopwatch is already defined for each representation that is currently provided and for the whole processes *Cognition* and *Motion*.

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

```
STOPWATCH("myCode") myCode();
```

Often rudimentary statistics (cf. Sect. 10.1.4.5) 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 `STOPWATCH_WITH_PLOT(<id>)` 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* and *Motion* processes and is designed to log representations in real-time.

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 [6]. The name of the log file consists of the name of the process it logs, the names of the head and body of the robot, the scenario, and the location. If connected to the GameController, the name of the opponent team, the half, and the player number are also added to the log file name. Otherwise, "Testing" is used instead. If a log file with the given name already exists, a number is added that is incremented for each duplicate.

To retain the real-time properties of the processes, the heavy lifting, i. e. compression and writing of the file, is done in separate threads without real-time scheduling, one for *Cognition* logging and one for *Motion* logging. These threads use every bit of remaining processor time that is not used by one of the real-time parts of the system. Communication with these threads is realized using very large ring buffers (usually around 500MB). Each element of the ring buffers represents one second of data. If a buffer is full, the current element is discarded. Due to the size of the buffers, writing of log files might continue for quite a while 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* (cf. Sect. 3.7.7) or *image patches* (cf. Sect. 3.7.8) are generated and used instead.

In addition to the logged representations, online log files also contain timing data, which can be seen using the TimingView (cf. Sect. 10.1.4.5).

3.7.2 Configuring the Online Loggers

The online loggers can be configured by changing values in the files *logger<Process>.cfg*, which should be located inside the configuration folder.

Following values can be changed:

log filePath: 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: An online logger is enabled if this is true. Note that it is not possible to enable online loggers inside the simulator.

writePriority: Priority of the logging process. Priorities greater than zero use the real-time scheduler, zero uses the normal scheduler, and negative values (-1 and -2) use idle priorities.

debugStatistics: If true a 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.

```
dr off
dr representation:Image
dr representation:JointAngles
dr representation:JointSensorData
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

In general, log files consist of serialized message queues (cf. Sect. 3.5.2). Each log file consists of up to three chunks. Each chunk is prefixed by a single byte defining its type. The enum `log fileFormat` in *Src/Tools/Logging/log fileFormat.h* defines these chunk identifiers in the namespace `Logging` that have to appear in the given sequence in the log file if they appear at all:

logFileMessageIDs: This chunk contains a string representation of the `MessageIDs` (cf. Sect. 3.5.2) stored in this log file. It is used to convert the `MessageIDs` from the log file to the ones defined in the version of *SimRobot* (cf. Sect. 10.1) that is replaying the log file. Thereby, log files still remain usable after the enumeration `MessageID` was changed.

logFileStreamSpecification: This chunk contains the specification of all datatypes used in the log file. It is used to convert that data logged to the specifications that are defined in the version of *SimRobot* that is replaying the log file. If the specification changed, messages will appear in SimRobot's console about the representations that are converted. Please note that a conversion is only possible for representations the specification of which is fully registered. This is not the case for representations that use `read` and `write` methods to serialize their data, e. g. `Image`, `JPEGImage`, `Thumbnail`, and`ImagePatches`. Therefore, such representations cannot be converted and will likely crash SimRobot when trying to replay log files containing them after they were changed.

logFileUncompressed: Uncompressed log data is stored in a single `MessageQueue`. Its serialized form starts with an eight byte header containing two values. These are the size used by the log followed by the number of messages in the queue. Those values can also

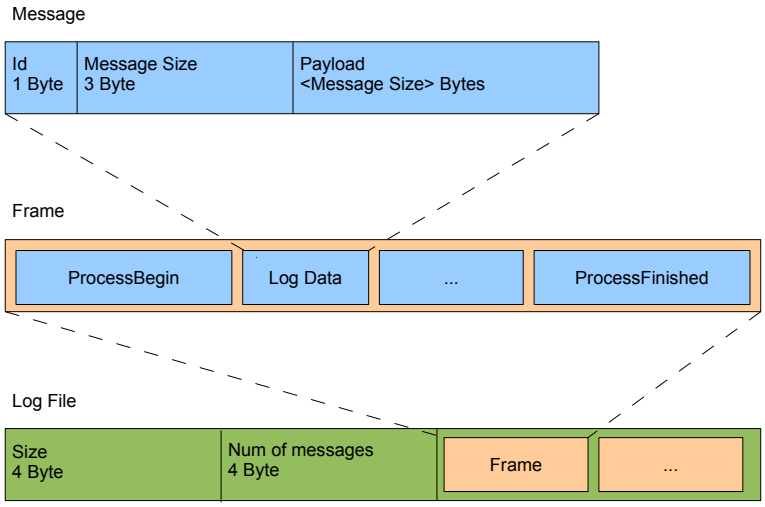


Figure 3.2: This graphic is borrowed from [29] and displays the regular log file format.

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). This kind of chunk is created by remote logging.

logFileCompressed: This chunk contains a sequence of compressed `MessageQueues`. Each queue contains a single second worth of log data and is compressed using Google's snappy [6] compression (cf. Fig. 3.3). Each compressed `MessageQueues` is prefixed by its compressed size. This kind of chunk is created by online logging.

The first two chunks are optional. Each log file must contain one of the latter two chunks, which is also the last chunk in the file.

3.7.5 Replaying Log Files

Log files are replayed using the simulator. Special modules, the CognitionLogDataProvider and the MotionLogDataProvider automatically provide 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. However, it is currently only possible to replay a single log file for a single instance of the B-Human code. For online logging, this means that i. e. either a *Cognition* log or a *Motion* log can be replayed. However, it is possible to merge the log file replayed with its counterpart from the other process using the console command *log merge* (cf. Sect. 10.1.6.3). If the name of a log file follows the naming convention used by the online logger, the head name, body name,

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

scenario, and location will be set to the ones given in the log file name before the log file is replayed. Thereby, modules will read the matching versions of their configuration files.

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. However, the most convenient way to control log file playback is the *log player view* (cf. Sect. 10.1.4.5). If you want to replay several log files at the same time, simply create a file *Config/Scenes/Includes/replayCognition.con* or *Config/Scenes/Includes/replayMotion.con* and add several *sl* statements (cf. Sect. 10.1.6.1) to it. The data of each log file will be fed into a separate instance of the B-Human code.

3.7.6 Annotations

To further enhance the usage of log files, we added the possibility for our modules to annotate individual frames of a log file with important information. This is, for example, information about a change of game state, the execution of a kick, or other information that may help us to debug our code. Thereby, when replaying the log file, we may consult a list of those annotations to see whether specific events actually did happen during the game. In addition, if an annotation was recorded, we are able to directly jump to the corresponding frame of the log file to review the emergence and effects of the event without having to search through the whole log file.

This feature is accessed via the `ANNOTATION`-Macro. An example is given below:

```
#include "Tools/Debugging/Annotation.h"
...
ANNOTATION("GroundContactDetector", "Lost GroundContact");
```

It is advised to be careful to not send an annotation in each frame because this will clutter the log file. When using annotations inside of a behavior option the output option `Annotation` should be used to make sure annotations are not sent multiple times.

The annotations recorded can be displayed while replaying the log file in the annotations view (cf. Sect. 10.1.4.5).

3.7.7 Thumbnail Images

Logging raw YUV422 images as the cameras capture them seems impossible with the current hardware. One second would consume about 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 frame, 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.

For compressing the images in a way that fulfills the criteria mentioned above, two separate methods have been implemented. One of these generates grayscaled images by taking the luminance channel (Y) of the camera image and averaging blocks of adjacent pixels. The other method compresses the YUV camera image by first eliminating every second row of pixels and then averaging every channel of adjacent blocks of pixels. Afterwards the size of each pixel is reduced to two bytes by eliminating one of the two Y channels in each pixel and using only six bits for the remaining luminance channel and five bits for each of the two chroma channels. In both of the two possible methods, averaging adjacent pixels is done using SSE instructions to speed up the computation.

The total compression rate as well as the usefulness of the resulting images depends on the `downscales` parameter of the ThumbnailProvider, which determines the size of the blocks of pixels being averaged as $2^{\texttt{downscales}} \times 2^{\texttt{downscales}}$ pixels. For a `downscales` parameter of 2, the resulting grayscale thumbnail images are 32 times smaller and the resulting colored images are 128 times smaller than the original camera image. Although a lot of detail is lost in the thumbnail images, it is still possible to see and assess the situations the robot was in (cf. Fig. 3.4).

3.7.8 Image Patches

While logging of thumbnail images allows analysis of the robot's behavior during the game, their compression restricts any use for later image processing. Therefore, image patches were implemented which enable logging of parts of the image. An image patch is defined by an offset into the current camera image as well as its width and height. When logging image patches, only the parts of the image belonging to a patch are written into the log file. This makes it possible to log parts of every image that are important for certain perception modules. An example of this would be the perception of the ball: as described in Sect. 4.2, only areas around ball spots obtained from analyzing vertical color-classified scanlines are considered for further steps of the ball perception. Thus, if functionality of the ball perceptor is to be evaluated and adjusted based on log files, it is sufficient to log image patches containing areas around the ball spots. An example of this can be seen in figure 3.4e.

In order to assure that the perception components work normally with images reconstructed from logged image patches, all areas not covered by an image patch are filled with a dark green color resembling the field color.

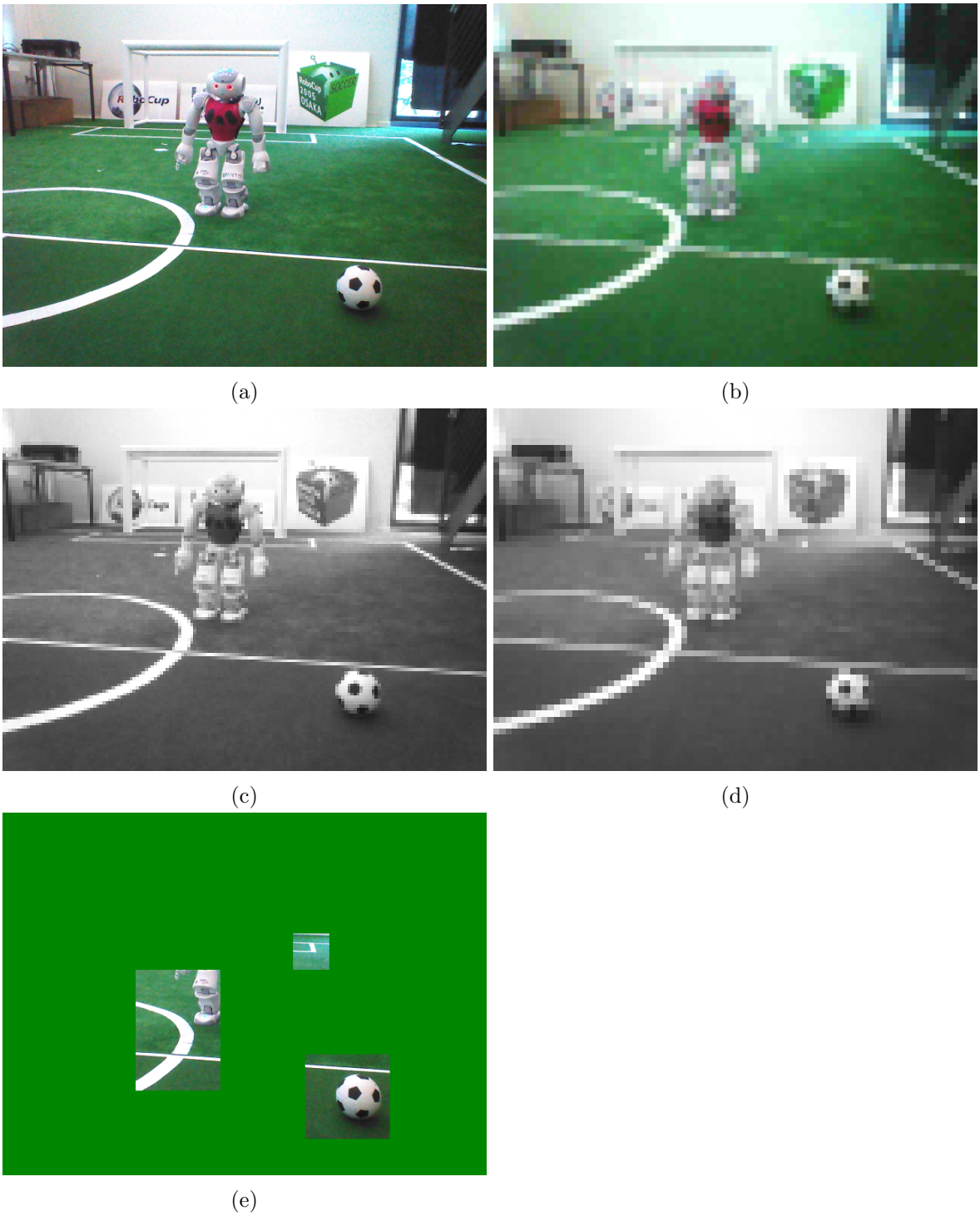


Figure 3.4: (a) Image from the upper camera of the NAO and (b) the corresponding colored thumbnail with a downscale factor of 2 as well as the corresponding grayscale thumbnails with downscale factors (c) 2 and (d) 3. (e) shows the image patches containing only areas of the image in which a potential ball candidate was seen.

Chapter 4

Perception

The perception modules run in the context of the process *Cognition*. They detect features in the image that was just taken by the camera. The modules can be separated into four categories. The modules of the perception infrastructure provide representations that deal with the perspective of the image taken, provide the image in different formats, and provide representations that limit the area interesting for further image processing steps. Based on these representations, modules detect features useful for self-localization, the ball, and obstacles. All information provided by the perception modules is relative to the robot's position.

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

4.1 Perception Infrastructure

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. 4 cm 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, as 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 60 Hz.

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 of this fact and run the lower camera with half the resolution of the upper camera, thereby saving a lot of computation time.

Both cameras deliver their images in the *YUV422* format. The upper camera provides 640×480 pixels while the lower camera only provides 320×240 pixels. As the perception of features in the images relies either on color classes (e. g. for region building) or the luminance values of the image pixels (e. g. for computing edges in the image), the *YUV422* images are converted to the "extracted and color-classified" *ECImage*. The *ECImage* consists of two images: the grayscaled image obtained from the Y channel of the camera image and a so-called "colored" image mapping

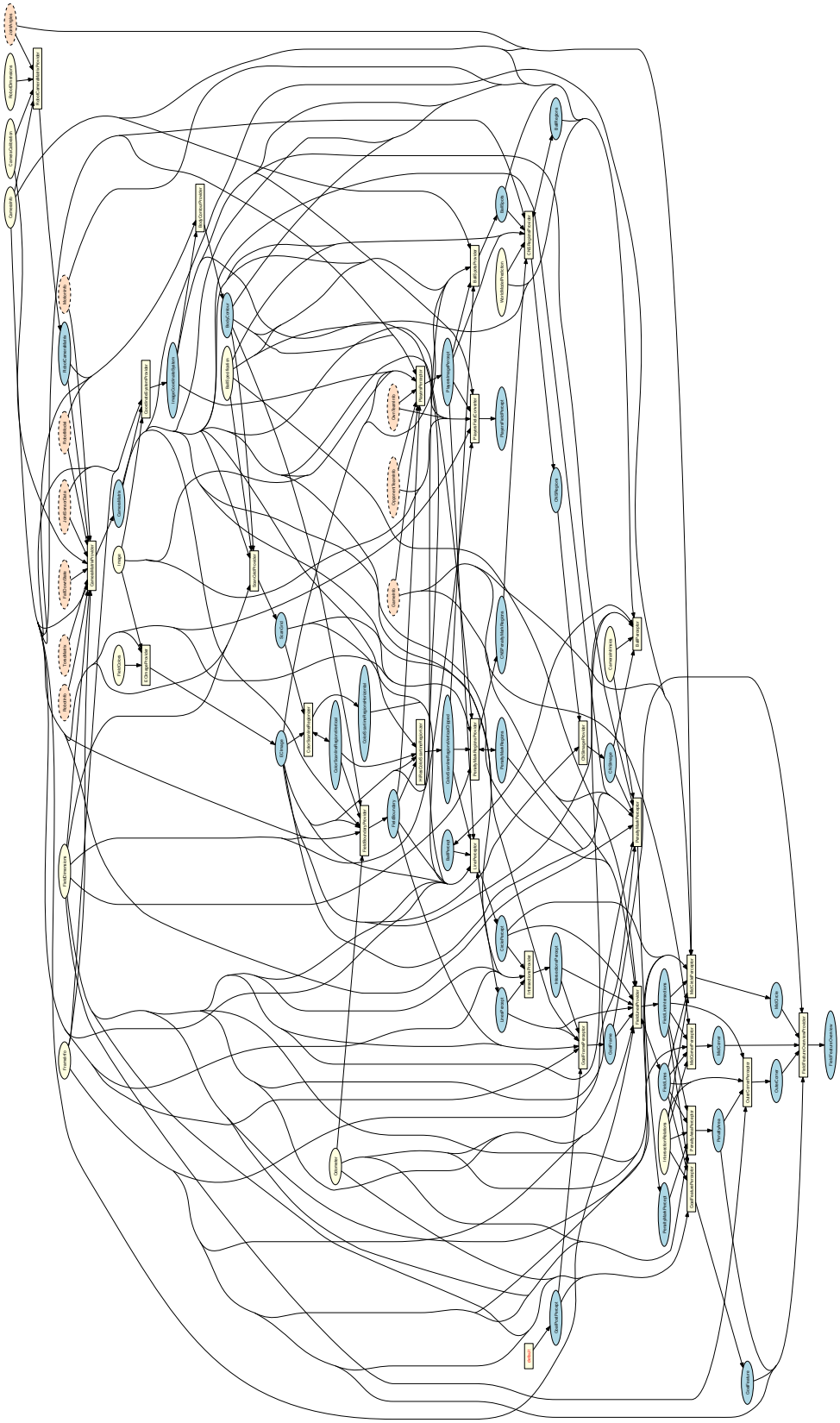


Figure 4.1: Perception module graph. Perception modules are depicted as yellow rectangles. All representations they provide are shown as blue ellipses. The representations they require are shown as ellipses colored in either yellow if they are provided by other *Cognition* modules and orange if they are received from the process *Motion*.

each image pixel to a color class.

Cognition modules processing an image need to know from which camera it comes. 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°, 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 and Camera Calibration

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. 7.4). 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 repair.

In order to overcome this problem, an automatic CameraCalibrator module was introduced. It collects points on the field lines fully autonomously. The points on the lines required are provided by the LineSpotProvider (see Sect. 4.3.1). They are collected for both cameras, after the head moved to predefined angles. Although this calibrator significantly reduces the time needed for a calibration, it has a drawback in terms of precision. The line detection has problems in detecting lines that are further away, in particular when the color calibration is not very good. This can lead to inaccurate values for the estimated tilts of the cameras.

Notable features of the automatic CameraCalibrator are:

- The user can mark arbitrary points on field lines if the automatic detection does not produce enough points. 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

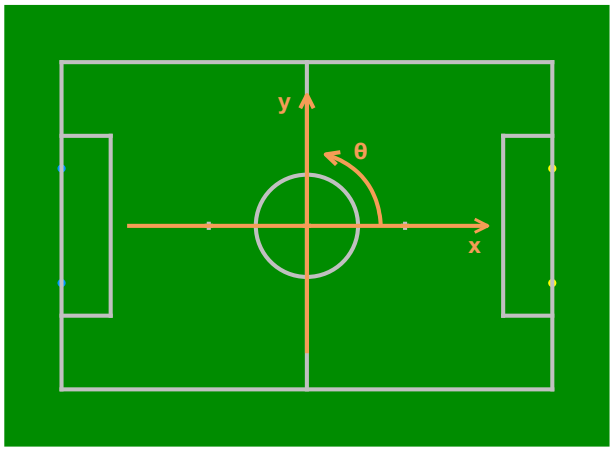


Figure 4.2: Visualization of the global coordinate system (opponent goal is marked in yellow)

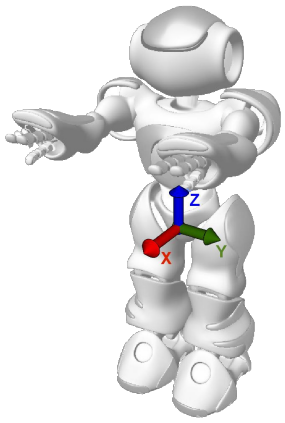


Figure 4.3: Visualization of the robot-relative coordinate system

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.

- 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.

- A manual deletion of samples is possible by left-clicking into the image and on the point the sample has been taken. Likewise, a manual insertion of samples is now possible by *CTRL* + left-clicking into the image at the point you want the sample to be.

- Command generation for correcting the body rotation. In case you don't want the *BodyRotationCorrection* stored in the *CameraCalibration*, you can manually call the JointCalibrator and transfer the values or you can use the command
  set module:AutomaticCameraCalibrator:setJointOffsets true
  before running the optimization. After the optimization, a bunch of commands will be generated and you can enter them in order of appearance to transfer the values into the *JointCalibration*.

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.

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

Figure 4.4: Origin of the *ImageCoordinateSystem*

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 [26].

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 [19].

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

Identifying the color classes of pixels in the image is done by the ECImageProvider when computing the *ECImage*. In order to be able to clearly distinguish different colors and easily define color classes while still being able to compute the *ECImage* for every camera image in real time, the YHS2 color space is used.

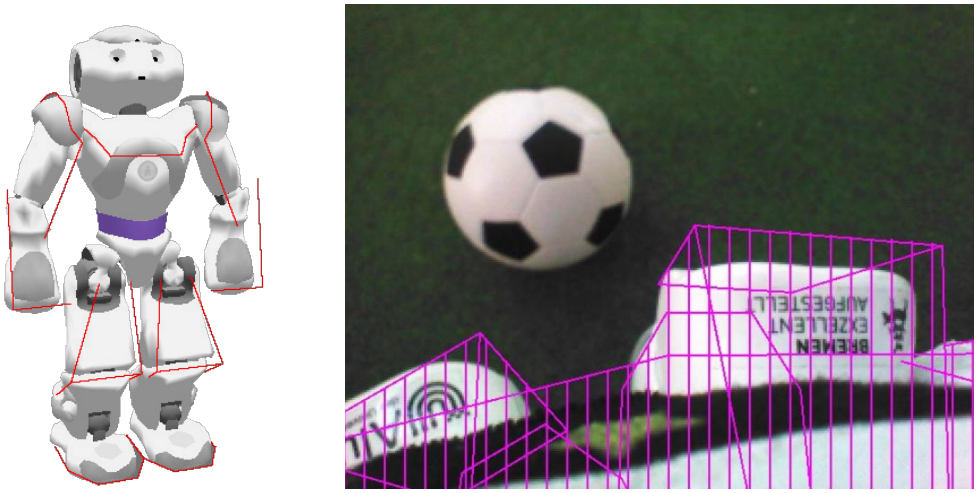


Figure 4.5: Body contour in 3-D (left) and projected to the camera image (right).

YHS2 Color Space

The YHS2 color space is defined by applying the idea behind the HSV color space, i. e. defining the chroma components as a vector in the RGB color wheel, to the YUV color space. In YHS2, the hue component H describes the angle of the vector of the U and V components of the color in the YUV color space, while the saturation component S describes the length of that vector divided by the luminance of the corresponding pixel. The luminance component Y is just the same as it is in YUV. By dividing the saturation by the luminance, the resulting saturation value describes the actual saturation of the color more accurately, making it more useful for separating black and white from actual colors. This is because in YUV, the chroma components are somewhat dependent of the luminance (cf. Fig. 4.6).

Classification Method

Classifying a pixel's color is done by first applying a threshold to the saturation channel. If it is below the given threshold, the pixel is considered to describe a non-color, i. e. black or white. In this case, whether the color is black or white is determined by applying another threshold to the luminance channel. However, if the saturation of the given pixel is above the saturation threshold, the pixel is of a certain color, if its hue value lies within the hue range defined for that color.

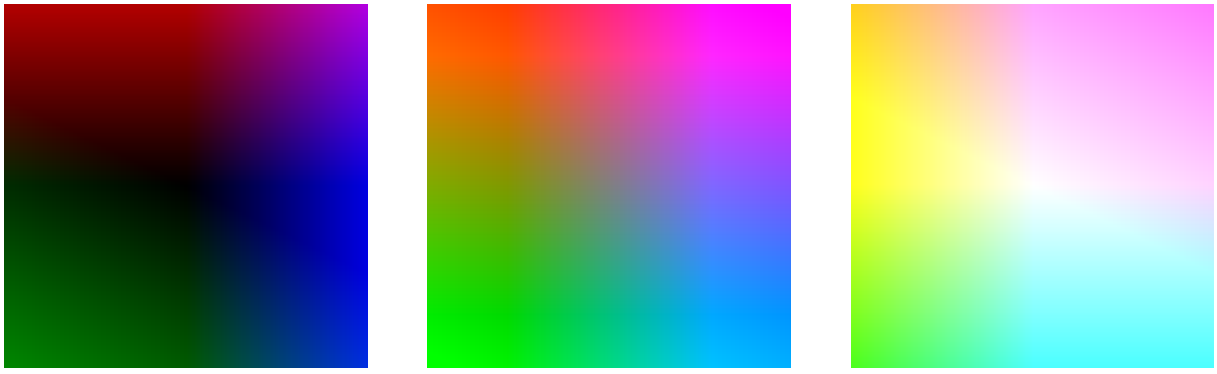
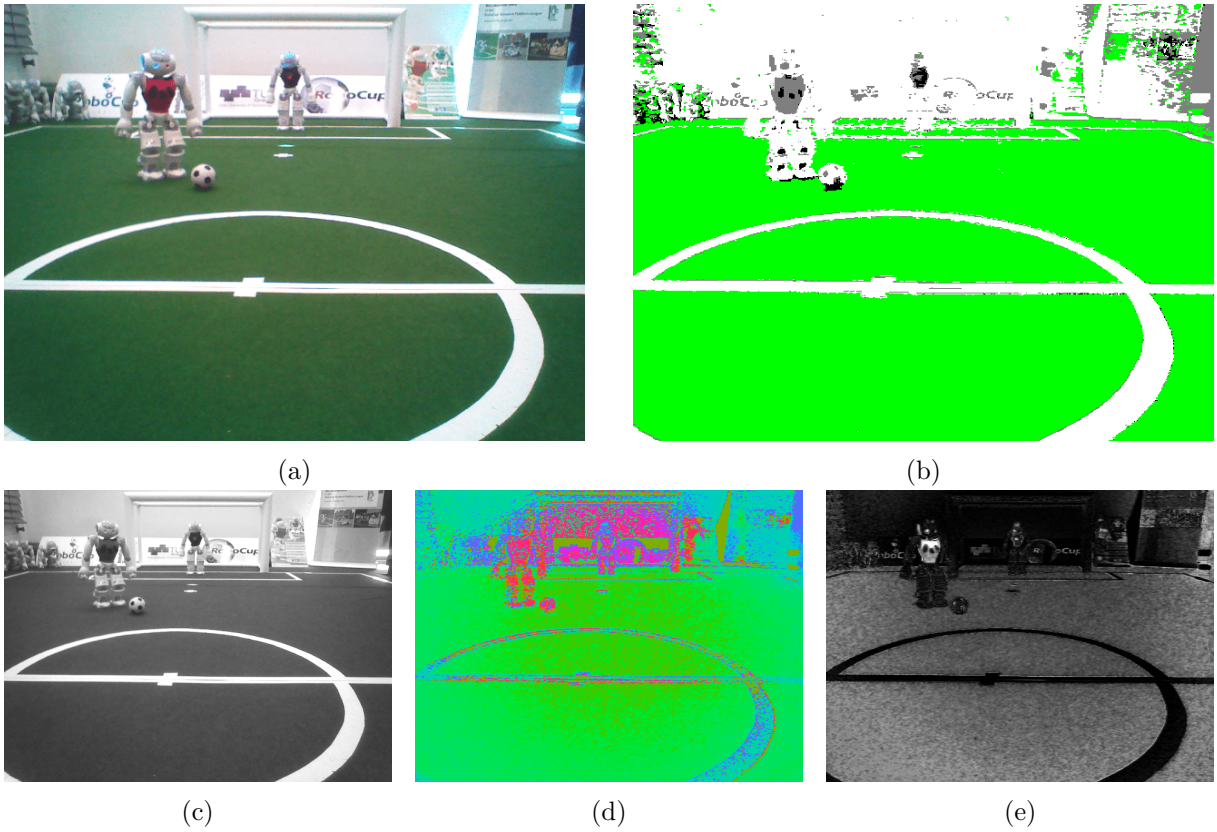


Figure 4.6: The UV plane of the YUV color space for $Y = 0$, $Y = 128$ and $Y = 255$.



(a) (b)

(c) (d) (e)

Figure 4.7: (a) An image from the upper camera of a NAO with (b) a corresponding color classification based on its (c) luminance, (d) hue and (e) saturation channels in the YHS2 color space.

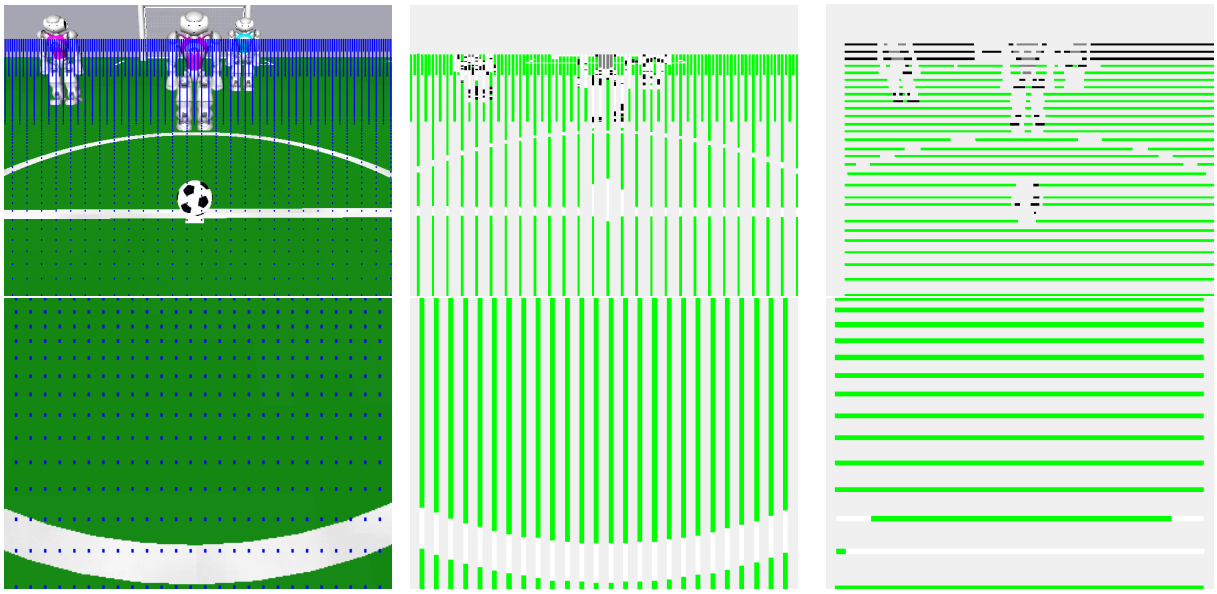
In order to classify the whole camera image in real time, both the color conversion to YHS2 and the color classification are done using SSE instructions.

Fig. 4.7 shows representations of an image from the upper camera in the YHS2 color space and a classification based on it for the colors white, black, green and "none" (displayed gray).

4.1.5 Segmentation and Region-Building

The ColorScanlineRegionizer scans along vertical and horizontal scanlines whose distances from each other are small enough to detect the field boundary, field lines, and the ball. These scanlines are segmented into regions, i. e. line segments of a similar color based on the `colored` part of the *ECImage*. In vertical direction, the sampling frequency is not constant. Instead, it is given by the *ScanGrid*, the steps of which depend on the vertical orientation of the camera and basically correspond to a distance of a bit less than the expected width of a horizontal field line at that position in the image. Whenever the color classes of two successive scan points on a vertical line differ, the region in between is scanned to find the actual position of the edge between the two neighboring regions. The best position is determined to be one pixel above the position where enough pixels were found to be classified as color classes different from the one of the previous region. Fig. 4.8 visualizes the subsampling.

In order to save computational time, the *ScanGrid* starts at the horizon as given by the *ImageCoordinateSystem*. Additionally, the vertical scanlines are divided into "low resolution" and "high resolution" scanlines. At first, only the low resolution scanlines are computed and used



(a) (b) (c)

Figure 4.8: (a) Visualization of the *ScanGrid*: the blue dots are checked for building regions of colors. (b) and (c) show the high resolution vertical and the horizontal scanline regions, respectively.

for determining the field boundary. Afterwards, the high resolution scanlines are computed for further processing, starting at the field boundary.

4.1.6 Detecting The Field Boundary

The 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. For this purpose, our current approach searches for vertical scanlines, 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 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 scanline 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 scanline. 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. Thus, 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 4.9 shows that most of the spots are placed on the actual boundary using this method.

Since the detection of the field lines (cf. Sect. 4.3.1) should only take place inside the area of the field, the *ColorScanlineRegionsVertical* are clipped by the *FieldBoundary* resulting in a rep-

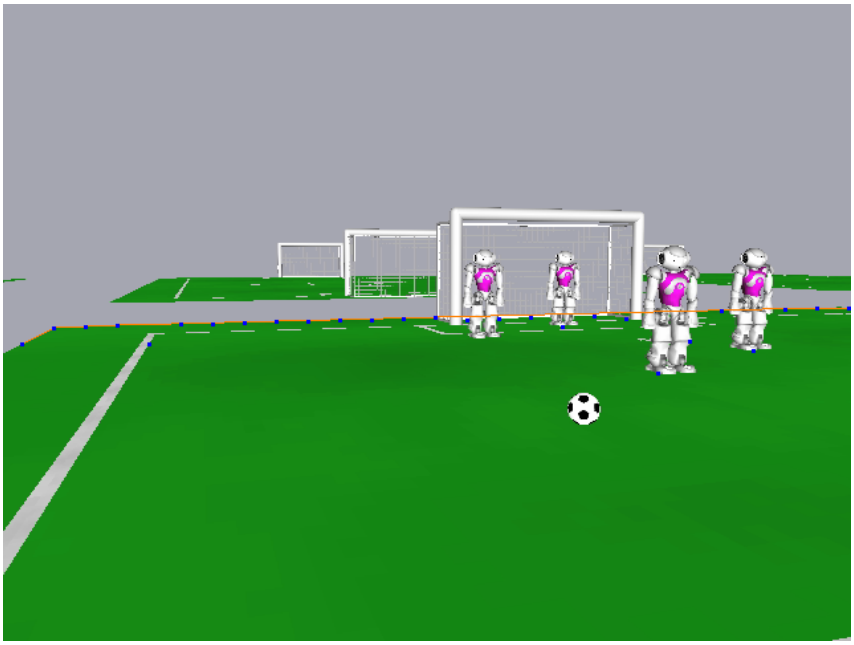


Figure 4.9: Boundary spots (blue) and the estimated field boundary (orange) in a simulated environment with multiple fields.

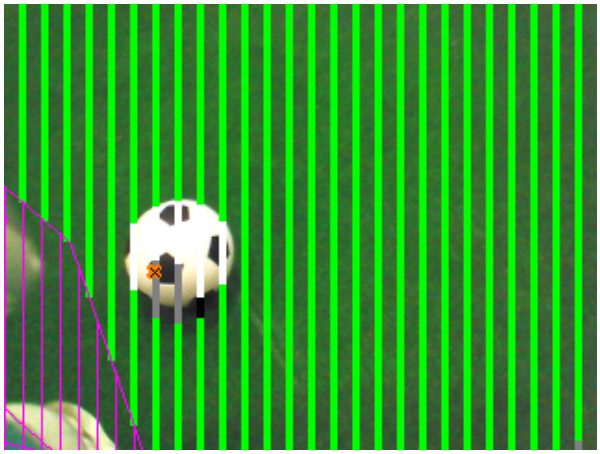
resentation called *ColorScanlineRegionsVerticalClipped* that is used for further processing. Since clipping away the area outside the field also means to remove the regions that are potentially cluttered the most, ignoring them in further processing also benefits the computation time.

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 [18, 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. 4.9.

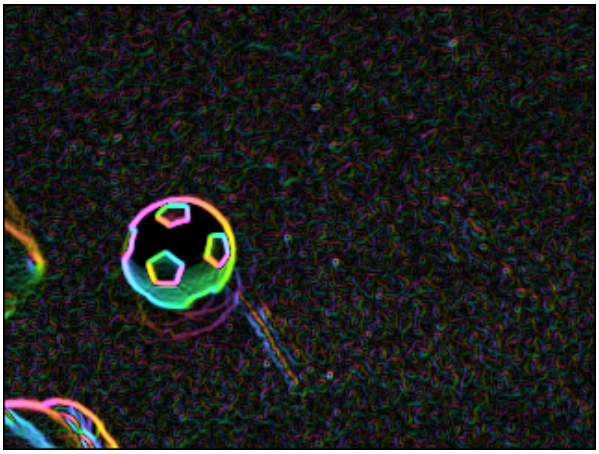
The lower camera does not see the actual field boundary most of the time. So calculating the boundary in every image would lead to oscillating representations. Thus, the field boundary is calculated on the upper image. Nevertheless, a convex hull is calculated on the lower image too, since the boundary 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.

4.2 Detecting the Black and White Ball

The introduction of the black and white ball is the major new challenge in the Standard Platform League in 2016. Until the RoboCup 2015, the ball was orange and rather easy to detect. In particular, it was the only orange object on the field. The new ball is mainly white with a regular pattern of black patches, just as a miniature version of a regular soccer ball. The main problem is that the field lines, the goals, and the NAO robots are also white. The latter even



(a) Vertical scan lines and a detected ball candidate (the cross). Parts of the robot's body are ignored (bottom left).



(b) Contrast-normalized Sobel image. The colors indicate the directions of the gradients.

(c) Visualization of the search space for the ball contour. The actual search is only performed around the ball candidate, but in single pixel steps in both dimensions.

(d) The contour with the highest response (green) and the sample grid to check the ball pattern (pixels classified as black are shown in red, white pixels in blue).

Figure 4.10: The main steps of the ball detection

have several round plastic parts and they also contain grey parts. Since the ball is often in the vicinity of the NAOs during a game, it is quite challenging to avoid a large number of false positives.

We use a multi-step approach for the detection of the ball. First, the vertical scan lines our vision system is mainly based on are searched for ball candidates. Then, a contour detector fits ball contours around the candidates' locations. Afterwards, fitted ball candidates are filtered using some general heuristics. Finally, the surface pattern inside each remaining candidate is checked.

4.2.1 Searching for Ball Candidates

Our vision system scans the image vertically using scan lines of different density based on the size that objects, in particular the ball, would have at a certain position inside the image. To determine ball candidates, the BallSpotProvider searches these scan lines for sufficiently large

gaps in the green that also have a sufficiently large horizontal extension and contain enough white (cf. Fig. 4.10a). Only the first and second resolutions of the scanlines are used. From a specific distance, the ball is that small inside the image that the calculation can not provide good results. For this reason, ball spots more far away than this distance will be generated on white blobs that paled of green. In addition, all initial spots are ignored if they are nearby a previous one[2] or are clearly inside a detected robot. In this context, "clearly inside a robot" means all parts except for feet and arms, as these elements cannot be detected accurately.

4.2.2 Fitting Ball Contours

As the position of a ball candidate is not necessarily in the center of an actual ball, the area around such a position is searched for the contour of the ball as it would appear in this part of the image given the intrinsic parameters of the camera and its pose relative to the field plane. The approach is very similar to the detection of objects in 3-D space using a stereo camera system as described by Müller et al. [15], but we only use a single image instead. For each ball candidate, the CNSRegionsProvider computes a rectangle that surrounds the area a ball could occupy in that part of the image. The CNSImageProvider then first joins neighboring areas to bigger rectangles and computes a contrast-normalized Sobel image for each area (cf. Fig. 4.10b). Technically, only a single Sobel image is filled, in which only the areas surrounding the candidate positions are updated. The BallPerceptor then performs the actual ball detection. It searches this contrast image in each of the areas provided by the CNSRegionsProvider for the best match with the expected ball contour (cf. Fig. 4.10c). The best match is then refined by adapting its hypothetical 3-D coordinates (cf. Fig. 4.10d).

4.2.3 Filtering Ball Candidates

The fitting process results in a measure, the *response*, for how well the image matches with the contour expected at the candidate's location. If this value is below a threshold, the ball candidate is dropped. The threshold is dynamically determined from the amount of green that surrounds the ball candidate. On the one hand, the less green is around the candidate, the higher the response has to be to reduce the amount of false positives inside robots. However, if a ball candidate is completely surrounded by green pixels and the response was high enough to exclude the possibility of being a penalty mark, the ball candidate is accepted right away, skipping the final step described below that might be failing if the ball is rolling quickly. All candidates that fit well enough are processed in descending order of their response. As a result, the candidate with the highest response that also passes all other checks will be accepted. These other checks include that the ball radius found must be similar to the radius that would be expected at that position inside the image.

4.2.4 Checking the Surface Pattern

For checking the black and white surface pattern, a fixed set of 3-D points on the surface of the ball candidate are projected into the image (cf. Fig. 4.10d). For each of these pixels, the brightness of the image at its location is determined. Since the ball usually shows a strong gradient in the image from its bright top to a much darker bottom half, the pixels are artificially brightened depending on their position inside the ball. Then, Otsu's method [17] is used to determine the optimal threshold between the black and the white parts of the ball for the pixels

[2]Despite this step is unnecessary for the following used ball perceptor, it saves a bit calculation time inside the ball spot provider itself, but even more inside different ball percept provider approaches.

sampled. If the average brightnesses of both classes are sufficiently different, all pixels sampled are classified as being either black or white. Then, this pattern is looked up in a pre-computed table to determine whether it is a valid combination for the official ball. The table was computed from a 2-D texture of the ball surface considering all possible rotations of the ball around all three axes and some variations close the transitions between the black and the white parts of the ball.

The approach allows our robots to detect the ball in distances of up to five meters with only a few false positive detections. It works better under good lighting conditions. In a rather dark environment, as at the site of RoboCup 2017, balls with lower responses had to be accepted in order to detect the ball at all. This resulted in more false positive detections, in particular in the feet of other robots, because they are also largely surrounded by green.

4.3 Localization Features

To provide input for the self-localization (cf. 5.1), there exist multiple approaches to extract information from the visual sensors. Every (visual) knowledge relevant for position estimation that our robot is aware of is based on the field line detection. Although the penalty mark and the center circle are basically lines as well, they are – at least partly – detected separately to ensure more robustness for each single detection. Together, the representations of all three basic detections make up the first layer in the process chain of the visual perception. Each layer combines previously known information to provide more specific features. Information of every layer can be used by the self-localization. The main layers after the basic feature detection are the intersections perception and the following field feature perception.

4.3.1 Detecting Lines

The perception of field lines by the LinePerceptor relies mostly on the scanline regions. In order to find horizontal lines in the image, adjacent white vertical regions that are not within a perceived obstacle (cf. Sect. 4.4) are combined to line segments. Correspondingly, vertical line segments are constructed from white horizontal regions. These line segments and the center points of their regions, called *line spots*, are then projected onto the field. Using linear regression of the line spots, the line segments are then merged together and extended to larger line percepts. During this step, line segments are only merged together if at least a given ratio of the resulting line consists of white pixels in the image. Fig. 4.11 shows the process of finding lines in the camera image.

4.3.2 Detecting the Center Circle

Besides providing the *LinesPercept* containing perceived field lines, the LinePerceptor also detects the center circle in an image if it is present. In order to do so, when combining line spots to line segments, their field coordinates are also fitted to circle candidates. After the *LinesPercept* was computed, spots on the circle candidates are then projected back into the image and adjusted so they lie in the middle of white regions in the image. These adjusted spots are then again projected onto the field and it is once again tried to fit a circle through them, excluding outliers.

If searching for the center circle using this approach did not yield any results, another method of finding the center circle is applied. We take all previously detected lines whose spots describe an arc and accumulate the center points of said arcs to cluster them together. If one of these

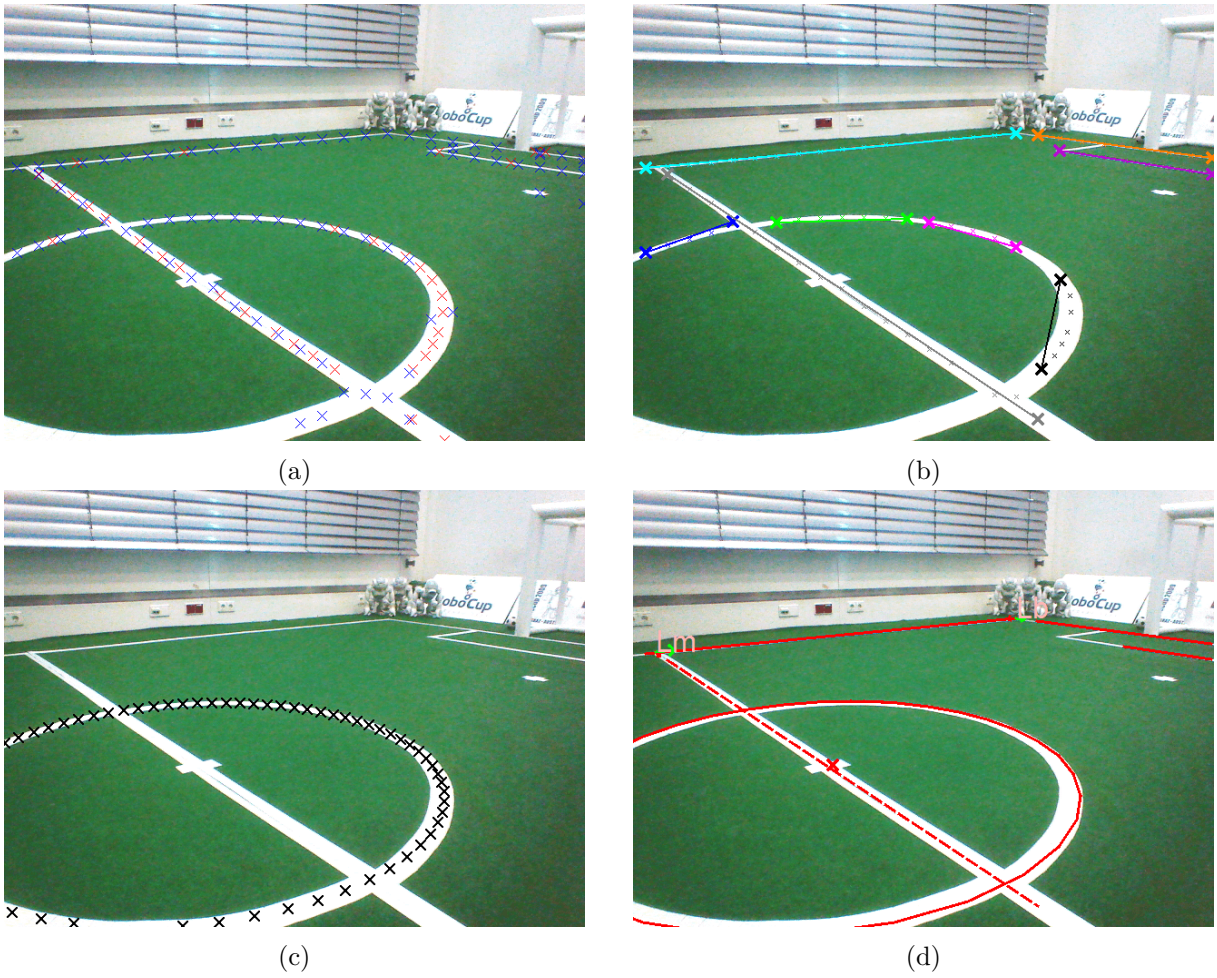


Figure 4.11: (a) Line spots constructed from the scanline regions. Spots from horizontal regions are marked red, spots from vertical regions blue. (b) The detected line segments in the image. (c) Points on the circle candidate that were projected back into the image for checking if the corresponding pixels are white. (d) Filtered line and circle percepts in the *FieldLines* representation.

clusters contains a sufficient number of center points, the average of them is considered to be the center of the center circle.

If a potential center circle was found by any of these two methods, it is accepted as a valid center circle only if – after projecting spots on the circle back into the image – at least a certain ratio of the corresponding pixels is white (cf. Fig. 4.11c).

4.3.3 Line Coincidence Detection

To find points where two lines coincide, we are calculating the intersections of the previously perceived lines (cf. 4.3.1). This approach is probably much faster and more accurate[3] than any separate (visual) perception such as e. g. corner search, but it means that detecting a line intersection without a line (percept) is impossible. In particular this disadvantage applies to line corners of a penalty area that extend into the image.

[3] Assuming a sufficiently accurate line detection.

The IntersectionProvider fills the *IntersectionPercept* based on the *LinesPercept*. Each known line will be compared to all other known lines, excluding those that are known to be part of the center circle. Because of the way the field is set up (cf. the current rules [2]), an intersection is only possible if the inclination of two compared lines in field coordinates is roughly $\frac{\pi}{2}$. In that case, the point of intersection of the two lines is calculated. In order to be valid, this point must lie in between both detected line segments. Next, we also want to specify the type of each intersection. The types of intersections are congruent with their names: `L`, `T`, and `X`. This means that intersections lying clearly in-between both lines are of type `X` while those that are clearly inside one line but roughly at one end of the other are considered to be of type `T`. If the point of intersection lies roughly at the ends of both lines, it is assumed to belong to an `L`-type intersection.

In case that parts of both lines are seen, but the point of intersection does not lie on either line – e. g. because an obstacle is standing in front of the intersection point – the lines will be virtually stretched to some extend so that the point of intersection lies roughly at their ends. The possible distance by which a line can be extended depends on the length of its recognized part.

The *IntersectionPercept* stores the type of each intersection as well as its position, its base lines, and its rotation.

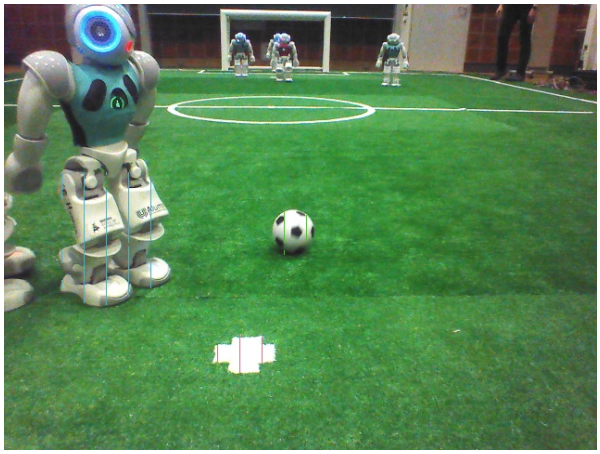
4.3.4 Preprocessed Lines and Intersections

Some features of the *LinesPercept* and *IntersectionPercept* – such as the spots from which the lines were fitted and image coordinates – are not needed for further processing. On the other hand, now that the positions of all lines and intersections are known, additional information can be gained by making assumptions based on the known layout of the field lines. Therefore, as a next step, the FieldLinesProvider provides adapted representations of the *LinesPercept* and *IntersectionPercept* for further use.

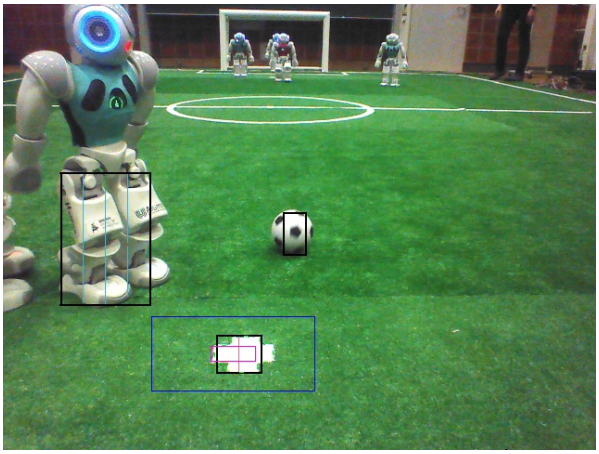
The representation *FieldLines* contains the processed line percepts (cf. Fig. 4.11d). For this, every line that is assumed to be seen on the center circle or inside the goal frame or net (cf. 4.3.6) was sorted out. Lines that extend into the goal frame are clipped to its boundaries. In addition to that, every line that is as long as a ball would appear at the corresponding position in the image is removed[4]. Apart from removing invalid percepts, lines in *FieldLines* also have additional properties. A line is marked as `mid`, if it goes through the center circle, which means that it is probably the center line. The `long` attribute is given to any line that is longer than a specific on-field distance. A long line means the detected line probably belongs to the outer lines, center line or penalty front lines. For this reason, it is advisable to check on a distance that is longer than the penalty area sidelines or a part line inside the center circle, but as short as possible to receive the additional information as often as possible.

The representation *FieldLineIntersections* contains the processed line intersection percepts. It consists only of intersections that do not apply to any previously removed lines. If one of the lines belonging to an intersection was cut, the intersection type is corrected accordingly. Additionally, an intersection can now have a type of `mid` or `big`. If both lines of the intersection are of type `mid`, the intersection is also considered to be `mid`. Analogous to that, an intersection consisting of two `long` lines has the attribute `big`. After all this processing, `L`-type intersections are filtered out, if the angle between their two lines is not about ninety degrees.

[4]This is a competition quick hack to avoid further calculations on false-positive lines that are fitted inside the ball.



(a) The vertical scanline regions are filtered for those that were not classified as green. They are clipped at a distance of 2 m from the observer (colored lines inside the robot, the ball, and the penalty mark). They are joined together as grouped regions (each group has a different color).



(b) Bounding boxes are computed for each group (black). For groups with the expected size and minimum white ratio, a search region for the center (red) and a region for the CNS image (blue) is computed. The contour points of the penalty mark are then fitted inside these regions (bright green).

Figure 4.12: Steps of the penalty mark detection

4.3.5 Penalty Mark Perception

The penalty mark is one of the most prominent features on the field, because it only exists twice. In addition, it is located in front of the penalty area and can thereby be easily seen by the goal keeper. The main goal of the development of a new penalty mark detector was to eliminate false positive detections to make the penalty mark a reliable feature for self-localization. This was achieved by limiting the detection distance to 2 m and by introducing a contrast-based shape check. As a result, e. g., the B-Human goal keeper detected the penalty mark 12597 times during the 2017 SPL final and none of these detections was a false positive.

4.3.5.1 Approach

Similar to the ball detection (cf. Sect. 4.2), for the penalty mark detection, it is first searched for a number of candidates and afterwards, these are checked for being the actual penalty mark. However, while it is very important to detect the ball as often as possible, it is sufficient for the self-localization to detect the penalty mark less frequently. Therefore, only a maximum of three candidates is checked per image to save computation time. Also similar to the ball detection, the search for candidates and the actual check are separated into two different modules. The check needs a contrast normalized Sobel (CNS) image (cf. Fig. 4.10b) for the candidate region, and since the computation of these regions costs time, all the regions requested by different modules are grouped together to avoid computing overlapping regions twice. Therefore, the search for candidates has to performed before the CNS image is computed and the shape check has to be executed afterwards.

4.3.5.2 Finding Candidate Regions

The PenaltyMarkRegionsProvider first collects all regions of the scanned low resolution grid that were not classified as green, i. e. white, black, and unclassified regions. In addition, the regions

are limited to a distance of 2 m from the robot, assuming they represent features on the field plane. Vertically neighboring regions are merged (cf. Fig. 4.12a) and the amount of pixels that were actually classified as white in each merged region is collected. Regions at the lower end and the upper end of the scanned area are marked as invalid, i. e. there should be a field colored region below and above each valid region. The regions are horizontally grouped using the *Union Find* approach. To achieve a better connectedness of, e. g., diagonal lines, each region is virtually extended by the expected height of three field line widths when checking for the neighborhood between regions. For each group, the bounding box (cf. Fig. 4.12b) and the ratio between white and non-green pixels is determined. For all groups with a size similar to the expected size of a penalty mark that are sufficiently white and that do not contain invalid regions, a search area for the center of the penalty mark and a search area for the outline of the penalty mark are computed. As the search will take place on 16×16 cells, the dimensions are extended to multiples of 16 pixels.

4.3.5.3 Computing CNS Regions

The CNSRegionsProvider integrates the outline regions into the set of regions for which the CNS image has to be computed. The CNSImageProvider will then perform the computation.

4.3.5.4 Checking Penalty Mark Candidates

The PenaltyMarkPerceptor goes through the list of possible center regions for the penalty mark and determines the best match of the contour of the penalty area in each of these regions. It searches in one-pixel steps and checks for four different orientations, i. e. in 22.5° steps. However, as the contour detector also iteratively refines each match, the orientation can also be matched arbitrarily precise. If the match is sufficiently good, it will be checked more thoroughly, whether the candidate is surrounded by green. If the candidate is surrounded by at least 90% green, it is returned as being a valid penalty mark.

4.3.6 Field Features

The self-localization has always used field lines, their crossings, and the center circle as measurements. Since 2015, these features are complemented by the perception of the penalty marks. All of these field elements are distributed over the whole field and can be detected very reliably, providing a constant input of measurements in most situations.

The *Field Features* – which were introduced in 2016 – are created by combining multiple basic field elements in a way that a robot pose (in global field coordinates) can be derived directly. The handling of the field symmetry, which leads to actually two poses, is described in Sect. 5.1.2.

Overall, this approach provides a much higher number of reliable pose estimates than the previous goal-based approach, as the field lines on which it is based can be seen from many perspectives and have a more robust context (straight white lines surrounded by green carpet) than the noisy unknown background of goal posts.

Every *FieldFeature* is a `Pose2f`, which is the pose of this in relative field coordinates. In addition, it has the following attributes and methods:

`isValid` is an attribute that describes, if the pose is valid or rather if the feature was detected in this frame.

`markedPoints`, `markedLines` **and** `markedIntersections` are attributes that associate several percepts with explicitly defined global field elements such as the center line.

`isLikeEnoughACorrectPerception(...)` calculates if more lines are unmarked inside a given space around this feature than allowed.

`getGlobalFeaturePosition()` calculates the two global robot poses that the robot can have according to this feature (if it is valid).

`clear()` clears the lists `markedPoints`, `markedLines` and `markedIntersections`. The intended use is inside a provider at the start of an `update` method.

`draw()` draws the result of `getGlobalFeaturePosition`() into a world state drawing context.

Normally, all feature detections are using the preprocessed field elements (cf. 4.3.4). We are currently computing the following field features:

Penalty Area The PenaltyAreaPerceptor, which provides *PenaltyArea*, tries to find a penalty area by the help of the penalty mark and a line or two intersections. If a penalty mark is known, it searches a line that matches the corresponding penalty area front line. The penalty area can also be found by combining two `small` (not `big`) `T`-intersections or one `small` `T`-intersection and one `small` `L`-intersection.

Outer Corner The *OuterCorner* refers to the areas where ground lines meet side lines, which means that the outer corner is on the right or left side. To find such a corner, the OuterCornerPerceptor combines a `big` `L`-intersection with an element of the penalty area.

Mid Circle The MidCircleProvider is combining a detected center circle with a detected `mid` line to provide the *MidCircle*.

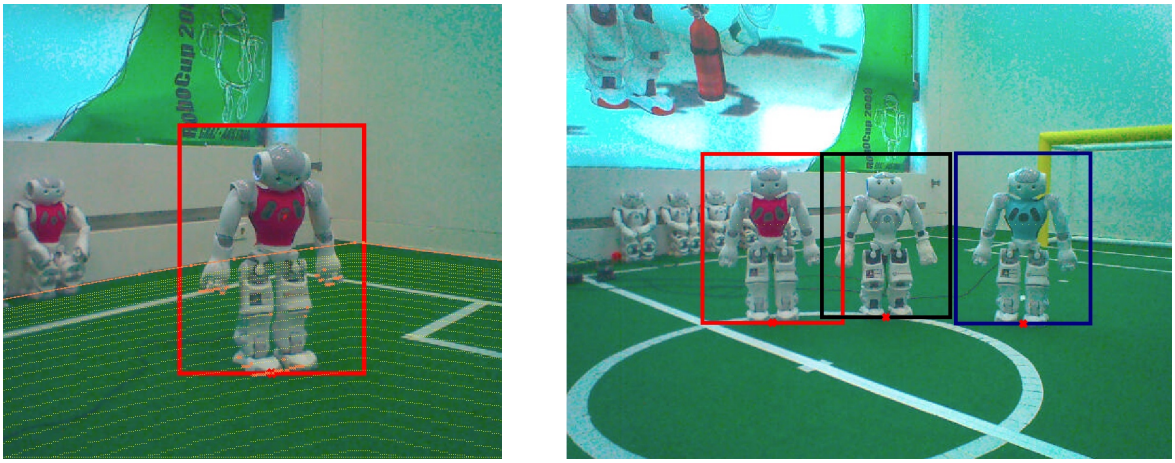
Mid Corner A *MidCorner*, which is provided by the *MidCornerPerceptor*, is simply derived from a `big` `T`-intersection.

Goal Frame Besides the obvious reason to have another field feature to get more information to put into the self-locator, the *GoalFrame* is also used to sort out false-positive lines that were detected inside the goal net. Thus, the GoalFrameDetector has to calculate directly based on the line percepts. In general, the module searches for points around the goal and validates the rotation with a special use of the *FieldBoundary*.

4.4 Detecting Other Robots and Obstacles

The PlayersPerceptor recognizes standing and fallen robots up to seven meters away in less than two milliseconds of computation time.

Initially, the approach searches for robot parts within the field boundary (cf. Fig. 4.13a). Starting at the boundary, it scans down in vertical lines, looking only at a subset of the pixels. The distance between these pixels is constantly growing. The space and its growing rate are calculated from the distance of the corresponding points on the field. If there are a couple of non-green pixels scanned in a vertical line, then the lowest of these pixels gets marked as a potential robot base point. These are possible spots at the feet and hands of a robot, and they can be merged to a bounding box. After calculating the surrounding box, the method searches for a jersey within it and tries to determine the jersey color (cf. Fig. 4.13b).



(a) (b)

Figure 4.13: (a) Searching for a robot below the boundary of the field (depicted in orange) at a subset of the pixels (depicted as yellow dots). (b) Jersey colors are recognized on robots.

For jersey association, the module uses the team color information from the GameController. The implementation was updated in 2017. Based on the expected height of a jersey of a standing robot, pixels in a square region are sampled, evaluated, and counted as own or opponent team color pixels. The method of pixel evaluation depends on the combination of team colors:

1. If at least one team color can be represented by the *ECImage* (i. e. black, green, or white), then just the *ECImage* is used for pixel evaluation. For teams with other jersey colors, the *FieldColor* `none` is used for comparison.

2. Otherwise, if one team color is gray, the saturation of each pixel is calculated. If the saturation exceeds a threshold, the pixel must be part of a colored jersey and is counted to the non-gray team. Otherwise, the pixel must be part of a gray jersey.

3. If none of both teams has black, gray, white, or green jerseys, the distance of the hue value is calculated. The pixel is evaluated as the team color of which the predefined hue value is closest to the pixel's hue value.

If enough pixels of a specific color are found and there are more pixels in this specific color compared with the count of pixels of the other color, the robot is classified as member of the team with this specific color.

Chapter 5

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. Since these modules have to update their output for each new image processed, they also run in the process *Cognition*. Together with their dependencies, they are depicted in Fig. 5.1.

5.1 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 [28] as this paradigm has been proven to provide robust and precise results in a variety of robot state estimation tasks. However, due to the high amount of noise and the problems arising from the field's symmetry, the main state estimation module is complemented by additional modules that generate new robot pose alternatives and check, if a robot's current orientation is point-symmetrically flipped.

5.1.1 Probabilistic State Estimation

As in previous years, the pose state estimation is handled by multiple hypotheses that are each modeled as an Unscented Kalman Filter [9]. The hypotheses management and hypotheses resetting (cf. Sect. 5.1.2) is realized as a particle filter [3]. Both approaches are straightforward textbook implementations [28], except for some adaptions to handle certain RoboCup-specific game states, such as the positioning after returning from a penalty. In addition, we only use a very low number of particles (currently 12), as multimodalities do not occur often in RoboCup games.

The current self-localization implementation is the SelfLocator that provides the *RobotPose*. Furthermore, for debugging purposes, the module also provides the loggable representation *SelfLocalizationHypotheses* which contains the states of all currently internally maintained pose hypotheses.

Overall, this combination of probabilistic methods enables a robust, precise and efficient self-localization on the RoboCup field. However, eventually, the result always depends on the quality and quantity of the incoming perceptions.

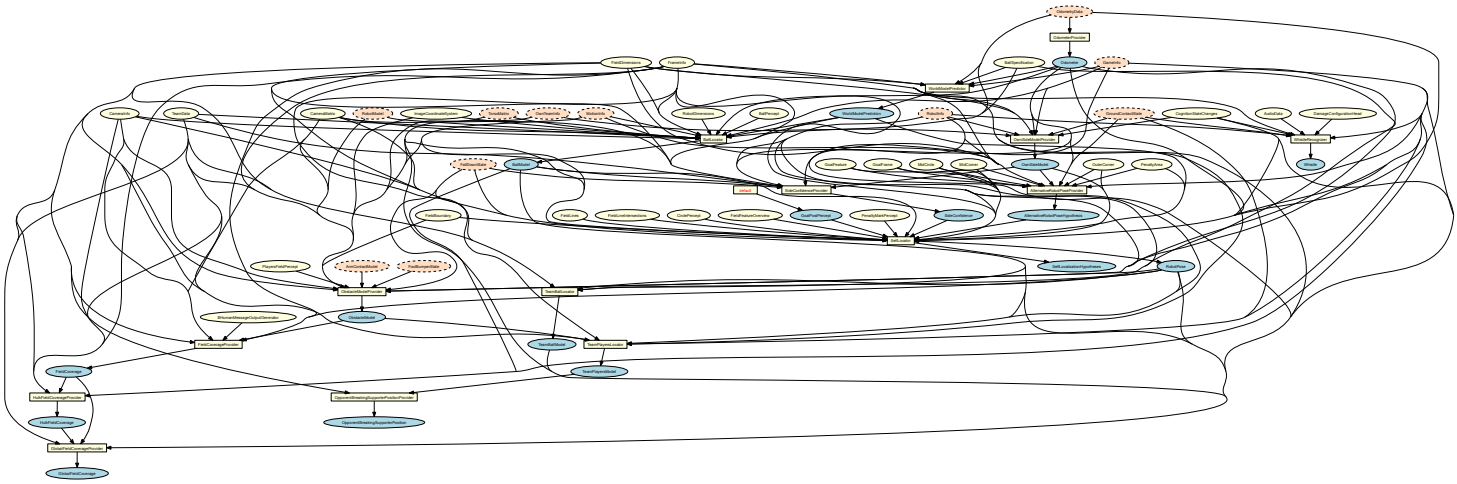


Figure 5.1: Modeling module graph. Modeling modules are depicted as yellow rectangles. All representations they provide are shown as blue ellipses. The representations they require are shown as ellipses colored in either yellow if they are provided by other *Cognition* modules and orange if they are received from the process *Motion*.

5.1.1.1 Perceptions Used for Self-Localization

In the past, B-Human used goals as a dominant feature for self-localization. When the field was smaller and the goal posts were painted yellow, they were easy to perceive from most positions and provided precise and valuable measurements for the pose estimation process. In particular the sensor resetting part (cf. Sect. 5.1.2), i. e. the creation of alternative pose estimates in case of a delocalization, was almost completely based on the goal posts perceived. In 2015, we still relied on this approach, using a detector for the white goals [25]. However, as it turned out that this detector required too much computation time and did not work reliably in some environments (requiring lots of calibration efforts), we decided to perform self-localization without goals but by using the new complex field features (cf. Sect. 4.3.6) instead.

In addition, the self-localization still uses basic features such as field lines, their crossings, and the center circle as measurements. Since 2015, these features are complemented by the perception of the penalty marks (cf. Sect. 4.3.5). All these field elements are distributed over the whole field and can be detected very reliably, providing a constant input of measurements in most situations.

Field features are also used as measurements but not as a perception of relative landmarks. Instead, an artificial measurement of a global pose is generated, reducing the translational error in both dimensions as well as the rotational error at once. Furthermore, in contrast to the basic field elements that are not unique, no data association (cf. Sect. 5.1.1.2) is required. The handling of the field symmetry, which leads to actually two poses, is similar to the one that is done for the sensor resetting as described in Sect. 5.1.2.

5.1.1.2 Resampling based on Particle Validity

One important part of any Monte Carlo localization approach is the *resampling* (cf. [3]), i. e. to determine which particles become copied how often to the new particle set representing the current state's probability distribution. For this purpose, each particle has a weighting that describes how "good" it is; the higher the weighting, the higher the likelihood that the particle will be copied to the new set. There is no general-purpose approach to compute these weights,

each application requires its own way to determine the quality of a particle instead.

In our implementation, each particle's validity is directly used as its weighting (combined with a base weighting, as described below). The validity is our measure for a particle's compatibility to the recent perceptions and is computed during the process of data association. To use a perception in the UKF's measurement step, it needs to be associated to a field element in global field coordinates first. For instance, given a perceived line and a particle's current pose estimate, the field line that is most plausible regarding length, angle, and distance is determined. In a second step, it is checked whether the difference between model and perception is small enough to consider the perception for the measurement step. For all kinds of perceptions, different thresholds exist, depending on the likelihood of false positives and the assumed precision of the perception modules. After the association process has been carried out for all perceptions, the current validity v_c can be computed by setting the number of successfully associated perceptions in relation to the total number of perception in this frame. For this computation, we consider different weights for different kinds of perceptions, i. e. a field feature – which is a very reliable perception – that cannot be associated has a higher impact on the validity than a field line that has not been matched.

To avoid strong oscillations of the validity, which might cause instabilities within the sample set and hence a robot pose that reacts too quickly to false measurements, a particle's validity v_p is filtered over n frames (currently, n is configured to 60):

$$v_p = \frac{v_p^{old} \times (n-1) + v_c}{n} \tag{5.1}$$

Furthermore, the addition of a base weighting w_b in the computation of a particle's weighting w_p contributes to a more stable resampling process:

$$w_p = w_b + (1 - w_b) \times v_p \tag{5.2}$$

5.1.2 Sensor Resetting based on Field Features

When using a particle filter on a computationally limited platform, only few particles, which cover the state space very sparsely, can be used. Therefore, to recover from a delocalization, it is a common approach to perform *sensor resetting*, i. e. to insert new particles based on recent measurements [14]. The new field features provide exactly this information – by combining multiple basic field elements in a way that a robot pose (in global field coordinates) can be derived directly – and thus are used by us for creating new particles. These particles are provided as *AlternativeRobotPoseHypothesis* by the AlternativeRobotPoseProvider.

Overall, this approach provides a much higher number of reliable pose estimates than the previous goal-based approach, as the field lines on which it is based can be seen from many perspectives and have a more robust context (straight white lines surrounded by green carpet) than the noisy unknown background of goal posts.

As false positives can be among the field features, e. g. caused by robot parts overlapping parts of lines and thereby inducing a wrong constellation of elements, an additional filtering step is necessary. All robot poses that can be derived from recently observed field features are clustered and only the largest cluster, which also needs to contain a minimum number of elements, is considered as a candidate for a new sample. This candidate is only inserted into the sample set in case it significantly differs from the current robot pose estimation.

To resolve the field's symmetry when handling the field features, we use the constraints given by the rules (e. g. all robots are in their own half when the game state switches to *Playing* or

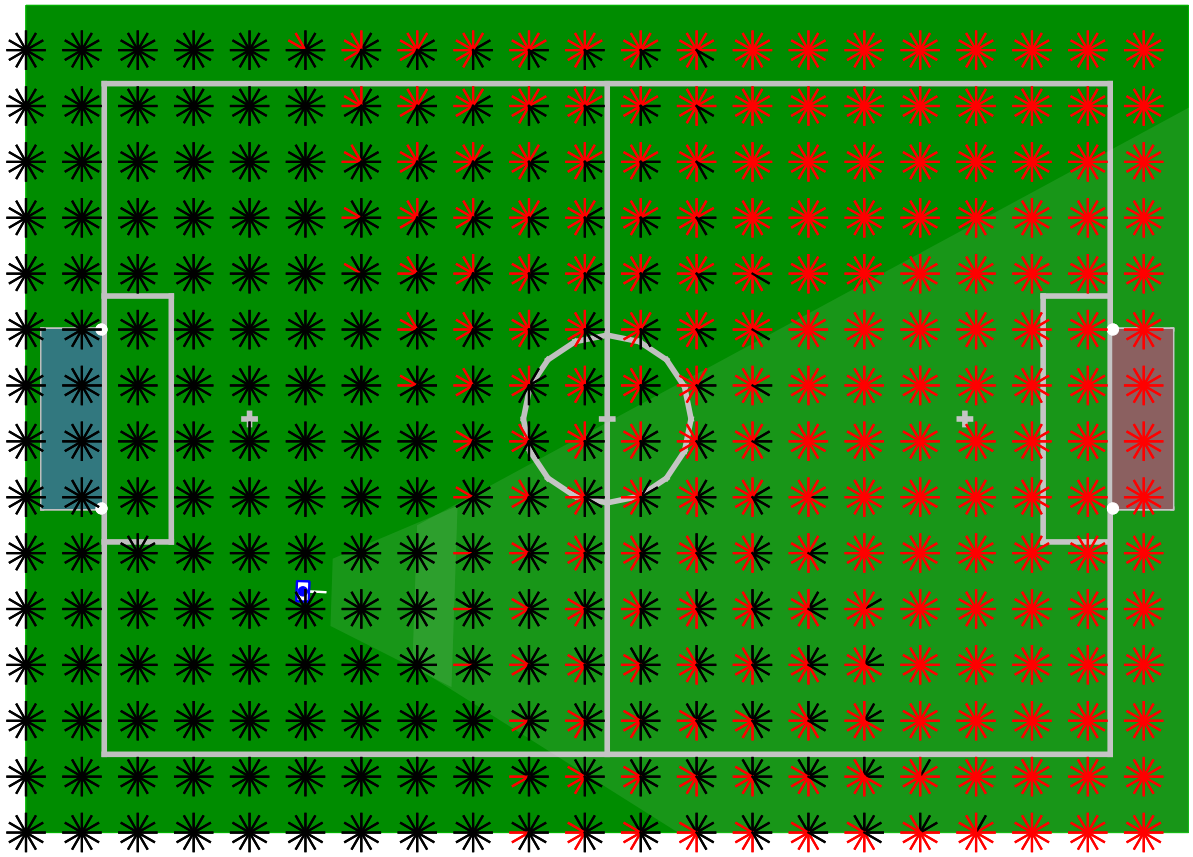


Figure 5.2: Whenever a new alternative pose is inserted into the sample set, it has to be decided, if it can be used directly or if it needs to be flipped. This decision depends on the current robot pose. This figure visualizes the possible decisions for one example robot pose (located on the right side of the own half). The stars denote example alternative poses. A red line means: *if a new pose is computed here, it has to be flipped before its insertion into the sample set.* Consequently, black lines mean that the pose can be used directly. As one can see, the current formula prefers the direction towards the opponent goal over the current position.

when they return from a penalty) as well as the assumption that the alternative that is more compatible to the previous robot pose is more likely than the other one. An example is depicted in Fig. 5.2. This assumption can be made, as no teleportation happens in real games. Instead, most localization errors result from situations in which robots lose track of their position and accumulate translational and rotational errors.

5.1.3 Handling the Field's Symmetry

As aforementioned, the self-localization already includes some mechanisms that keep a robot playing in the right direction. However, it still happens that sometimes a single robot loses track of its playing direction. To detect such situations, a separate module – the SideConfidenceProvider – compares a robot's ball observations with those of the teammates. If the current constellation indicates that the robot has probably been flipped, the representation *SideConfidence* is set accordingly and the subsequently executed SelfLocator can mirror all particles.

In the past, the comparison of the ball observations has been made between the own observation and a special team ball, similar to the one described in Sect. 5.2.3. As such a team ball always

describes an interpolated position and might be strongly influenced by single robots, we have replaced this implementation by a list of mutual agreements. Whenever a teammate communicates a new ball, it is checked, whether both robots can *agree* about the ball position or if they *disagree*, i. e. agree about the flipped alternative. If none of both alternatives appears to be likely, the status is considered to be *unknown*. The SideConfidenceProvider keeps an internal list of all teammates and the result of the last comparisons.

To agree about a ball position, both observations need to have occurred within a certain amount of time (currently one second) and at a similar place (currently up to one meter distance). As not all robots can see the ball frequently, each agreement / disagreement will be remembered for some time (currently eight seconds). However, if the (own or teammate) ball is located at certain positions, it will not be used to compute any agreement / disagreement: as the field is point-symmetric, the area around the field's center is useless for any computations; as there might be false positives on penalty marks or on lines that are close to robots or close to other lines, all balls close to any of these are discarded. Furthermore, as the robots have to agree within a certain time slice, a fast rolling ball could be perceived at very different positions and has to be ignored, too.

Finally, to make a decision whether a robot is flipped or not, the list of mutual agreements is checked:

- The localization is considered as being correct, if the robot agrees with more teammates about the ball position than it disagrees with.
- The robot is considered as flipped, if it disagrees with multiple teammates but does not agree with anyone.

5.2 Ball Tracking

To keep track of the current ball state, all robots maintain two ball models: a local one that estimates the ball position and velocity based on the own observations and a global one that fuses the own observations with observations communicated by the teammates. Furthermore, to predict upcoming positions of rolling balls, a model for the current carpet's friction can be learned.

5.2.1 Local Ball Model

Given the ball perceptions described in Sect. 4.2, the BallLocator uses Kalman filters to derive the current ball position and velocity, represented as the *BallModel*. 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 these 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

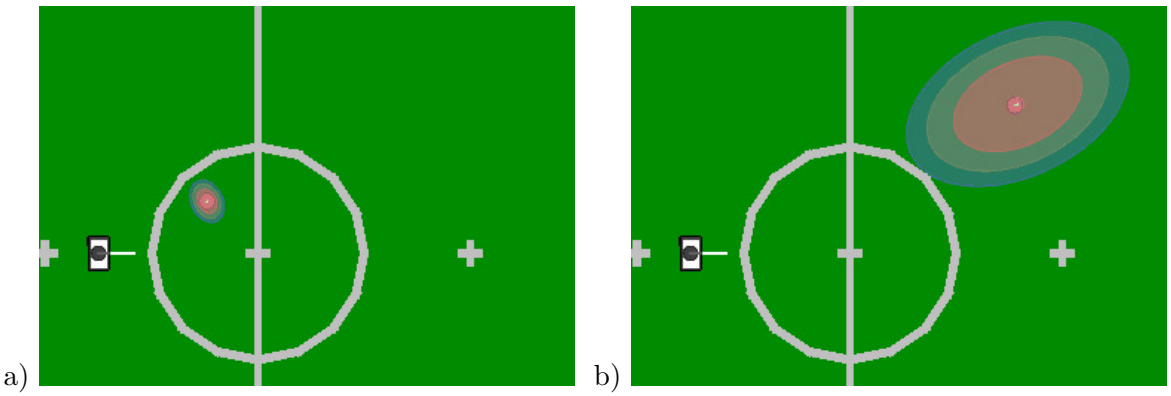


Figure 5.3: Ball model and measurement covariances for (a) short and (b) medium distances. The orange circles show the ball models computed from the probability distribution. The larger ellipses show the assumed covariances of the measurements.

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.

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.

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. 5.3).

The major parts of this module remained unchanged for many years. However, the introduction of the new black and white ball required the addition of a few more checks. Before 2016, the number of false positive ball perceptions has been zero in most games. Hence, the ball tracking was implemented as being as reactive as possible, i. e. every perception was considered. Although the new ball perception is quite robust in general (cf. Sect. 4.2), several false positives per game cannot be avoided due to the similarity between the ball's shape and surface and some robot parts. Therefore, there must be multiple ball perceptions within a certain area and within a maximum time frame before a perception is considered for the tracking process. This slightly reduces the module's reactivity but is still fast enough to allow the execution of ball blocking moves in a timely manner. Furthermore, a common problem is the detection of balls inside robots that are located at the image's border and thus are not perceived by our software. A part of these perceptions, i. e. those resulting from our teammates, is excluded by checking against the communicated teammate positions.

By adding many checks to exclude potentially false ball observations in perception (cf. 4.2) as

well as during state estimation, many true positives become rejected, too. Especially many rolling balls often fail the check for the black and white pattern. However, perceiving rolling balls and predicting their velocity is important to successfully perform blocking motions. To compensate for this problem, the BallPerceptor also returns some ball percept candidates that did not pass all checks, tagging them as being a *guess*. Subsequently, the 2017 BallLocator performs a check, if a series of these guessed candidates is on a line that is compliant with the hypothesis of a rolling ball. If this is the case, these perceptions can be used in the ball state estimation process, too.

5.2.2 Friction and Prediction

For a precise Kalman filter prediction step as well as for an estimation of a rolling ball's end position, which is required by some behaviors, a model of the friction between ball and ground is essential. For this model, we assume a linear model for ball deceleration

$$s = v \times t + \frac{1}{2} \times a \times t^2 \tag{5.3}$$

with s being the distance rolled by the ball, t the time, and a the friction coefficient. The coefficient can be configured for each individual field and has to be set in the file *fieldDimensions.cfg* (cf. Sect. 2.9).

The coefficient can be guessed by rolling a ball and comparing its end position with the predicted end position. To make this process more convenient, the code contains the module FrictionLearner, which is deactivated during games, in order to automatically determine the floor-dependent friction parameter. When the module is active, a robot with remote connection can be placed on the field and the ball has to be rolled through its field of view with medium speed. Preferably, the start and end position of the ball should both be outside the robot's field of view and the field should be even. The module buffers all ball perceptions and after the ball is outside the field of view or has come to a stop, a least squares optimization determines the coefficient a that provides the best explanation for the ball's movement. The result is printed to the SimRobot console window afterwards. As this process is subject to noise and outliers happen quite often, it should be carried out multiple times and the results have to be checked and compared carefully by the user.

All computations for ball state prediction that involve friction have been encapsulated in the class *BallPhysics* to be used by different modules.

Some modules, such as the ball perception (cf. Sect. 4.2), might require the current ball position estimate before the *BallModel* has been updated in current execution frame. Hence, the WorldModelPredictor provides a *WorldModelPrediction* by using the previous frame's robot pose and ball state estimate and applying odometry and motion updates.

5.2.3 Team Ball Model

The team 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 team 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.

Based on these factors, a common ball model, containing an approximated position and velocity, is calculated and provided as the representation *TeamBallModel*.

Among other things, the team 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.

5.3 Obstacle Modeling

Similar to the previously described ball tracking, all robots maintain two different models for the obstacles in their environment, one based solely on own observations and one incorporating information sent by teammates.

5.3.1 Local Obstacle Model

To compute the local obstacle model, namely the *ObstacleModel*, the ObstacleModelProvider creates and maintains an Extended Kalman Filter for each obstacle. All obstacles are held in a list. The position of an obstacle is relative to the position of the robot on the field plane. Sources for measurements are visually recognized robots (sometimes including a team identification by jersey color, sometimes not), and sensed contacts with arms and feet. Arm and foot contacts are interpreted as an obstacle somewhere near the shoulder or foot. In the prediction step of the Extended Kalman Filter, the odometry offset since the last update is applied to all obstacle positions. The update step tries to find the best match for a given measurement by using Euclidean distance and updates that Kalman sample accordingly. If there is no suitable sample to match, a new one is created. Due to noise in images and motion, not every measurement might create an obstacle if no best match was found. To prevent false positive obstacles in the model, there is a minimum number of measurements within a fixed duration required to generate an obstacle. If an obstacle was not seen for some time, it is removed. This might also happen if an obstacle cannot be perceived although its estimated position is in the current field of view.

5.3.2 Global Obstacle Model

As in some situations, a local model as base for team cooperation is not sufficient, an additional global obstacle model – the *TeamPlayersModel* – is computed by the TeamPlayersLocator, which makes use of communicated percepts.

5.3.2.1 Positions of Teammates

For the coordination of the own team, for instance for the role selection or for the execution of certain tactics, it is important to know the current positions of all teammates. Computing these positions does not require special calculations such as filtering or clustering, because each robot sends its already 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 just 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.

5.3.2.2 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 to compute their positions. Compared to only using the local models, which are solely based on local measurements and may differ for different teammates, it is more difficult 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 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 that are located near a teammate are used.

After the clustering is finished, the positions inside each cluster that represents a single opponent robot have to be merged to a single position. For this 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. In addition, the four goal posts are also regarded as opponent robots because they are obstacles that are not teammates.

5.4 Field Coverage

Since the introduction of the new ball in 2016, it is not possible to detect it across the whole field. In addition, there exist several configurations that obstruct ball detection from certain perspectives, such as a ball behind another robot. As a consequence, there are more periods of time during which no robot knows the current ball position. To overcome this problem, we carry out a cooperative ball search (as presented in [11]), which takes into account which parts of the field are actually visible for each robot and accordingly computes search areas for each individual robot. Due to the team-wide ball model (cf. 5.2.3), 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 that its team members do the same. Knowing which parts of the field are visible to the team members, searching the ball can happen dynamically and effective.

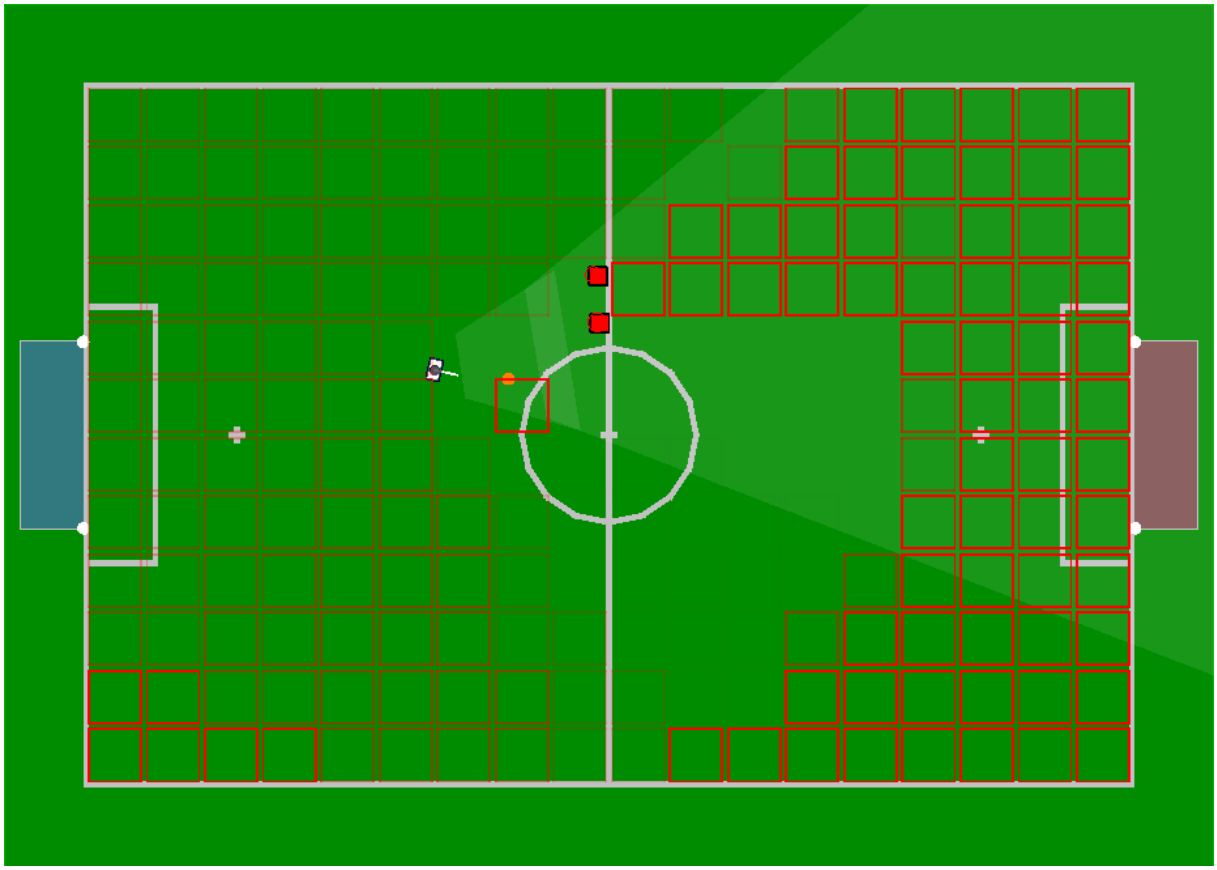


Figure 5.4: The local field coverage grid after walking around on the field. The more intense the red borders of the cells are, the less these cells are covered. Two opponent robots prevent cells on the top right of the field from being marked as visible.

5.4.1 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 0.25 m^2. To determine which of the cells are currently visible, the current field of view is projected onto the field. Then, all cells whose center lies within the field of view are candidates for being marked as visible.

There may be other robots obstructing the view to certain parts of the field. Thus, depending on the point of view of the viewing robot, another robot may create 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. 5.4.

Having determined the set of visible cells, each of the cells gets a time stamp. These time stamps are later used to build the global field coverage model. They are also used to determine the cell, which has not been visited the longest, to generate the head motion for scanning the field while searching for the ball.

A special situation arises when the ball goes out. If this happens, the time stamps of the cells become reset to values depending on the next point of entry. The cell of the entry point gets the lowest time stamp to force robots to look there first. The point of entry is determined by the last known intersection of the ball trajectory and a field boundary, before the GameController sent the information that the ball is out. In addition, all other cells that are near the field border, are set to a slightly higher time stamp, so that the side lines become targets for the ball search

in case of a referee error or an error in the estimation of the ball motion.

Another special case is the ball's movement as represented in the *BallModel*. If the ball touches a cell, its time stamp is set back to zero. This way, if the ball is lost while moving, the robots will search for it along its last known path first.

5.4.2 Global Field Coverage

To make use of the field coverage grids of the other robots, each robot has to communicate its grid to its teammates. To do so, each robot maintains a global field coverage grid, which is incrementally updated in every team communication cycle and includes all information from its own local field coverage. This global grid looks roughly the same for all teammates. Hence, calculations based on the grid come to sufficiently similar results for all teammates.

Since we want to change the actual time stamp of each cell in case the ball goes out, we actually need two time stamps in team communication. One to determine the time when the cell was last seen as described above and one to keep track of the time when this value was changed. The former will be called *coverage value* from here on. The merging of the grids is then done simply by looking at the time stamp of each cell in the existing global grid and a received local grid. If the received cell has a more recent time stamp, it is carried over to the global grid.

Given the current field dimensions, 4 byte time stamps, and a cell's edge length of $\frac{1}{2}m$, there are $2 * 4\ bytes * \frac{6m*9m}{(\frac{1}{2}m)^2} = 1728\ bytes$ that 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 the current rules, the grid is not sent as a whole but in intervals of 18 cells each. In addition, the time stamps are compressed into 2 byte values and a base time stamp for each interval. This way, only $18 * 2 * 2 + 4\ bytes = 76\ bytes$ are required each cycle.

To calculate each robot's patrol target, the cells of the global grid are sorted descending by their *coverage value*. For each cell in this list, the distance to each robot is calculated. If the distance of the cell to the current robot is less than the distance to one of its teammates, the patrol target is found.

5.5 Whistle Recognition

The implementation for detecting a referee whistle is more or less the same as in the last three years. The approach is based on the correlation between the robot's current audio input and a previously recorded reference whistle. As different whistles might be used during a competition and as a single whistle might produce quite different sounds, our implementation allows to compare the audio input with multiple reference whistles since 2016. Furthermore, the 2017 implementation adds a more fine-grained configuration for comparing the correlations of different whistles. The whistle recognition is implemented in the WhistleRecognizer module, providing the *Whistle* representation. In addition to this detection, we also implemented a mechanism for a team-wide majority vote.

5.5.1 Correlating Whistle Signals

For the whistle detection, we use a cross-correlation. To calculate the correlation between a given whistle signal and a actual recorded signal in a fast way, we transform the actual signal into the frequency domain, multiply it with the conjugate complex stored reference whistle signal

and transform it back into the time domain, where we detect the compliance with the reference signal.

First, we need a reference signal s_{ref}. The reference signal is also recorded by the robot, where we take a window length of N samples. We use the sampling frequency f_s of the robot's audio hardware without downsampling. The signal must be extended with zeros by length N, because the result of the correlation is twice the window length. In addition, correlating signals by Fourier transformating them results in a cyclic correlation, which is avoided by the zero padding. This signal is transformed to the frequency domain by a Fast Fourier Transformation (we use the highly efficient FFTW3 implementation by [4]).

$$\mathcal{F}(s_{\text{ref}}[n]) = \underline{S}_{\text{ref}}[k]. \tag{5.4}$$

The underline denotes complex values. This signal is stored as a reference and has not to be recalculated in every step.

For the actual recorded signal s_{act}, we use the same procedure by adding zeros to get the doubled length and transform it into the frequency domain to get $\underline{S}_{\text{act}}$.

We make use of the fact that a correlation can be represented by a convolution using one signal in reorder. The advantage of a convolution is that it can be calculated easily in the frequency domain by just multiplying the signals. As we do not want to store the signal in reverse order, we can transform the reversion into the frequency domain, too. Thus, we get

$$s_{\text{ref}}[n] * s_{\text{act}}[-n] = \underline{S}_{\text{ref}}[k] \cdot \underline{S}^*_{\text{act}}[k], \tag{5.5}$$

where $\underline{S}^*_{\text{act}}$ denotes the conjugate complex of the signal.

As it does not matter which signal is conjugated, we store the reference signal as conjugate. Multiplying both signals gives us the result in the frequency domain and the inverse Fourier Transform is the correlation result s_{corr} in the time domain.

The best result can be achieved if the reference signal is equal to the actual signal or the negative actual signal. Using this value, which is the autocorrelation value, and dividing the correlation signal s_{corr} by this factor, we always get a percent value of the best possible result. An example is depicted in Fig. 5.5.

If the percentage is above a defined threshold, the whistle is detected. As step by step formulation, we have the following implementation in discrete form:

$$\begin{aligned}
s'_{\text{act}}[n] &= \begin{pmatrix} s_{\text{act}}[n] & \mathbf{0}_N \end{pmatrix} \\
\underline{S}_{\text{act}}[k] &= \mathcal{F}\left(s'_{\text{act}}[n]\right) \\
s_{\text{corr}}[n] &= \mathcal{F}^{-1}\left(\underline{S}_{\text{act}}[k] \cdot \underline{S}^*_{\text{ref}}[k]\right) \\
\varphi_{cc} &= \frac{\max(|s_{\text{corr}}[n]|)}{\varphi_{ac\max}} \in [0,1] \\
w &= \begin{cases} true & \text{for } \varphi_{cc} \geq \alpha \\ false & \text{otherwise} \end{cases}.
\end{aligned}$$

φ denotes the correlation value for the cross correlation and the auto correlation of the reference signal respectively. The parameters n and k are discrete time and frequency respectively.

The correlation procedure is repeated for every stored reference whistle. The computed *Whistle* representation contains the information about the signal that has the best correlation. The module's required computing time scales linearly with the number of reference whistles. During RoboCup 2017, we used the reference signals of three different whistles. However, all computations are only executed during the game's *SET* state.

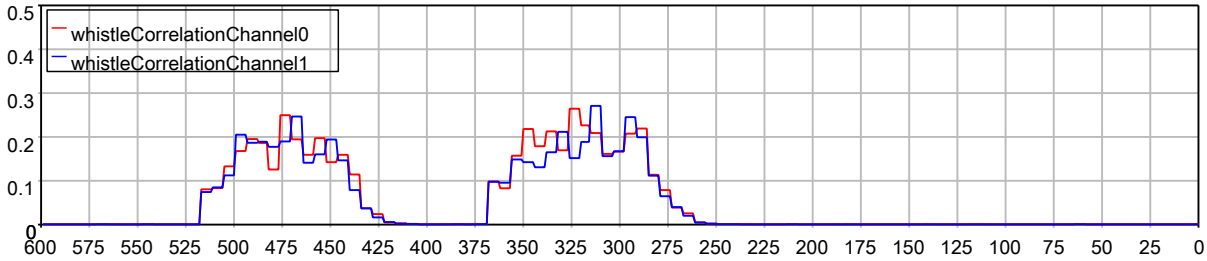


Figure 5.5: Correlation between the current signal and a pre-recorded reference whistle signal over 600 frames (equal 10 seconds). During this time, a whistle has been blown twice. The threshold for whistle detection during this experiment has been set to 0.15. The differently colored lines depict the different results from the two used audio channels.

5.5.2 Sound Playback during Whistle Recognition

It is possible that a sound played by the robot is recognized as a whistle. To prevent false positives during sound playback, the actual whistle recognition is skipped and the confidence is negated. The confidence has to be negative to exclude the robot from the majority vote, which is described in Sect. 5.5.3. When the sound playback is finished, the confidence is reset to the original value.

5.5.3 Majority Vote

To decide whether or not a whistle has been blown, we don't rely on the detection of each individual robot alone. Instead, we take all detections on the different audio channels on each robot into account and start a majority vote on whether or not the team should start playing. To do this, each robot sends the time stamp and its confidence of the last whistle detection to the teammates via team communication. If the robot never detected a whistle, this time stamp is zero. The confidence is calculated as follows:

- 100: If the robot detected the signal on both audio channels
- 66: If the robot detected the signal on one audio channel while the other one is deactivated due to being damaged
- 33: If the robot detected the signal only on one audio channel although both microphones are supposed to be working fine
- 0: If the robot did not detect a signal
- -1: If both audio channels are deactivated (i. e. the robot is deaf)

All detections that happened after the beginning of the last *SET* state and have a confidence $>= 0$ are now taken and sorted by their time stamps. Each detection is now grouped with all other detections that happened within 500 milliseconds of each other. The total confidence of each group is then divided by the number of non-deaf robots. If the average confidence of one group is above 48 (i. e. at least half of the team has heard a whistle), the team starts playing.

Chapter 6

Behavior Control

The part of the B-Human system that performs the action selection is called *Behavior Control*. It also runs in the context of the process *Cognition*. The behavior is modeled using the C-based Agent Behavior Specification Language (CABSL). The main module – BehaviorControl – provides many representations that either control the actions of the robot or are sent to teammates. It is accompanied by several additional modules such as the VGPathPlanner (cf. Sect. 6.4), the KickPoseProvider (cf. Sect. 6.5), the CameraControlEngine (cf. Sect. 6.6), and the LEDHandler (cf. Sect. 6.7) as well as the RoleProvider (cf. Sect. 6.2.1).

This chapter begins with a short overview of the behavior specification language 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 2017.

6.1 CABSL

CABSL [27] 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. [24, Chap. 5]) that we used a few years ago. 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.

```
option(exampleOption)
{
  initial_state(firstState)
  {
    transition
    {
      if(booleanExpression)
```

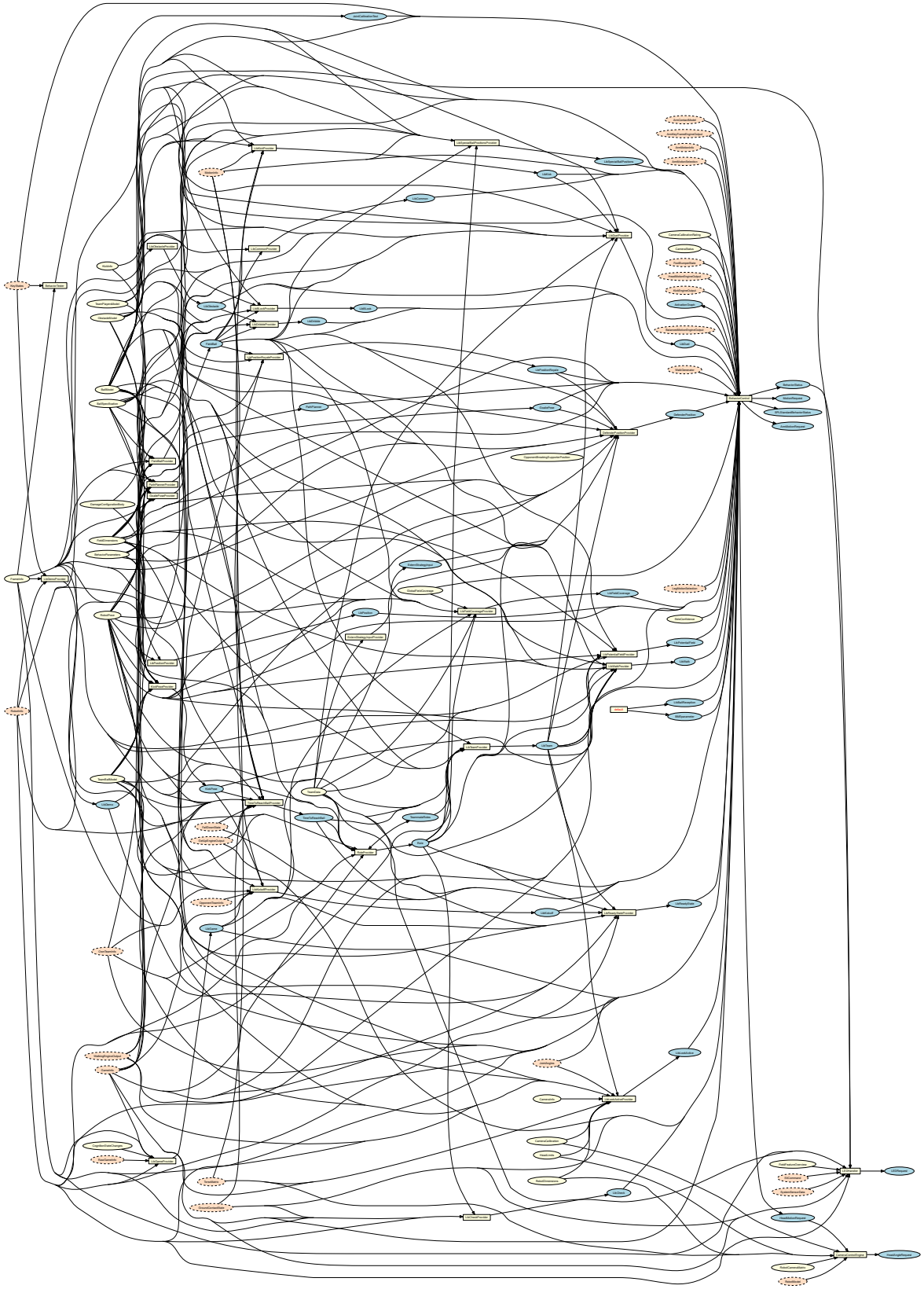


Figure 6.1: Behavior control module graph. Behavior control modules are depicted as yellow rectangles. All representations they provide are shown as blue ellipses. The representations they require are shown as ellipses colored in either yellow (if they are provided by other *Cognition* modules) and orange (if they are received from the process *Motion*).

```
        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 "else-branches" of the common transition, because they are just 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. The specification of parameters uses the same syntax as the streamable data type definition in our system. Thereby, the actual parameters of each option can be recorded in log files and they can also be shown in the behavior dialog (cf. Fig. 6.2). Please note that the source code shown in Fig. 6.2 is still understood by a C++ compiler and thereby by the editors of C++ IDEs.

6.2 Behavior Used at RoboCup 2017

The behavior 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

```
option(SetHeadPanTilt,
       (float) pan,
       (float) tilt,
       (float)(pi) speed,
       ((HeadMotionRequest) CameraControlMode)(autoCamera) camera)
{
  initial_state(setRequest)
  {
    transition
    {
      if(state_time > 200 && !theHeadJointRequest.moving)
        goto targetReached;
    }
    action
    {
      theHeadMotionRequest.mode = HeadMotionRequest::panTiltMode;
      theHeadMotionRequest.cameraControlMode = camera;
      theHeadMotionRequest.pan = pan;
      theHeadMotionRequest.tilt = tilt;
      theHeadMotionRequest.speed = speed;
    }
  }

  target_state(targetReached)
  {
```

robot2.behavior

Soccer	47.04
state = playSoccer	44.90
HandlePenaltyState	44.90
state = notPenalized	44.90
HandleGameState	44.90
state = set	3.86
Activity	47.04
activity = standAndWait	
state = setActivity	47.04
Stand	5.98
state = requestIsExecuted	4.91
HeadControl	44.90
state = lookLeftAndRight	3.86
LookLeftAndRight	3.86
state = lookRight	0.63
SetHeadPanTilt	3.86
pan = -0.872665	
tilt = 0.401426	
speed = 1.74533	
state = setRequest	0.63

Figure 6.2: The behavior view shows the actual parameters, e. g. of the option *SetHeadPanTilt* (left: source, right: view). The display of values equaling default parameters (specified in a second pair of parentheses) is suppressed. On the right of the view, the number of seconds an option or state is active is shown.

is then used by the "HeadControl" to move the head as described in Sect. 6.2.7. Both parts are called in the behavior's main option named *Soccer*, which also handles the initial stand up of the robot if its chest button is pressed as well as the sit down motion, if the chest button 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 chest button 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 the "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. 6.2.6). 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. 6.2.1) where each robot gets chosen for one of the different roles (cf. Sect. 6.2.1).

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 a robot, that robot immediately turns in the direction of the ball and tries to take the pass.

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 provided by the FieldCoverageProvider (cf. Sect. 5.4). If, after a whole turn, no ball was observed, each robot patrols to a position provided by the GlobalFieldCoverageProvider (cf. Sect. 5.4). This also includes a special handling for balls that went out.

6.2.1 Roles and Tactic

Our behavior is aimed at a defensive style of play and minimal movement. This prevents opponents from dribbling into our goal and saves battery power. Apart from this, it also prevents joint heat and therefore loss of joint stiffness / movement quality, which is needed in critical moments. To accomplish this, we played with the following lineup:

1. One Keeper: The goalkeeper robot.
2. Two Defenders: Robots that are positioned defensively to help the Keeper by defending their own goal.
3. One Striker: The ball-playing robot.
4. One Supporter: A robot that is positioned offensively to help the Striker.

In a lineup with less than five robots, at least one robot should be the Defender. The role selection is done by the RoleProviderRoyale. The general procedure is that every robot is calculating a role selection suggestion for itself and each connected teammate, but uses the role selection of the captain robot. The captain robot is determined by the lowest (not penalized) connected teammate number. The reason behind this procedure is to always have a suggestion available but inhibit the chance that the robots play with different role selections at the same time.

The keeper role is always assigned to the robot with the number one and will not be changed at any point. The Striker role will be assigned to the robot which can reach the ball fastest. The calculation will favor the current striker to prevent oscillations. All other available robots will now distribute to the two Defender roles and one Supporter role. To decide which robot becomes a Supporter and which a Defender, the global x-position will be used with a favor for the previous role. If there are more robots than the goalkeeper and the striker, at least one defender will be assigned.

6.2.2 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 striker walks to the *KickPose* (cf. Sect. 6.5). Depending on the distance to the target, the coordinates are then either passed to the PathPlanner (cf. Sect. 6.4) 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. 8.3).

After reaching the pose, the striker either executes the kick that was selected by the *KickPoseProvider* (cf. Sect. 6.5) 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. 6.3):

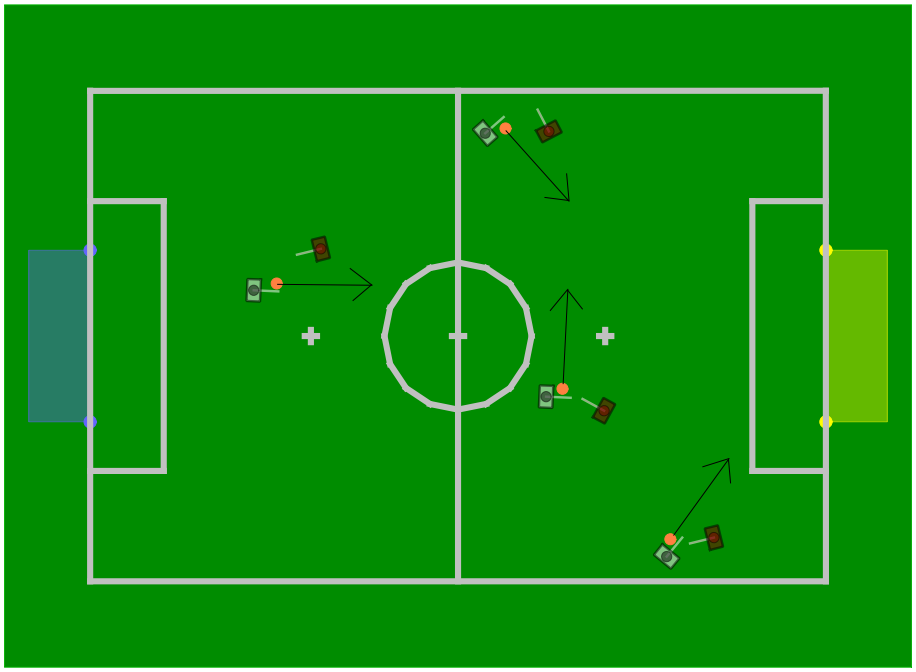


Figure 6.3: A set of possible tackling situations. The black lines mark the desired kick direction.

- 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 our 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.

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.

KickingOut: The striker will kick the ball out of the field, if the position at which it should be put in again is strategically better than the current ball position.

6.2.3 Supporter

Our supporter knows two modes: Directly supporting the striker or waiting near the opponent goal. Directly supporting means following the striker to catch the ball in case the striker loses the ball against an opponent. The aim of waiting near the opponent goal is to catch the ball in case the opponent defense successfully blocked a long range shot at their goal.

6.2.4 Defender

A Defender has five modes, which differ only slightly. These are: back-left, back-middle, back-right, forward-left and forward-right – the areas where the robot can position itself. Whereas the role Defender is taken upon consultation with the robot's team members, the decision of the taken mode is made by the Defender alone. It makes its decision on the basis of the received pose of the second Defender (or on the absence of such data). If there are two Defenders, they must be either left or right and never on the same side. In addition, one of them must hold a back-mode but the other one may be forward. Otherwise, if there is just one defending robot, it must be a back one.

In each frame, the Defender recomputes

1. if it is left or right (or in special case middle),
2. if it is forward,
3. which position it should take, and
4. if it should move.

The Left / Right Decision

If the Defender is left, it means that it will calculate a position on the left side of the virtual line between the calculated goalkeeper position and the ball. A middle defending position means that the Defender stands on this line. This position is just a special case if neither a Keeper is inside the penalty area nor a team member performs a second Defender. In all other cases, you do not want to interfere with the direct goalkeeper sight of the ball.

- Being a single Defender (with a present Keeper), a robot will always decide for the same side like last frame as long as the ball is not too far on the other side.
- In the case of two Defenders, each robot will compare its (signed) distance to the goalkeeper-ball line and the distance of the last known position of the other Defender to that line. If the robot can clearly decide if it is more left of the line, it takes the decision. If this is not the case, it tries to decide by looking up which of the both poses is clearly more left on the field (given the global coordinate system as described in Sect. 4.1.2). If the robot still does not know which side it belongs to, it uses the decision of the last frame.

The Back / Forward Decision

A back position is defined as a position directly outside the penalty area, a forward position could be farther away. In the case of a single defending robot, the position will be back as mentioned above. In the other case, the robot decides on the basis of its actual position, if it is distinctly forward or back. It does the same for the position of the second Defender. The robot will hold its decision, if it differs from the position about the second Defender or if it is back and the ball is not far away. If both cases are not applicable and the robots are clearly looking in different directions, the robot that is looking in the direction of the opponent goal will move forward. If none of the cases were applicable and the Defender robot is on the same side as the ball, it is moving forward. If the ball side is not clear, both Defenders stay back.

The Position Selection

All selected positions for the Defender are located on a semicircle, which is centered around the center of the own goalposts. The radius is decided by the previously described back / forward mode. If it is back, the semicircle goes barely around the penalty area, but the maximum semicircle for a forward Defender gives the own goal a wide berth. As long as the ball is not inside the own penalty area, the forward radius is always smaller than the distance of the semicircle origin to the ball. The Defender will take a position on the selected semicircle with that he can cover the space between the goalpost on his side and the goalie blocking coverage area. This means that a forward Defender is standing nearer to the goalie-ball-line, which results in a restriction to the goalkeeper's sight of the ball. At that range our goalie will not see the new ball as good as the old one. Thus, this positioning is still acceptable. To avoid that the distance of the back-standing robots to the penalty area gets too large, the semicircles may be cut along the lines of the penalty area. There is also a special handling when the ball is entering the penalty area or is lying close to the own ground line.

The Decision to Move

Beside the general idea, there are no sophisticated methods implemented to make this decision. Instead, our robots always wanted to move to the calculated position. The underlying path finding modules inhibits this for very short distances.

It is important that the Defender is doing its job, but as the last instance, between opponent scoring or a successful defense, there is still the goalkeeper. To give him time for a timely reaction, it must see the ball early enough. To maximize this time, the Defenders are also performing arm movements to make themselves as thin as possible, if the goalkeeper-ball line comes close to the robot.

6.2.5 Keeper

The keeper's 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 again. 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. When walking omni-directionally, the keeper avoids obstacles in the same way as

the other players do. The second situation, in which the robot leaves the own goal, is given 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 a goalie is not penalized for pushing 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 like described below. 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 looks 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 sidewards 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 2017). 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 such a 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 a rough situation in the penalty area, the keeper's rotation is wrong by 180°, letting the robot turn in the wrong direction and stare into the goal net. This situation is resolved by the SelfLocator by checking for a close *FieldBoundary* and getting no percepts of major field elements for a certain time. In this case, the goalie localization is turned back by 180 degrees, letting the behavior move to the right pose again.

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 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 main idea of the goalie positioning is to inhibit the opponent robot to score a goal, i. e. to inhibit the ball to move at a straight line into the goal. We are directly calculating which parts of the field will be covered with a specific robot pose. The perfect position would be if the robot is able to cover the whole goal by just standing. Of course, in nearly every situation this is not recommendable because of the distance to the goal the robot must hold to meet this condition. The second best situation would be if our robot is able to cover the whole goal with our blocking motion (sitting down and spreading the legs) since this motion needs nearly no time to perform. This is why we are trying to maximize the goal coverage with this motion at all time. This intention is restricted by the secondary idea of the goalie positioning: the goalie should also be able to perform his dive motion at any moment. It is possible to perform a dive

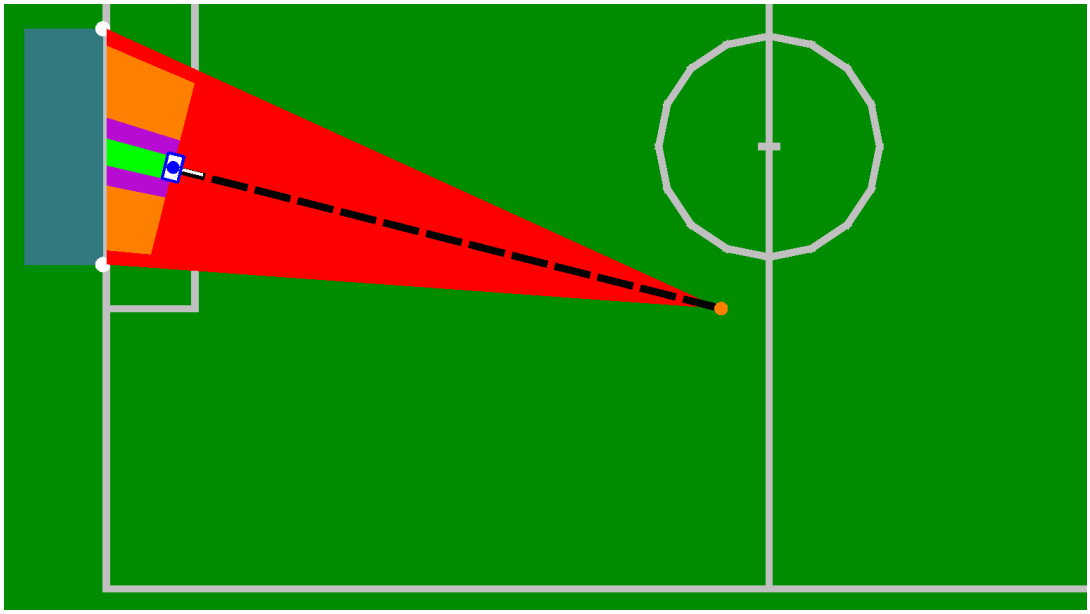


Figure 6.4: The blue figure indicates the pose of a goalkeeper that is rotated towards the ball (depicted as a small orange circle). The area between ball and goal is shown in red. The areas that are covered with the keeper actions `standing` / `blocking` / `diving` are colored in green / violet / orange.

motion if the goalie is inside the penalty area and if it will not dive into the goal post/frame to prevent damages. In addition, a restriction is added that the robot is not touching the penalty area lines with the feet to keep the lines detectable for the vision system. This should help the robot to maintain an accurate self-localization which is essential to find the best spot for covering the goal.

With this goalie positioning, we are able to cover (nearly) the whole goal at all time and to perform the dive motion before the ball crosses the keeper. The coverage at an example situation is depicted in Fig. 6.4;

6.2.6 Kickoff

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*.

6.2.6.1 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 arrive 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 will walk to that position. If our team has kickoff, this position is equal to the center of the field. If the opponent team has kickoff, this position is between the center of the field and our own goal, right at the border of the center circle.

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.

6.2.6.2 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.

6.2.7 Head Control

The head control sets 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 BallPerceptor, and the LinePerceptor provide input for the modeling modules (cf. Sect. 5)

that provide localization information, and the ball position. However, providing images from all relevant areas on the field is often mutually exclusive. For instance, 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 before the game, after the game, and when the robot is penalized.

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. 6.3).

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.

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 field features, 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 will 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, every robot that is currently not looking at the ball will turn its 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, as 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 is moving towards the robot – but still has to 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.

6.3 Penalty Shoot-out Behavior

The penalty shoot-out behavior of 2017 did not change significantly to previous years. However, particularly with regard to the *Penalty Shot Challenge* (cf. Sect. 9.1), some details have been tweaked.

Since the keeper can simply track the ball and jump to catch it, the main idea for the penalty taker is to kick the ball as near to one of the goal posts as possible. In addition, 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 posts. In previous years, our keeper was not able to cover its complete goal line. Thus, it had to make a few steps forward, thereby narrowing the angle for the penalty taker. However, due to an updated jump motion, which allows the keeper's hand to reach the goal post (cf. Fig. 9.2), this year's penalty keeper can now remain on the goal line.

The RoboCup 2017 penalty shoot-out behavior is embedded in the behavior described above. It is implemented by introducing two additional roles: *PenaltyStriker*, and *PenaltyKeeper*. Depending on the secondary game state, the current kick-off team, and the selected robot number, the RoleProvider assigns one of these two roles to the robot. In order to prevent the joints from getting too hot by long standing, all other robots, which are penalized by the GameController, sit down. Only the currently selected, non-penalized *PenaltyKeeper* or *PenaltyStriker* stands up. Game state handling and head control is processed the same way as during an ordinary game.

In detail, the two roles work as follows:

PenaltyKeeper: When the penalty shot attempt starts, the penalty keeper sits down immediately as the sitting posture allows a faster jump towards the goal posts. 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. The penalty keeper always keeps the ball in its field of view and does not look for any other features on the field.

PenaltyStriker: At the beginning of the penalty shot attempt, the robot randomly decides, if it wants to kick at 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 makes two short stops, 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. Finally, our standard forward kick is executed. There also exists a fallback behavior, in case there is just little time left for the shot. This behavior lets the robot just move towards the ball and kick it, using the KickPoseProvider (cf. Sect. 6.5). For the head control, the penalty striker simply uses `lookAtBall`, because it needs to focus at the ball all the time. This is acceptable, as due to its initial position on the field and its initial rotation towards the goal, the penalty area always remains in sight and provides enough information for precise self-localization.

6.4 Path Planner

In some situations, robots have to walk longer distances to reach their next target location, e. g., when walking to their kickoff positions or when walking to a distant ball. In these cases, a purely reactive control can be disadvantageous, because it usually would not consider obstacles that are further away, which might result in getting stuck. Therefore, our robots use a path planner in these situations since 2011. Until 2014, it was based on the Rapidly-Exploring Random Tree approach [13] with re-planning in each *Cognition* cycle. Although the planner worked quite well, it had two major problems: On the one hand, the randomness sometimes resulted in suboptimal paths and in oscillations[1]. On the other hand, it seemed that the RRT approach is not really necessary for solving a 2-D planning problem, as the planner actually did. Thus, it was slower than it needed to be.

6.4.1 Approach

Therefore, a new planner was developed for RoboCup 2015. It is a visibility-graph-based 2-D A* planner (cf. Fig. 6.5). It represents obstacles as circles on which they can be surrounded and the path between them as tangential straight lines. As a result, a path is always an alternating sequence of straight lines and circle segments. There are four connecting tangents between each pair of non-overlapping obstacle circles, only two between circles that overlap, and none if one circle contains the other. With up to nine other robots on the field, four goal posts, and the ball, the number of edges in the visibility graph can be quite high. Thus, the creation of the entire graph could be a very time-consuming task. Therefore, the planner creates the graph while planning, i. e. it only creates the outgoing edges from nodes that were already reached by the A* planning algorithm. Thereby, the A* heuristic (which is the Euclidean distance) not only speeds up the search, but it also reduces the number of nodes that are expanded. When a node is expanded, the tangents to all other *visible* nodes that have not been visited before are computed. *Visible* means that no closer obstacle circles intersect with the tangent, which

[1] Although the planner tried to keep each new plan close to the previous one.

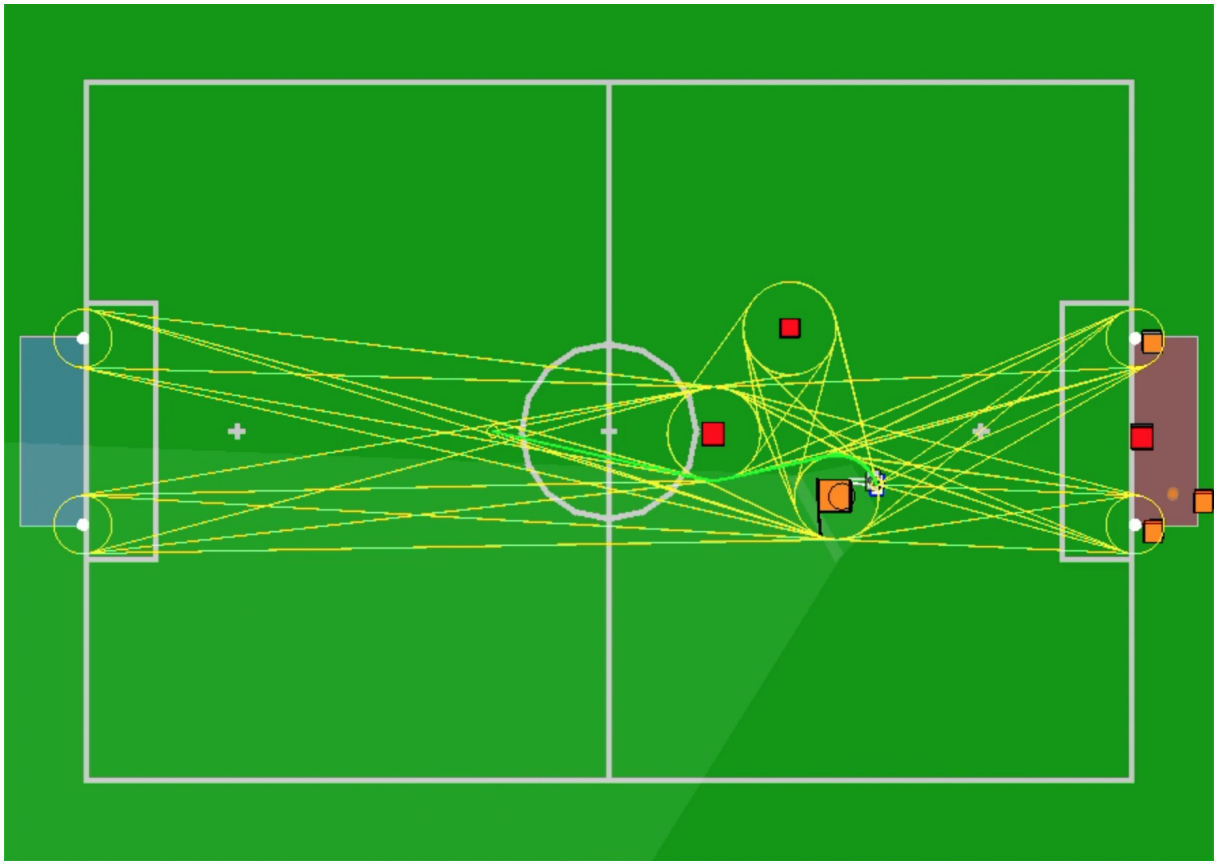


Figure 6.5: Visualization of the planning process. The robot's position is shown as a small rectangle right of the big orange square. The target position is depicted as small circle left of the center circle. Obstacles are shown as red and yellow squares, some of which are ignored. Obstacle circles are depicted in yellow. Yellow lines indicate expanded edges. The shortest path is depicted in bright green.

would prevent traveling directly from one circle to another. To compute the visibility efficiently, a sweep line approach is used. As a result, the planning process never took longer than 1 ms per *Cognition* cycle in typical games.

6.4.2 Avoiding Oscillations

Re-planning in each Cognition cycle bears the risk of oscillations, i. e. repeatedly changing the decision, for instance, to avoid the closest obstacle on either the left or the right side. The planner introduces some stability into the planning process by adding an extra distance to all outgoing edges of the start node based on how far the robot had to turn to walk in the direction of that edge and whether the first obstacle is passed on the same side again (no extra penalty) or not (extra penalty). Note that this does not violate the requirement of the A* algorithm that the heuristic is not allowed to overestimate the remaining distance, because the heuristic is never used for the outgoing edges of the start node.

6.4.3 Overlapping Obstacle Circles

The planning process is a little bit more complex than it appears at first sight: As obstacles can overlap, ingoing and outgoing edges of the same circle are not necessarily connected, because the robot cannot walk on their connecting circle segment if this is also inside another obstacle region. Therefore, the planner manages a set of walkable (non-overlapping) segments for each circle, which reduces the number of outgoing edges that are expanded when a circle is reached from a certain ingoing edge. However, this also breaks the association between the obstacle circles and the nodes of the search graph, because since some outgoing edges are unreachable from a certain ingoing one, the same circle can be reached again later through another ingoing edge that now opens up the connection to other outgoing edges. To solve this problem, circles are cloned for each yet unreached segment, which makes the circle segments the actual nodes in the search graph. However, as the graph is created during the search process, this cloning also only happens on demand.

6.4.4 Forbidden Areas

There are two other extensions in the planning process. Another source for unreachable segments on obstacle circles is a virtual border around the field. In theory, the shortest path to a location could be to surround another robot outside of the carpet. The virtual border makes sure that no paths are planned that are closer to the edge of the carpet than it is safe. On demand, the planner can also activate lines surrounding the own penalty area to avoid entering it. The lines prevent passing obstacles on the inner side of the penalty area. In addition, edges of the visibility graph are not allowed to intersect with these lines. To give the planner still a chance to find a shortest path around the penalty area, four obstacle circles are placed on its corners in this mode. A similar approach is also used to prevent the robot from walking through the goal nets.

6.4.5 Avoiding Impossible Plans

In practice, it is possible that the robot should reach a position that the planner assumes to be unreachable. On the one hand, the start position or the target position could be inside obstacle circles. In these cases, the obstacle circles are "pushed away" from these locations in the direction they have to be moved the least to not overlap with the start/target position anymore before the planning is started[2]. On the other hand, due to localization errors, the start and target location could be on different sides of lines that should not be passed. In these cases, the closest line is "pushed away". For instance, if the robot is inside its penalty area although it should not be, this would move the closest border of the penalty area far enough inward so that the robot's start position appears to be outside for the planning process so that a plan can be found.

6.5 Kick Pose Provider

To calculate the optimal position for a kick toward the opponent's goal, a module called KickPoseProvider is used. Most of the time, at least some parts of the goal are covered by opponent

[2]Note that when executing the plan, these situations are handled differently to avoid bumping into other robots.

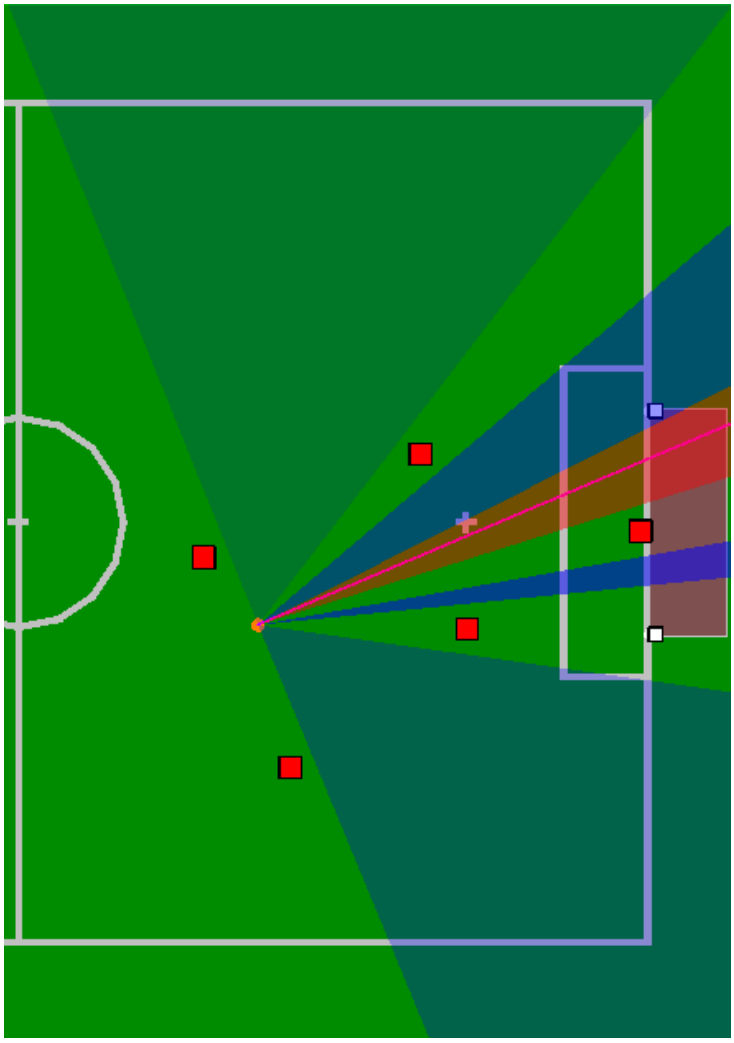


Figure 6.6: Visualization of all openings between opponent robots (red squares) as seen from the ball. The red opening was chosen to be the kick target.

robots. Therefore, we need to calculate the part of the goal that has the largest opening angle as seen from the ball (cf. Fig. 6.6). The center of this part will be the target for our kick.

Now we need to choose which kick to use. For every possible kick, 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 between the robot orientation and the target direction. Based on this information, the module calculates a *KickPose* for each kick, i. e. the position the robot has to stand at and the 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 mainly based on the execution time, which consists of the time to reach the pose and the time to perform the actual kick. Furthermore, it is taken into account whether the kick is strong enough to reach the opponent goal. To influence the probability to be chosen, the execution time can be modified. If, for instance, a kick is rather unstable and should only be used if the robot is already standing at an almost perfect position, the probability of the kick being chosen can be reduced by increasing its configured execution time.

There are some cases that need a special handling. Inside our own penalty area, we are using

only fast kicks and no walk kicks. Inside the opponent penalty area, it is the opposite choice. When the ball is near an opponent goal post or at the side of the opponent penalty area, a special side kick is used.

6.6 Camera Control Engine

The CameraControlEngine takes the *HeadMotionRequest* provided by the BehaviorControl2015 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. 8.3.4). 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.

6.7 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
Magenta	All	Robot is connected to external power source (Only if no Head LEDs are available.)
Blue	Keeper	
White	Defender	
Green	Supporter	
Yellow	Bishop	
Red	Striker	

Left Eye

Color	Information
Yellow	No ground contact
White	Ball was seen
Blue	Field Feature was seen
Red	Ball and Field Feature were seen

Torso (Chest Button)

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

Feet

- The left foot shows the team color. For instance, if the team is currently the red team, the color of the LED is red. If the team color is black, this LED will be switched off.
- 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). Without game controller access, two LEDs are blinking, the rest are off.

Head (if available)

If the robot is charging, the LEDs will be turned on and off one after another. Due to the circular arrangement of the LEDs, this looks like a rotating beam of light. If these LEDs are not available and the robot is charging, the right eye will be colored magenta.

Chapter 7

Proprioception (Sensing)

The NAO has an inertial measurement unit (IMU) with three acceleration sensors (for the x-, y-, and z-direction) and two or three gyroscopes (for the rotation around the x-, y- and in V5 NAOs also the z-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 ultrasound sensors with different measurement modes, and sensors for the load, temperature, and angle of each joint. We currently do not use the data from the ultrasound sensors.

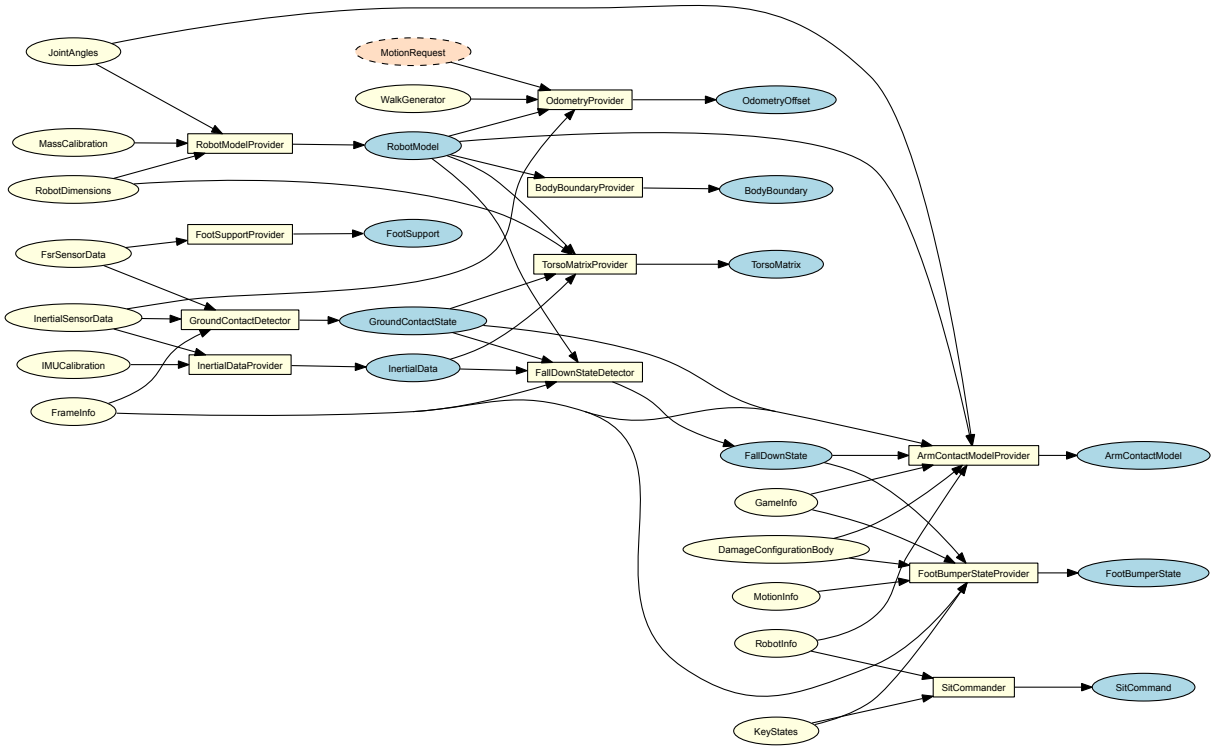


Figure 7.1: Sensing module graph. Sensing modules are depicted as yellow rectangles. All representations they provide are shown as blue ellipses. The representations they require are shown as ellipses colored in either yellow if they are provided by other *Motion* modules and orange if they are received from the process *Cognition*.

In the process *Motion*, 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 *FsrSensorData*, *InertialSensorData*, *JointSensorData*, *KeyStates*, and *SystemSensorData*. The *JointAngles* encapsulates the *JointSensorData* for the whole system, which is used to set the `WristJoints` angles in case they are not supported by the NAO version.

The *RobotModel* (cf. Sect. 7.2) provides the positions of the robot's limbs and the GroundContactDetector provides the *GroundContactState* (cf. Sect. 7.1), which is the information whether the robot's feet are touching the ground. Based on the *JointAngles*, the *InertialData*, the *RobotModel*, the *GroundContactState*, and the currently executed motion, it is possible to calculate the *TorsoMatrix* (cf. Sect. 7.4), which describes a transformation from the ground to an origin point within the torso of the NAO. The representation *InertialData* is also considered to detect whether a robot has fallen down by the module FallDownStateDetector, which provides the *FallDownState* (cf. Sect. 7.5). Figure 7.1 shows all *Sensing* modules and the provided representations.

7.1 Ground Contact Detection

To improve the handling of the robots for team members and assistant referees, it is advantageous if the robots can detected when they are picked up and when they are put back on the ground, because they can then stop moving if in the air. This information is provided by the GroundContactDetector in the representation *GroundContactState*. The new GroundContactDetector uses the FSR sensors under the feet and the gyroscopes of the NAO to provide this information. It uses a simple state machine with only two states:

1. If the robot has had ground contact so far, it assumes that it has been picked up if the overall pressure on both feet has been below a threshold for at least 100 ms.

2. If the robot has had no ground contact so far, it assumes that it has regained ground contact if the overall pressure on both feet has been above a threshold, the individual pressure on each foot has been above another threshold, the measurements (rotational speeds of the torso) of the forward and sideways gyroscopes have been below a third threshold, and all of this for consecutive 300 ms. Thereby, it is ensured that the robot is standing on both feet and not shaking when regaining ground contact.

7.2 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 *JointAngles*, the calculation of each limb is ensured by the consecutive computations of the kinematic chains. Similar to the inverse kinematic (cf. Sect.8.3.4), 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 (7.1) 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.

$$\vec{r}_{com} = \frac{\sum_{i=1}^{n} \vec{r}_i m_i}{\sum_{i=1}^{n} m_i} \tag{7.1}$$

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 SoftBank Robotics.

7.3 Inertia Sensor Data Filtering

The InertialDataFilter module determines the orientation of the robot's torso relative to the ground. Therefore, the calibrated IMU sensor readings (*InertialSensorData*) and the measured stance of the robot (*RobotModel*) are processed using an Unscented Kalman filter (UKF) [9].

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. Because of noisy measurements by the gyroscopes the prediction has to be corrected. This is achieved by accelrometer measurements of the gravity vector. To avoid confusion in existing modules the orientation is calculated with and without the rotation around the z-axis. The resulting orientations are provided in the representation *InertialData*.

7.4 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*.

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. 7.3). The estimated rotation is represented as rotation matrix R.

$$f_l = -R \cdot t_l \tag{7.2}$$

$$f_r = -R \cdot t_r \tag{7.3}$$

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

$$s = f_r - f_l \tag{7.4}$$

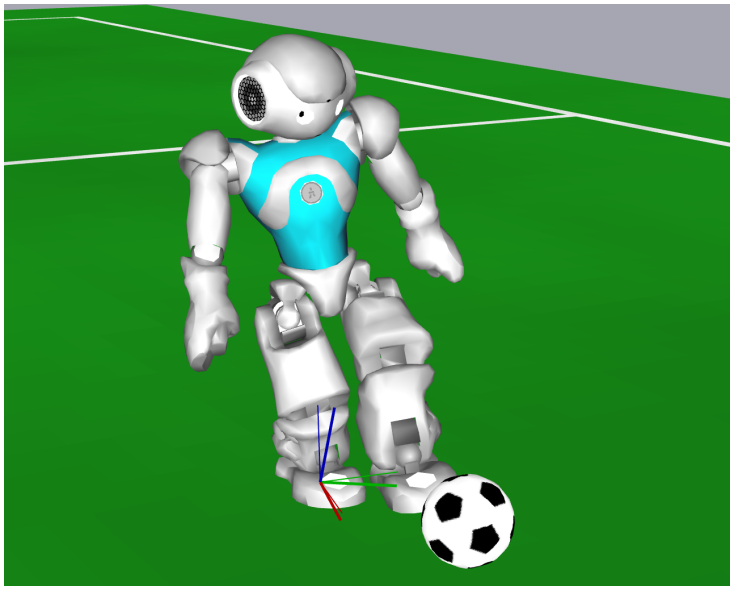


Figure 7.2: The coordinate system of the support foot (thin) and the estimated coordinate system of the field plane (thick).

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.

$$p_{im} = \begin{cases} s/2 + f_l & \text{if } (f_l)_z > (f_r)_z \\ -s/2 + f_r & \text{otherwise} \end{cases} \tag{7.5}$$

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.

7.5 Detecting a Fall

It is important to detect when a robot falls, when it reaches the ground, and in which direction it has fallen on the ground. Fall detection has two phases. The first one is to determine that the robot is currently falling down. The second one is to detect that the robot has reached the ground. It is important to prepare for the ground impact while falling. In case of our team, the head's yaw joint is turned back to its center position and the pitch joint is turned away from the fall direction. After that, the stiffness of all joints is significantly lowered. After the robot has hit the ground, a getup motion is started. So far, both, falling and hitting the ground, were simply detected through thresholds of the torso's orientation in yaw and pitch directions. This leads to several special cases for motions that changed the torso's orientation intentionally. The main innovation of our new FallDownStateDetector is not to use the torso's orientation anymore, but the orientation of the support foot relative to the ground (cf. Fig. 7.2). The support foot is the foot that is extended more in the direction of gravity than the other foot. This allows detecting falling rather early, independent from the motion that is currently executed. The orientation of the torso is only used later, when a fall has already been detected, to determine the direction of the fall and while getting up again. Returning to the upright state can also be determined without dedicated feedback from the getup procedure by simply observing how

far the torso is above the ground, i. e. the length of the support leg in the direction of gravity. However, being upright is not always the intended posture of the robot. Sometimes, the robot is squatting instead. Therefore, this is also a state the FallDownStateDetector recognizes.

7.6 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, at least if the robot is a V4 model. This, in turn, affects the self-localization (cf. Sect. 5.1).

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 100 times per second. The difference of the intended and the actual position of the shoulder joint per frame is calculated and also buffered over several frames. This allows to calculate an average error. 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. In addition, 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[1] earlier as well as the newest measured position.

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. to 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:

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

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

	x	y
is positive	N	E
is negative	S	W

Table 7.1: Converting signed error values into compass directions for the right shoulder and with $-y$ for the left

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 7.1 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 above rules if **both** components are greater than their thresholds, e. g. $x < 0 \wedge y < 0$ results into direction SW. The push direction of each arm is used to add an obstacle to the robot's obstacle model, causing the robot to perform an evasion movement when it hits an obstacle with one of its arms.

Arm contacts are also used to move the arm out of the way to avoid further interference with the obstacle (cf. Sect. 8.9). Please note that while arm motions from the ArmMotionEngine are active, no arm contacts are detected for that arm.

Chapter 8

Motion Control

The B-Human motion control system provides and controls the motions needed to play soccer with a robot. They are six different types of motions that can be requested: *walk*, *stand*, *kick*, *fall*, *getUp* and *specialAction*. These motions are generated by the corresponding motion engines. Additionally, there are specific engines for the head motions (cf. Sect. 8.8) and arm motions (cf. Sect. 8.8). The *walk* and *stand* motions are dynamically generated by the WalkingEngine (cf. Sect. 8.3). Moreover, this module provides a special kind of kicks, the in-walk kicks (cf. Sect. 8.3.3). All the other kicks are generated by the KickEngine [16] that models kicks by using Bézier splines and inverse kinematics. When a robot is unintentionally falling the FallEngine prevents damage (cf. Sect. 8.4). A robot which is laying on the floor can get back to stand by using the GetUpEngine (cf. Sect. 8.6). While doing so, the it takes advantage of *specialAction* motions. The module SpecialActions takes a sequence of predefined joint angles as an input and creates a motion on that basis. It is used wherever an own engine would be too expensive.

Each motion engine generates joint angles. Depending on the selected motions (cf. Sect. 8.1) for the arms and legs, these angles are combined into the *JointRequest* at the end of the process *Motion* (cf. Sect. 8.2). Figure 8.1 shows all motion control modules and the representations they provide.

8.1 Motion Selection

According to the representation *MotionRequest* the MotionSelector determines which motion is executed and calculates interpolation ratios in order to switch between them. In doing so this module takes into account which motion is demanded and which motion is currently executed, and whether those motions can be interrupted. The interruptibility of a motion is handled by the corresponding motion engine in order to ensure the motion is not interrupted in an unstable state.

An exception to this behavior is the *fall* motion which activates itself when needed, to achieve a reflex reaction. Thereby it ignores the interruptibility because if the robot is falling it is unstable by default.

The information about the chosen motion is then provided in the representations *LegMotionSelection* and *ArmMotionSelection*. At First the *LegMotionSelection* needs to be chosen. After that, *ArmMotionSelection* can be filled. In the most cases these two representations do not differ from each other. That means the engine which provides the *LegMotionRequest*, also provides the *ArmMotionRequest*.

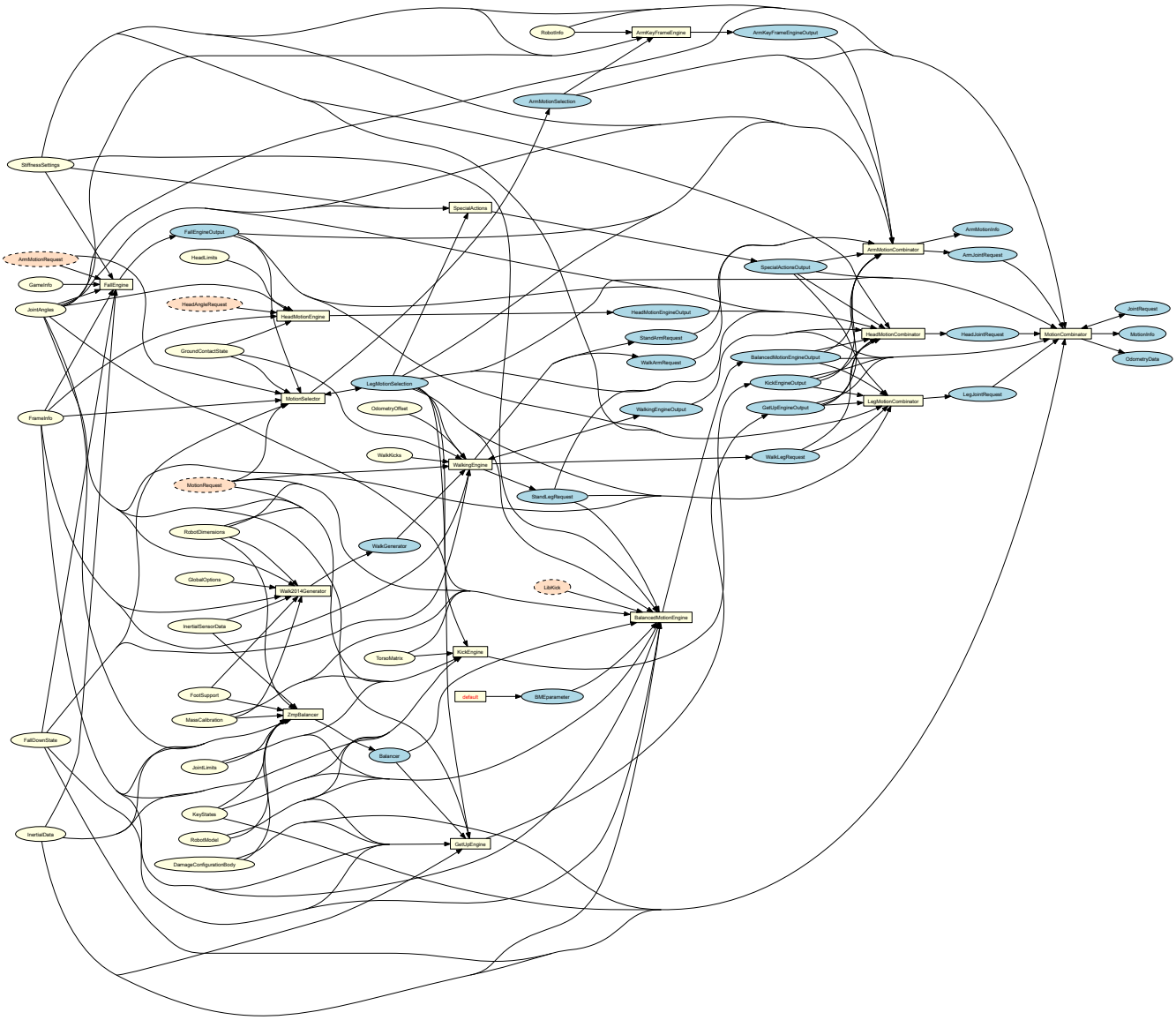


Figure 8.1: Motion control module graph. Motion control modules are depicted as yellow rectangles. All representations they provide are shown as blue ellipses. The representations they require are shown as ellipses colored in either yellow if they are provided by other *Motion* modules and orange if they are received from the process *Cognition*.

8.2 Motion Combination

Motion combination is the two-layered task of merging the generated joint angles of all active motions into one. On the upper layer the MotionCombinator takes the three partial *JointRequests* for head, arms as well as legs and puts them into a full *JointRequest*. On the lower layer each partial *JointRequest* is put together from different motions according to the ratios provided by the MotionSelector. This is done in three separate subtasks to enable the possibility to for example use the information about the upcoming arm motion to adapt the leg motion.

The HeadMotionCombinator takes the HeadYaw and HeadPitch joint angles from the HeadMotionEngine by default. These are overwritten by the selected target motion if the regarding joint angles are not set to be ignored. Afterwards the chosen joint angles are

interpolated and stored in the *HeadJointRequest*.

The ArmMotionCombinator merges the joint angles for the arms, by taking into account the outputs of all modules producing such angles. Its output is the *ArmJointRequest* which contains those angles and the *ArmMotionInfo* describing the executed motions per arm.

The LegMotionCombinator is responsible for merging the joint angles for the legs and outputs them in form of the *LegJointRequest*.

The MotionCombinator is executed at the end of the motion cycle. In Addition to the things mentioned above, it fills the *MotionInfo* ,which contains the actual executed motion, and the *OdometryData*.

8.3 Walking

The WalkingEngine provides the joint angles for walking and standing. The module coordinates the actual generation of the walk pattern with the execution of in-walk kicks (cf. Sect. 8.3.3) and it combines the odometry computed from the request motion with the odometry computed from measured motion.

The major change from last year's walking engine is that the walk patterns are now generated by a different module. In 2016, B-Human participated in the Outdoor Competition. Although we still became the runner-up, it was quite obvious that the walk that we have used since 2012 is not well-suited for artificial grass. Another walk that was very successful in recent years is the one developed by Bernhard Hengst [7] for the team UNSW Australia/rUNSWift. It won the RoboCup 2014 and 2015 and reached the final in 2016 as part of UT Austin Villa's system. It also appeared to work quite well during the Outdoor Competition.

8.3.1 Walk2014Generator

The *Walk2014*, as the walk is called, is based on three major ideas:

1. As in all walks, the body swings from left to right and back during walking to shift away the weight from the swing foot, allowing it to be lifted above the ground. In contrast to many other walks, this is achieved without any roll movements in the leg's joints (unless walking sideways), i. e. the swing foot is just lifted from the ground and set back down again, which is enough to keep the torso in a pendulum-like swinging motion.

2. As a result, there is a clearly measurable event, when the weight of the robot is transferred from the support foot to the swing foot, which then becomes the new support foot. This event is detected by the pressure sensors under the feet of the robot. In the moment this weight shift is detected, the previous step is finished and the new step begins. This means that a transition between a step and its successor is not based on a model, but on a measurement, which makes the walk quite robust to external disturbances.

3. During each step, the body is balanced in forward direction by the support foot. The approach is very simple but effective: the lowpass-filtered measurements of the pitch gyroscope are scaled by a factor and directly added to the pitch ankle joint of the support foot. As a result, the faster the torso turns forward, the more the support foot presses against this motion.

8.3.2 Improvements

In contrast to some other teams that integrated the Walk2014, we only use parts of its core implementation from the class `Walk2014Generator`. We have removed everything not related to walking from that class, refactored the code, converted it into a B-Human module, integrated B-Human data types, and integrated our in-walk kicks. We also replaced the inverse kinematics by our own, which results in a different rotational behavior. In the original implementation, the inverse kinematics is first solved for the legs ignoring any yaw rotation of the feet. Then, the yaw rotation is added as a rotation around the hip yaw-pitch joint (which is not entirely correct). Afterwards, an iterative algorithm is used to return the feet to the positions they had before the rotation was added. Our inverse kinematics simply computes the joint angles for the 6-D positions of both feet, equally distributing the error introduced by the linked hip yaw-pitch joint to both feet. The computation is not only more precise, it is also a lot faster, as no iteration is involved. We now also use linear trajectories for forward, sideways, and rotational motions of the support foot.

One of the core ideas of the Walk2014 is that a step ends when the weight transfer happens and the next step starts from where the last step ended. The latter was only fully implemented in the Walk2014 for forward and rotational movement, but not for sideways movement and the height of the swing foot, which may not have been set down completely before the weight has already shifted. We implemented this, which results in less disturbances in situations in which a step did not behave as expected anyway. We also implemented a mode to reach a certain relative position on the field, always considering which motion is unavoidable, since both legs always have to swing back to their zero position before the robot can come to a full stop.

We use different gyroscope balancing parameters for forward and backward rotation, because the feet are shorter at the back and stability benefits from slightly more aggressive balancing in that direction. We also found that a lot of differences between different robots can be compensated by varying the balancing parameters. Typically, older robots need more aggressive balancing. We also integrated gyro-based sideways balancing, but it is only active when the robot is standing. It dampens the swinging from one foot to the other in situations where the stand was activated while the robot was heavily swinging from one side to the other.

The Walk2014 assumes a fixed position of the center of mass in the torso. Since our robots sometimes take their arms behind their back to better avoid obstacles, this assumption is not valid for them. Therefore, we introduced a dynamic center of mass computation into the walk, which considers the impact of the arms' positions. Iteratively, the torso is tilted in a way that the center of mass's projection to the ground keeps at the same forward position relative to the feet as the center of mass with regular arm positions would have.

We also changed the acceleration of the robot to a linear model. In addition, if the weight transition between the support leg and the swing leg is not recognized, the current speed is reset to zero, resulting in the robot accelerating again in a controlled manner afterwards. We also detect if the weight transition is not recognized repeatedly, which can happen if the robot leans against a static object, e. g. a goalpost. In that case, it starts moving sideways away from the obstacle, which frees it from the deadlock. We also recognized that robots sometimes get stuck in a lower swing frequency. This is also detected and ended through a short stand phase.

8.3.3 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 modules are likely to let the robot stumble or fall over, the WalkingEngine is able to carry out simple kicks while walking.

At the moment, there are three kick types: `forward`, `sidewardsInner`, and `turnOut`. Those kicks are described individually by the config files located in *Config/WalkKicks*. Every kick is defined by its duration, step sizes before and during the kick, and key frames for the 3D position and rotation of the kicking foot. These key frames are later interpolated by cubic splines to describe the actual trajectory of the foot, by overlaying the original trajectory of the swinging foot and thereby describing the actual kicking motion. In doing so the overlaying trajectory starts and stops at 0 in all dimensions both 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.

8.3.4 Inverse Kinematic

The inverse kinematics for the ankles of the NAO are a central component of the module WalkingEngine. 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.

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. 8.2.

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]

$$Foot2Hip = Trans_y\left(\frac{l_{dist}}{2}\right) \cdot Foot2Torso \tag{8.1}$$

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.

$$Foot2HipOrthogonal = Rot_x(\frac{\pi}{4}) \cdot Foot2Hip \tag{8.2}$$

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

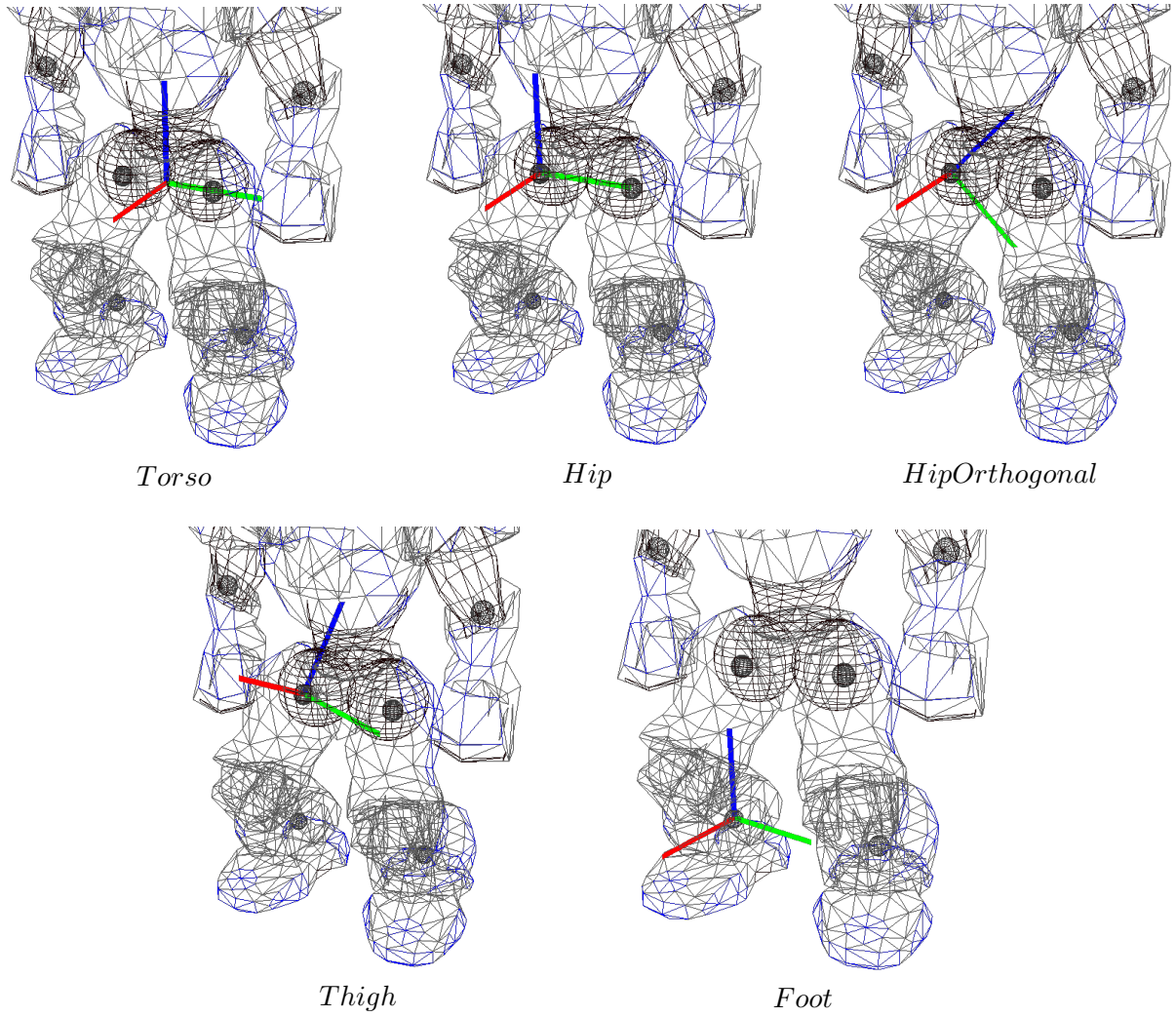


Figure 8.2: Visualization of coordinate frames used in the inverse kinematic. Red = x-axis, green = y-axis, blue = z-axis.

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.

$$HipOrthogonal2Foot = Foot2HipOrthogonal^{-1} \tag{8.3}$$

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 (8.4). Knowing that the angle enclosed by the limbs corresponds to the knee joint, that joint angle is computed by equation (8.5).

$$c^2 = a^2 + b^2 - 2 \cdot a \cdot b \cdot \cos\gamma \tag{8.4}$$

$$\gamma = \arccos \frac{{l_{upperLeg}}^2 + {l_{lowerLeg}}^2 - {l_{trans}}^2}{2 \cdot l_{upperLeg} \cdot l_{lowerLeg}} \tag{8.5}$$

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

$$\delta_{knee} = \pi - \gamma \tag{8.6}$$

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

$$\delta_{footPitch1} = \arccos \frac{{l_{lowerLeg}}^2 + {l_{trans}}^2 - {l_{upperLeg}}^2}{2 \cdot l_{lowerLeg} \cdot l_{trans}} \tag{8.7}$$

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]

$$\delta_{footPitch2} = \text{atan2}(x, \sqrt{y^2 + z^2}) \tag{8.8}$$

$$\delta_{footRoll} = \text{atan2}(y, z) \tag{8.9}$$

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.

$$\delta_{footPitch} = \delta_{footPitch1} + \delta_{footPitch2} \tag{8.10}$$

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

$$HipOrthogonal2Thigh = Thigh2Foot^{-1} \cdot HipOrthogonal2Foot \tag{8.11}$$

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

$$Thigh2Foot = Rot_x(\delta_{footRoll}) \cdot Rot_y(\delta_{footPitch}) \cdot Trans_z(l_{lowerLeg}) \cdot Rot_y(\delta_{knee}) \cdot Trans_z(l_{upperLeg}) \tag{8.12}$$

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 $\cos \delta_x$).

$$Rot_{Hip} = Rot_z(\delta_z) \cdot Rot_x(\delta_x) \cdot Rot_y(\delta_y) = \begin{pmatrix} c_y c_z - s_x s_y s_z & -c_x s_z & c_z s_y + c_y s_x s_z \\ c_z s_x s_y + c_y s_z & c_x c_z & -c_y c_z s_x + s_y s_z \\ -c_x s_y & s_x & c_x c_y \end{pmatrix} \tag{8.13}$$

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

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

$$\frac{-r_{01}}{r_{11}} = \frac{\cos\delta_x \cdot \sin\delta_z}{\cos\delta_x \cdot \cos\delta_z} = \frac{\sin\delta_z}{\cos\delta_z} = \tan\delta_z \tag{8.14}$$

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

$$\delta_z = \delta_{hipYaw} = \text{atan2}(-r_{01}, r_{11}) \tag{8.15}$$

$$\delta_y = \delta_{hipPitch} = \text{atan2}(-r_{20}, r_{22}) \tag{8.16}$$

At last the rotation by 45 degrees (cf. eq. 8.2) has to be compensated in joint space.

$$\delta_{hipRoll} = \delta_x - \frac{\pi}{4} \tag{8.17}$$

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.

8.4 Falling

When a robot is unintentionally falling to the ground there is a high risk that the robot breaks. Therefore falls are detected (cf. Sect. 7.5), so that damage can be prevented by executing certain motions and softening some limbs. Depending on whether the robot is falling to the front or the back, different motions are executed. If the robot falls backwards a squatting position is adopted to decrease the drop height of the head. Additionally, the arms are brought back to a neutral position, if necessary, and the head is tilted to the front. If the robot falls forward the *stand* motion is called and the head is placed in the neck. In this case the squatting position is not adopted because this would produce an unfavorable angle between the head and the

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

floor when hitting it. Once the FallEngine has been activated, it is left if a *getUp* motion is demanded or the robot has been put up manually. To avoid a false activation of the FallEngine, certain situations are excluded: For example the GetUpEngine cannot trigger the FallEngine and *specialAction* motions are entirely ignored if no game controller packages are received.

8.5 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 *Config/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, wrist, hand), right arm (shoulder pitch/roll, elbow yaw/roll, wrist, hand), 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 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 stiffness 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 stiffness as default. This is done by a line starting with the keyword *stiffness*, followed by a value between 0 and 100 for each joint (in the same order as for specifying actual joint angles). In the file *Config/stiffnessSettings.cfg* default values are specified. If only the stiffness of certain joints should be changed, the others can be set to '*'. This will cause those joints to use the default stiffness. At the end of one stiffness line the time has to be specified that it will take to reach the new stiffness values. This interpolation time runs in parallel to the interpolation time between target joint angles. In addition, the stiffness values defined will not be reached if another stiffness command is executed 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
```

[5] Ignored

```
label start
stiffness 20 20 60 60 60 60 60 60 60 60 60 60 60 90 90 90 90 90 90 90 90 90 90 90 90 50
"HP HT AL0 AL1 AL2 AL3 AL4 AL5 AR0 AR1 AR2 AR3 AR4 AR5 LL0 LL1 LL2 LL3 LL4 LL5 LR0 LR1 LR2 LR3 LR4 LR5 Int Dur
  *  *   0 -50  -2 -40   0   0   0 -50  -2 -40   0   0  -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.

8.6 Get Up Motion

To be able to recover after a robot fell down, there is an engine based on speciel actions. GetUp motions are modelled with piecewise linear trajectories for every joint and can be converted to special action files.

Additionally, 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 be able to retry from the beginning. Since the introduction of artificial grass, GetUp motions tend to fail more often and the success depends on the hardware condition of the robot and its situation, e. g. position on the field and other robots nearby. To increase the probability of success without loosing the chance to stand up very fast, there is a list of different motions for every robot. Normally, we start with fast motions and increase reliability by decreasing speed for the next trials. After the second trial, the robot waits a moment and then takes the start position very slow. We observed that referees often react too late to a pushing robot inhibiting our robots to get back on their feet. Waiting on the ground increases the probability of a safe GetUp motion without collisions.

If the last attempt fails, a hardware defect is assumed causing the engine to set the stiffness of all joints to zero. Thus, the robot remains lying down in order to prevent any more damage. Furthermore, a cry for help is initiated causing the robot to play the sound "help me".

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 stand up movement or manually by holding the robot in an upright position as a referee would do following the fallen robot rules.

On the new ground, our robots tend to stagger more often while getting up, so a PID controller was not sufficient anymore. To handle this problem in general, we developed an universal balancing module, which is parameterized and can be adapted to different tasks (see Sect. 8.7).

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 torso rotation of the representation *InertialData* 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 case that the robot has not fallen at all, the engine terminates immediately, i. e. flags itself as "leaving is possible" and initiates a stand motion. Furthermore, after each unsuccessful stand up attempt, the engine reviews the torso rotation in case an interfering robot has changed it.

Thus, if a fall is detected, one only needs 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).

8.6.1 Modify Get Up Motions

After a couple of years, we finally needed to adjust our stand-up motions, which did not feel very comfortable. For that reason, we tried to improve that process by using special actions (cf. Sect. 8.5) during creation and convert them afterwards into the file *getUpEngine.cfg*. In doing so, we first included *getUpEngineDummy.mof* as a dummy file that is later used to edit and execute the stand-up motion.

To load the data from an existing motion into the dummy file we use `set module:GetUpEngine:generateMofOf nameOfMotion;` in the simulator. This checks for the motion name in the file *getUpEngine.cfg* and converts the data of that specific stand-up motion from the config-file into the file *getUpEngineDummy.mof*. Afterwards, it can be written back to the file *getUpEngine.cfg* using `set module:GetUpEngine:generateMotionFromDummyMof nameOfMotion;`. Once you generated the file *getUpEngineDummy.mof*, it can be used like a normal special action. Of course this includes the simulator command `mof` that compiles and copies special actions from your hard drive to the robot without a restart.

As all conversions are done on file level, it is advisable to always convert in offline mode. Otherwise, in online mode, you have to look up the files on the robot's memory drive instead of your own harddrive.

Since GetUpEngine motions are a bit different from normal special actions, we included some special comments. These include critical phases (`"@critical`), the phase the balance mechanism should be activated (`"@balanceStartLine`), and how the odometry changes after the whole motion is done (`"@odometryOffset x y rotation`). These special comments are used if the motion is converted back into the file *getUpEngine.cfg* and are implemented by the GetUpEngine. If the motion is executed by the module SpecialActions, the special comments are ignored. Thus, balancing and upright position verification are not enabled.

At last, the GetUpEngine is now able to change the joint stiffness during execution using the `stiffness` command. As special actions and the GetUpEngine are different from each other, there is a compatibility issue regarding the stiffness. In the GetUpEngine the stiffness is applied at the start of each interpolation process in a single joints key-frame. In the module SpecialActions the stiffness is interpolated simultaneously to the joints key-frame interpolations. Thus, the stiffness interpolation time can differ from the joints interpolation time and take more than a single joints key-frame. Consequently, comparable behaviors of both engines are only reached if the interpolation time of stiffness key-frames is set to 0.

The following example shows how to use the special comments if a motion is created as special action:

```
motion_id = getUpEngineDummy
label start
stiffness 20 20 60 60 60 60 60 60 60 60 60 60 60 60 90 90 90 90 90 90 90 90 90 90 90 90 0
0 0 99 74 -5 -80 -90 0 99 -74 5 80 90 0 0 0 0 0 60 0 0 0 0 0 60 0 1 500

stiffness 30 30 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 0
0 0 99 74 -5 -80 -90 0 99 -74 5 80 90 0 0 0 0 0 60 0 0 0 0 0 60 0 1 500

"@critical (upright check at start of interpolation to next line)
0 0 99 74 -5 -80 -90 0 99 -74 5 80 90 0 0 0 0 0 60 0 0 0 0 0 60 0 1 500
```

```
"@balanceStartLine (turn on balance at start of interpolation to next line)
0 0 99 74 -5 -80 -90 0 99 -74 5 80 90 0 0 0 0 0 60 0 0 0 0 0 60 0 1 500

label repeat
0 0 99 74 -5 -80 -90 0 99 -74 5 80 90 0 0 0 0 0 60 0 0 0 0 0 60 0 1 500

"@odometryOffset 230.00 -40.00 -45.00

transition getUpEngineDummy getUpEngineDummy repeat
transition allMotions extern start
```

8.7 ZMP Balancing

The balancing module contains a lot of different approaches which can be individually adjusted and switched on and off using a parameter set.

The robot is modelled as an inverted pendulum trying to hold the zero moment point (short: ZMP, center of mass under consideration of dynamic effects) dynamically above the support polygon of one foot. It assumes that this foot is always on the ground. Since it cannot change its position, all values are calculated in relation to this foot. The module can also be used to balance on two legs by deactivating the preview controller to ignore the dynamics of the inverted pendulum. In this case the future ZMP values are ignored and only the next one is used as the center of mass.

With every call, the same balancing methods are used:
Using `calcBalancedJoints` a new *jointRequest* is calculated. A swing leg pose and a trajectory of future ZMP values are specified in relation to a support foot.
`addBalance` modifies a *jointRequest* as little as possible to keep a specified ZMP trajectory in relation to the support foot.
The function `balanceJointRequest` stabilizes a *jointRequest*. The robot is prevented from tilting using the functions above with the center of mass of the requested *jointRequest* as the ZMP.

There are two different ways of initialization. The function `init` only resets all intern states and calculates the current state of the inverse pendulum. `initWithTarget` can be used to reach a stable position before executing other motions. It will calculate a ZMP trajectory which overrides the ZMP values until a stable position is reached. In this case `calcBalancedJoints`, `addBalance` and `balanceJointRequest` will return `true`.

Using the IMU, the module calculates the current tilt, which is often caused by wobbly ground. Tilting can be prevented using the arms, but due to oscillation the movements have to be damped. It is very important that the support foot remains flat on the ground to retain grip. To do so, the ankle is turned depending on the tilt. Balancing on one leg is often not possible due to weak knee or ankle joints. When there is a high deviation between requested and measured joint angles, it can be counteracted and balanced using the hip joint. To ensure dynamic stability during fast movements, a preview controller is used based on the cart-table model.

At the time of publication, the module is not relibale enough to balance slightly damaged robots on one leg during fast motions in a real game on artifical grass. The main problem is to quickly reach a stable one-legged start position. This is hard to do mainly because of two reasons: During a game, a robot is always moving and because of that it is very difficult to compute the current fluctuation. The other reason is that joint conditions differ a lot between robots, so it is hard to calculate the power needed to reach a one-legged position without falling over.

8.8 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 KickEngine 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 themselves with this.

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

8.9 Arm Motions

In addition to the arm motions that are calculated combined with the leg motions (e. g. during a special action or a kick) we also use a motion engine that provides only output for the arm joints: The ArmKeyFrameEngine will only be selected during a walking or standing (leg) motion to follow key frames.

Although we are mainly using this feature to move the arms behind the robot's back to prevent (further) contact with other robots, it generally supports any desired arm key frame motion. A motion is defined by a set of states, which consist of all target angles of an arm (similar to a special action, cf. Sect. 8.5). Upon executing an arm motion, the motion engine interpolates intermediate angles between two states to provide a smooth motion. When reaching the final state, the arm remains there until the engine gets a new request. The ArmKeyFrameEngine performs all the calculations and interpolations for the desired arm motion.

Arm motions are configured in the file *armKeyFrameEngine.cfg*. It 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 motions triggered by arm contact).

allMotions: Array of different arm motions. The entry with the id `useDefault` is a native motion, which should neither be modified nor should it get another index within the array.

Despite the available motions, you may add further ones to the file following the same structure under a new *id*. The new *id* has to be appended to the end of the enumeration `ArmKeyFrameId` within the representation *ArmKeyFrameRequest*.

```
// [..] config entries left out
allMotions = [
  {
    // [..] native motions left out
  },{
    id = sampleArmMotion;
    states = [
      {angles=[-2.05, 0.2, 0.174, -0.2, -90deg, 0deg]; stiffness = [90, 60,
          80, 90, 90, 90]; steps = 80; },
```

```
        {angles=[-2.11, -0.33, 0.4, -0.32, -90deg, 1]; stiffness = [90, 60, 80,
            90, 90, 90]; steps = 80; }
      ];
    }
  ];
```

Listing 8.1: Example of an arm motion definition

It is important to know that all motions will only be declared for the left arm. Inside the ArmKeyFrameEngine, the motion will then be correctly mirrored for the right arm. The angles and stiffness values of each state correspond in their order of occurrence to: `ShoulderPitch`, `ShoulderRoll`, `ElbowYaw`, `ElbowRoll`, `WristYaw`, and `Hand`. For each stiffness entry, the special value -1 may be used, which indicates the use of the default stiffness value for that joint as specified in the file *stiffnessSettings.cfg*. The attribute `step` specifies how many *Motion* frames should be used to interpolate intermediate angles between two entries in the array `states`.

Arm motions can be triggered by filling out the representation *ArmKeyFrameRequest*. This is done by the behavior output options *KeyFrameArms*, *KeyFrameLeftArm*, or *KeyFrameRightArm*. The first will start the motion selected for both arms. The representation mentioned has two 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, `reverse` means that you request the reverse motion of the last executed (key frame) motion.

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

Chapter 9

Technical Challenge and Mixed-Team Competition

In addition to the main soccer competition, B-Human also participated in the *Penalty Shot Challenge* as well as in the *Mixed Team Competition*. For these competitions, all standard modules that have been described in the previous chapters have been used and only a few minor adaptions have been necessary.

9.1 Penalty Shot Challenge

As penalty shots are also a part of the normal game rules, the standard behavior, which is described in detail in 6.3, has been active for the penalty taker as well as for the penalty keeper.

In 2016, we already had successful penalty behaviors that enabled us to win the penalty shootout of the RoboCup 2016 final with a 3:0 (after a 0:0 in the normal game). The penalty taker reliably scored with all three shots and the penalty keeper was able to save one shot (the other two were failed attempts by the opponent penalty taker). However, the technical challenge was a strong incentive to further improve these behaviors.

The focus of the penalty taker improvements has been on reliability. The aim is to reproducibly shoot as hard as possible to a position as close as possible next to one of the two goal posts. As a result, our penalty taker was able to successfully score at each attempt during the *Penalty Shot Challenge*, scoring 8 goals in 4 games. Figure 9.1 shows one of these attempts. We did not make any efforts to minimize the time between the start of an attempt and the kick, although this is a criterion in case of a draw.

For the penalty keeper, improving its reactivity was most important, as not much time remains after a ball has been kicked from the penalty spot, which is quite close to the goal. Thus, a kick as well as its direction have to be detected as early as possible, avoiding any false positives. In addition, the final blocking pose must be reached with high speed. Our keeper is currently fast and reliable enough to block all shots that are kicked with medium speed. One example from the final challenge shootout is shown in Fig. 9.2. Due to its reactivity, our keeper let only two shots pass during the whole challenge. However, it should be noted that our penalty keeper is not (yet) fast enough to catch the shots made by our penalty taker as they are too fast and thus do not leave enough time to reach the final blocking pose.

An overview of the games is given at the league's web site at `http://spl.robocup.org/results-2017/`.

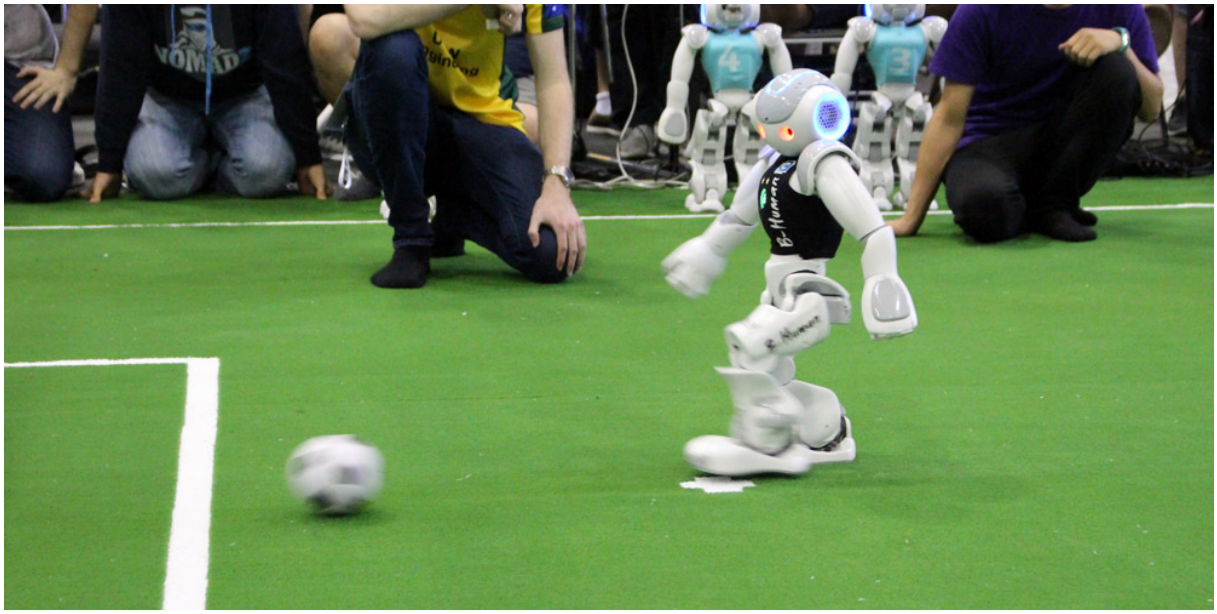


Figure 9.1: B-Human penalty taker in the final of the *Penalty Shot Challenge* against *NomadZ*

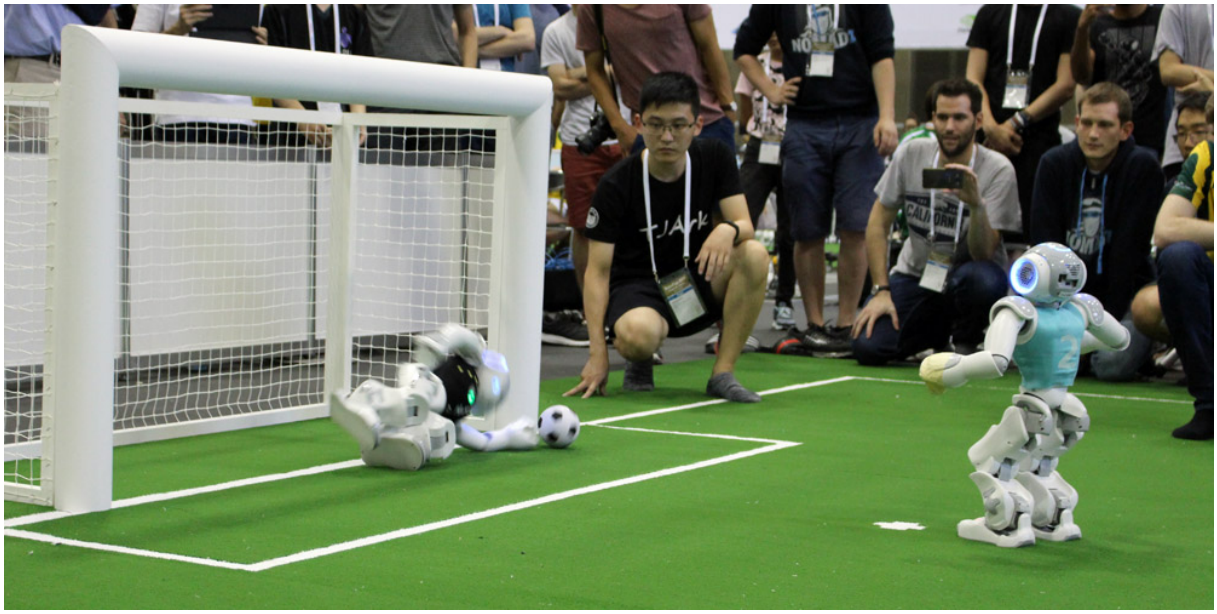


Figure 9.2: B-Human penalty keeper blocking the ball in the final of the *Penalty Shot Challenge* against *NomadZ*

9.2 The *B-HULKs* in the Mixed Team Competition

In 2017, the *Mixed Team Competition* was held for the first time, replacing the *Drop-In Competition*, which was the Standard Platform League's testbed for multi-agent cooperation from 2014 until 2016 and which B-Human won twice. For the first year of this new competition, each mixed team consists of a pair of normal teams. This pairing remains fixed over the whole competition and has to be defined in advance, i. e. together with the teams' application for the main competition.

As we are convinced of the necessity of performing many full system tests under realistic conditions to achieve a high performance in a competition, we were looking for a partner team that we could meet multiple times for testing and coordination. A remote collaboration was considered as too error-prone, as many implementation problems might only manifest during some actual game situations. Given these prerequisites, the *HULKs* have been a perfect partner. Their university is only about one hour away from Bremen. This allowed us to meet, discuss, and test with many robots multiple times. Furthermore, the HULKs have the same strong commitment to the Mixed Team Competition as we have.

Together, we are the *B-HULKs*. Most team members are shown in Fig. 9.3.

As in this competition both members of a mixed team use their own soccer competition code base, only two major issues have to be resolved for playing together as a team: agreeing on a strategy for playing with six robots and specifying a standard communication protocol that extends the rather basic elements of the SPL standard message. The technical details of the B-HULKs' solutions will be presented in a separate report that will be published by both teams together. The extended communication protocol is also used by B-Human in normal games, its integration is described in Sect. 3.5.4.

Furthermore, in addition to any technical issues, we would like to highlight the fact that the *B-HULKs* have been the only team that designed an own logo as well as custom robot jerseys for this competition. Both merge the typical design elements of the two original teams. The jerseys are shown in Fig. 9.4. The team logo is part of Fig. 9.3 as well as of Fig. 9.4.

During the competition at RoboCup 2017, the *B-HULKs* played four games and won all of them, although the final win required an additional penalty shootout. As both teams have robust implementations of all required basic abilities, such as stable walking, ball recognition, and self-localization, all robots on the field were able to play together reliably according to the previously defined strategy. To sum up, one could say that the robots from *B-Human* and the *HULKs* equally contributed to our success in this competition.

An overview of the results is given at the league's web site at `http://spl.robocup.org/results-2017/`.

Figure 9.3: The human team members of the *B-HULKs* that participated in RoboCup 2017 in Nagoya

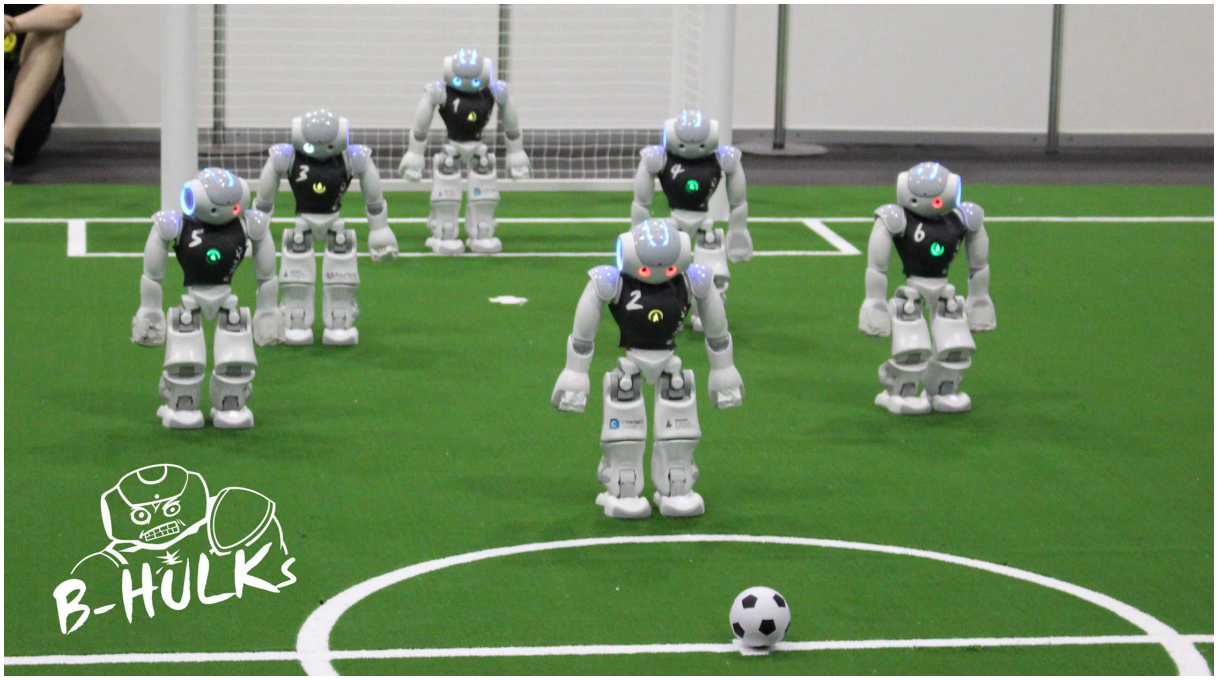


Figure 9.4: The robot team members of the *B-HULKs*

Chapter 10

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* and the tools accompanying it were also developed by B-Human, they are also described at the end of this chapter.

10.1 SimRobot

The B-Human software package uses the physical robotics simulator SimRobot [12, 10] 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 physical robots via Ethernet or WiFi.

10.1.1 Architecture

Three dynamic libraries are created when SimRobot is built. These are *SimulatedNao*, *SimRobotCore2* and *SimRobotEditor* (cf. Fig. 10.1)[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.

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.

10.1.2 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. 10.2).

[1] The actual names of the libraries have platform-dependent prefixes and suffixes, i. e. *.dll*, *.dylib*, and *lib .so*.

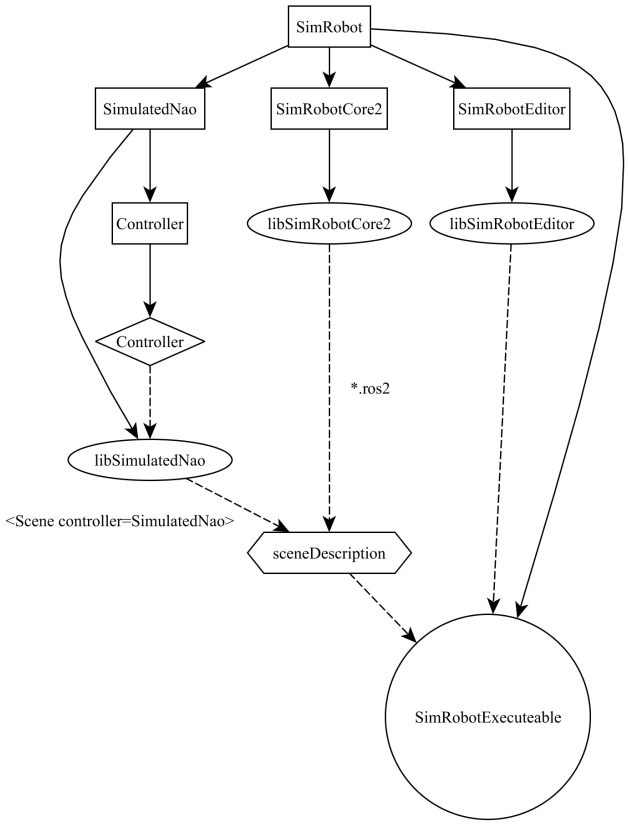


Figure 10.1: This figure shows the most important libraries of *SimRobot* (excluding foreign libraries). The rectangles represent the projects, which create the appropriate library. Dynamic libraries are represented by ellipses and the static one by a diamond. Note: The static library will be linked together with the *SimulatedNao* code. The result is the library *SimulatedNao*.


Lets the robot stand up.


Lets the robot sit down.


Allows moving the robot's head by hand.

10.1.3 Scene View

The scene view (cf. Fig. 10.3 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, zoomed, and it supports several mouse operations:

- Left-clicking an object allows dragging it to another position. Robots and the ball can be moved in that way.

Figure 10.2: This figure shows the three buttons from the *BHToolBar*.

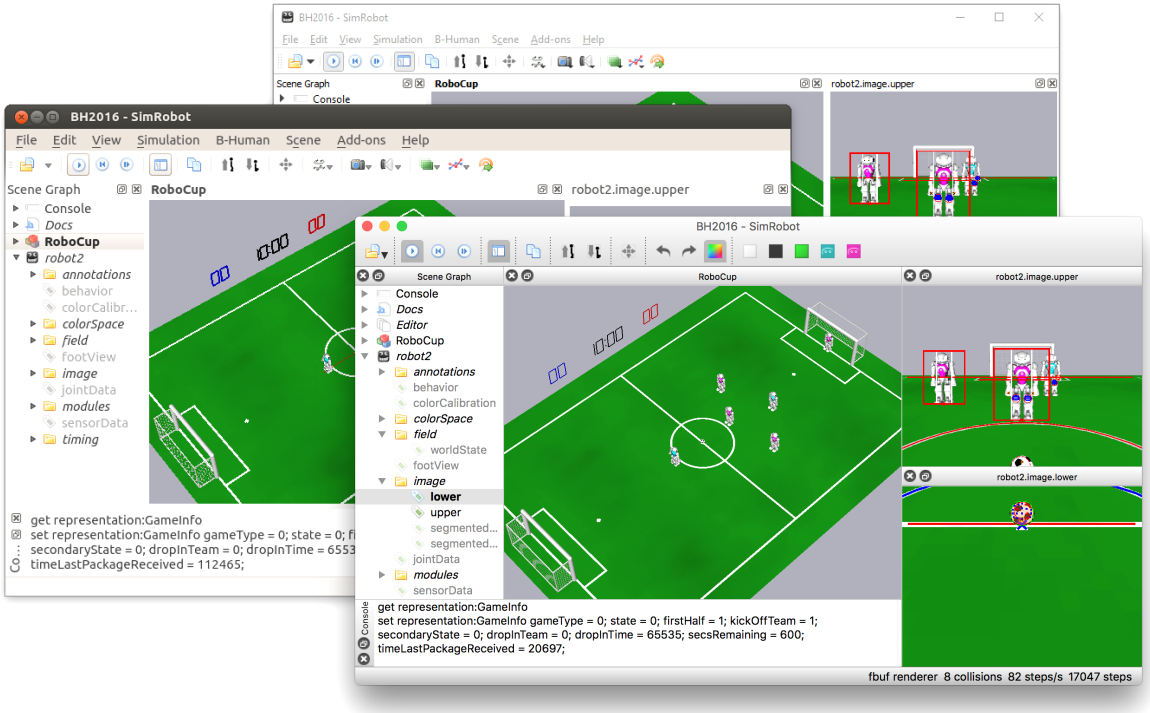


Figure 10.3: SimRobot running on Windows, Linux, and macOS. The left pane shows the scene graph, the center pane shows a scene view, and on the right there are two views showing the images of both cameras. The console window is shown at the bottom.

- Left-clicking while pressing the *Shift* key allows rotating objects around their centers. The axes of the rotating objects can be selected first by pressing *x*, *y* or *z* key, before pressing and holding the *Shift* key.
- 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. 10.1.5). Robot console commands are sent to the selected robot only (see also the command *robot*).

10.1.4 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 fifteen kinds of information views, which are structured here into the five categories *cognition*, *behavior control*, *sensing*, *motion control*, and *general debugging support*. All information views can be selected from the scene graph (cf. Fig. 10.3 left).

10.1.4.1 Cognition

Image Views

An image view (cf. left of Fig. 10.4) displays debug information in the coordinate system of the camera image. It is defined by an debug image, an optional flag for the image, an optional

Figure 10.4: Image view and field view with several debug drawings

switch for JPEG compression, an optional switch for segmentation and a name using the console command *vi*. More formally its syntax is defined as:

```
vi <debug image> [upper] [jpeg] [segmented] [name]
```

The *debug image* can be the regular camera image and any other image that can be sent using the `SEND_DEBUG_IMAGE` macro. The following *debug image* are currently defined:

ColoredImage: A color segmented image. Only contains the relevant colors.

GrayscaledImage: The Y channel of the image.

SaturatedImage: An image only containing the calculated saturation visualized as a grayscale image.

HuedImage: An image only containing the calculated hue visualized as a grayscale image.

CNSImage: A contrast normalized Sobel image which is a Sobel image containing information about the direction. The angles are visualized with different colors (cf. Fig. 4.10b).

image: The image provided by the camera.

imagePatches: Image cutouts smaller than the full image, containing all channels. Only available if a provider for such image patches is active.

corrected: Similar to *image*, but without the rolling shutter effects.

horizonAligned: Similar to *image*, but aligned to the horizon.

none: Displays an empty background.

The default is to show data based on the images taken by the lower camera. With *upper*, the upper camera is selected instead. 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 pixels value itself.

The console command *vid* adds the debug drawings. For instance, the image view with the flag *segmented* is defined as:

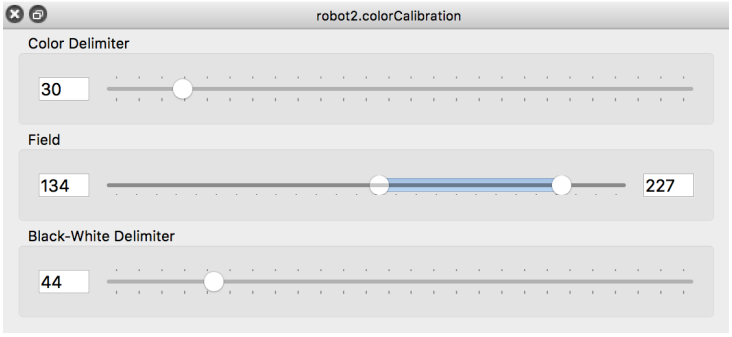


Figure 10.5: An example calibration

```
vi image segmented
vid segmentedLower representation:LinesPercept:image
vid segmentedLower representation:BallPercept:image
vid segmentedLower representation:PlayersPercept:image
```

You can deactivate debug drawings by appending *off* to a *vid* command.

Color Calibration View

The color calibration view provides direct access to the parameter settings for all existing color classes. The parameters of a color class can be changed by moving the sliders. Color classes are defined in the YHS color space as described in Sect. 4.1.4.

The following buttons are added to the toolbar when the *colorCalibration* view is focused.

 saves the local color configuration

 undoes the latest slider change

 redoes a reverted slider change

 black, white, and field color delimiters

 hue value for your own team's color

 hue value for the opponent team's color

Furthermore each button will show the current settings of the corresponding color class in the *colorCalibration* view. Both jersey colors use range selectors for their hue values in the YHS colorspace. Figure 10.5 shows an example calibration for the color delimiter, the field color, and the black/white delimiter. The color delimiter defines the minimum saturation of a pixel necessary to be classified as colored. *Field* defines the hue range for pixels being classified as field color if they are colored. The black-white delimiter defines the minimum brightness for non-colored pixels to be classified as white instead of black.

Snapshot View

The snapshot view helps saving images from the robot's upper and lower cameras. They will be stored in the directory *Config*. The images can either be taken instantly (*Quick Snapshot*) or repeatedly (*Start Logging*) with the chosen number of seconds in between each consecutive snapshot. You can also specify a file name prefix and a starting number, from which the names for new snapshots will be counted up. You can also select whether to take images from the upper, lower, or both cameras.

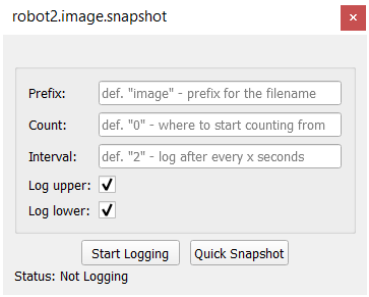


Figure 10.6: The options of the snapshot view

It is important to consider that the counter resets to the given starting value each time *Start Logging* is pressed. Conflicting files will be overwritten.

Color Space Views

Color space views visualize image information in 3-D (cf. Fig. 10.7). 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:

Image Color Channel: This view displays an image while using a certain color channel as height information (cf. Fig. 10.7 left).

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. 10.7 right).

The two kinds of views have to be added manually for the camera image or any debug image. For instance, to add a set of views for the camera image under the name *raw*, the following command has to be executed:

```
v3 image raw
```

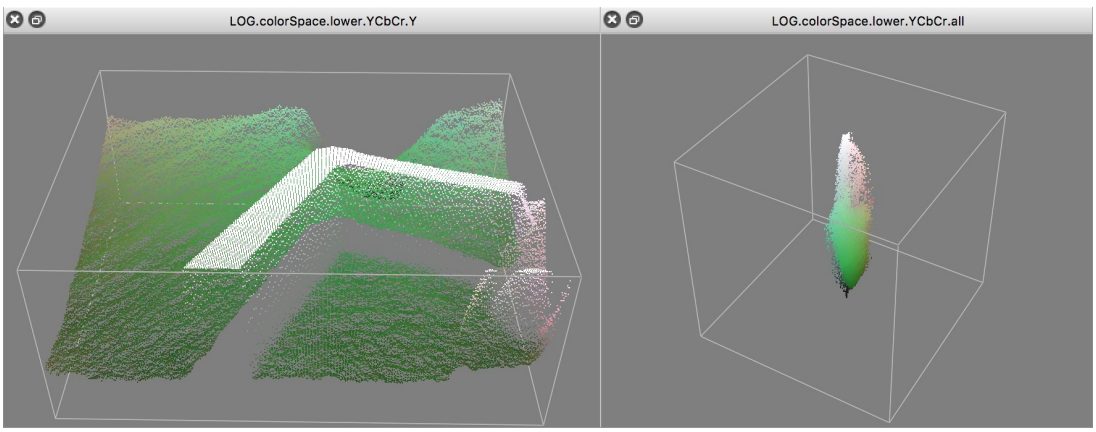


Figure 10.7: Color channel views (left) and image color space view (right)

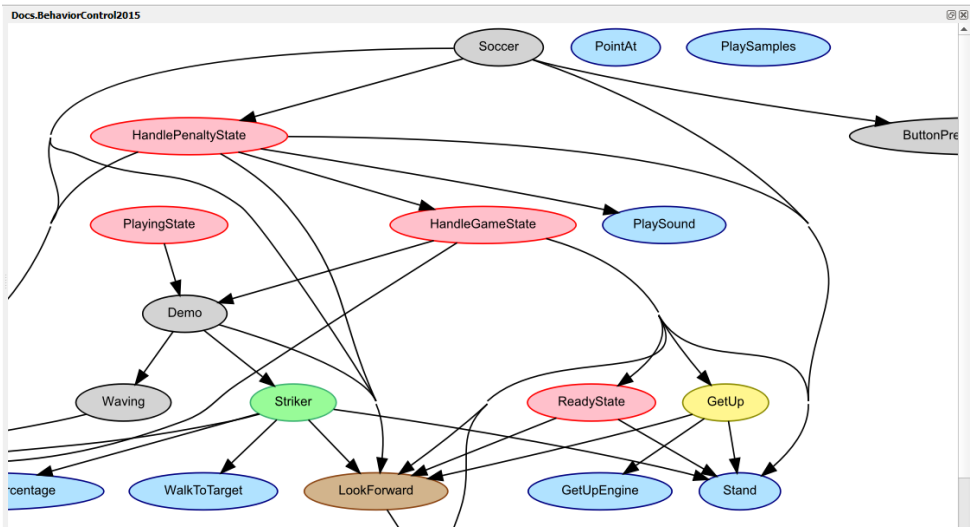


Figure 10.8: Option graph view

Field Views

A field view (cf. right of Fig. 10.4) 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 goalFrame
vfd worldState fieldPolygons
vfd worldState representation:RobotPose
# ...

# views relative to robot
vfd worldState origin:RobotPose
vfd worldState representation:BallModel:endPosition
# ...

# 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*, touch gestures, or the *page up/down* buttons. It can also be dragged around with the left mouse button or by touch gestures. Double clicking the view resets it to its initial position and scale.

10.1.4.2 Behavior Control

Option Graph View

The *option graph view* (cf. Fig. 10.8) 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 *BehaviorControl*. The

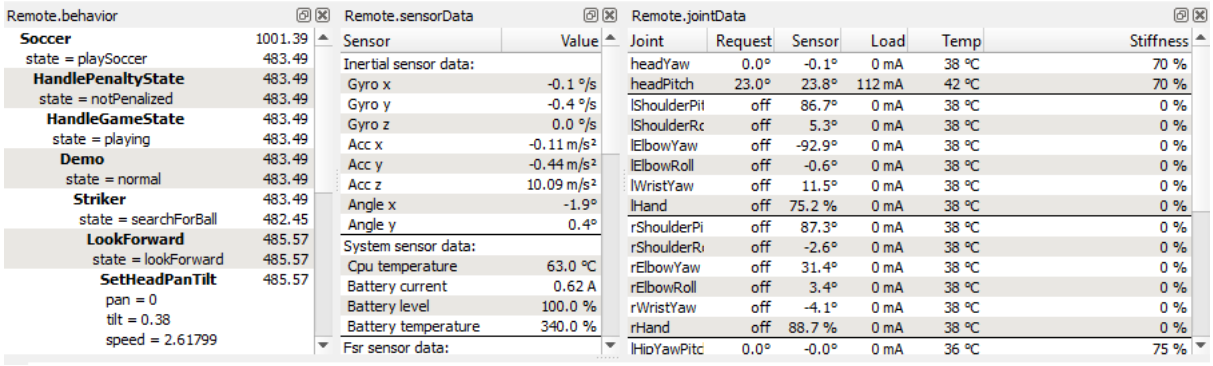


Figure 10.9: Behavior view, sensor data view and joint data view

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.

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*, touch gestures, or the *page up/down* buttons. It can also be dragged around with the left mouse button or by touch gestures.

Behavior View

The *behavior view* (cf. left of Fig. 10.9) shows all the options and states that are currently active in the behavior.

10.1.4.3 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. middle view in Fig. 10.9). To display this information, the following debug requests must be sent:

```
dr representation:InertialSensorData
dr representation:SystemSensorData
dr representation:FsrSensorData
dr representation:KeyStates
```

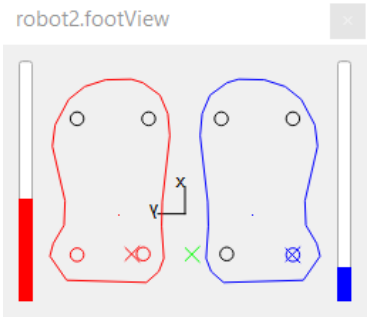


Figure 10.10: Foot View, displaying the pressure sensors' data

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, temperatures, and loads (cf. right view in Fig. 10.9). To display this information, the following debug requests must be sent:

```
dr representation:JointRequest
dr representation:JointSensorData
```

Foot View

The Foot View displays the robot's pressure sensor data from its feet. It can be seen in Fig. 10.10. The bars left and right visualize the current weight on the corresponding foot in relation to the robot's overall weight. In the middle, there are also feet sketches, which display each foot's center of pressure as well as their combined center of pressure (green).

10.1.4.4 Motion Control

Kick View

The basic idea of the kick view shown in Figure 10.11 is to visualize and edit basic configurations of motions for the KickEngine described in [16]. In doing so the central element of 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 KickEngine 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 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 10.1:

$$c(t) = \sum_{i=0}^{n} B_{i,n}(t) P_i \tag{10.1}$$

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

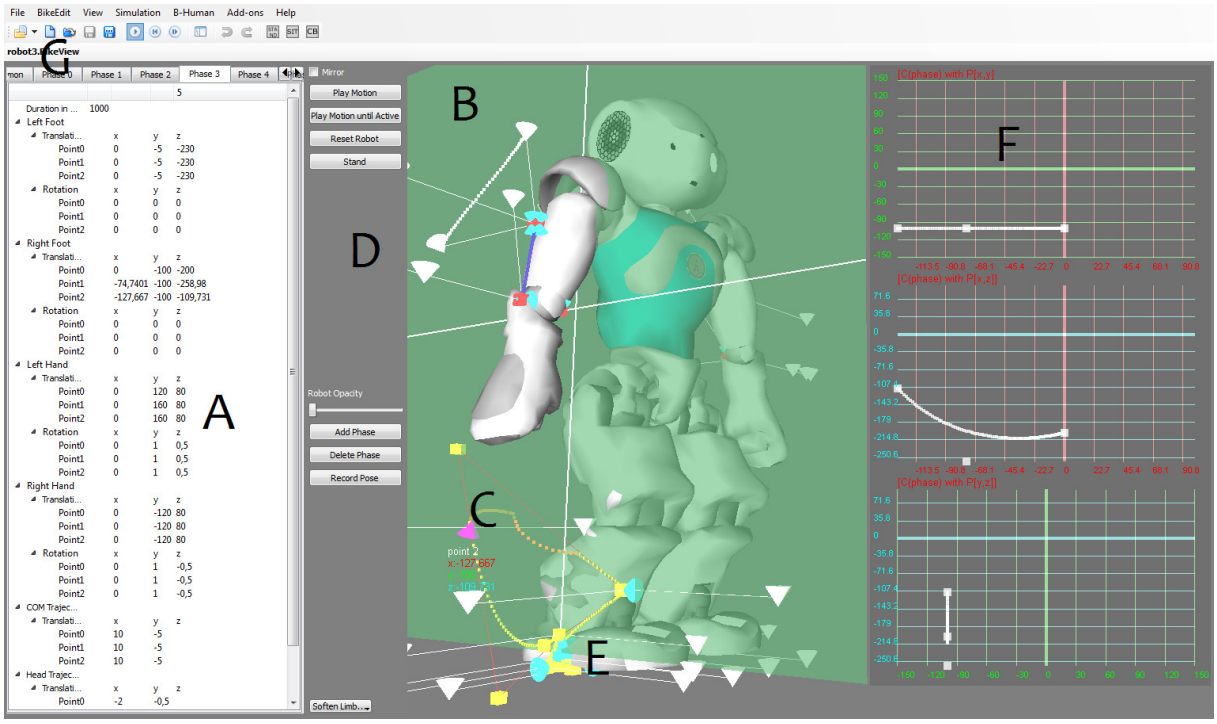


Figure 10.11: The kick view. *A* marks the editor view, *B* denotes the drag and drop plane, *C* points at a clipped curve, *D* tags the buttons that control the 3D-View, e. g. play the motion or reset the robot to a standing position, *E* labels one of the control points, *F* points at the subviews and *G* points at the tool bar, where a motion can be saved or loaded.

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. The user can click on one of the visualized control points (cf. Fig. 10.11 at *E*) and drag it to the desired position. In order to visualize the current dragging plane, a light green area (cf. Fig. 10.11 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. 10.11 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. 10.11 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 second way to display a sub view is by clicking at the *BikeEdit* entry in the menu. This displays the same menu, which appears as context menu.

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

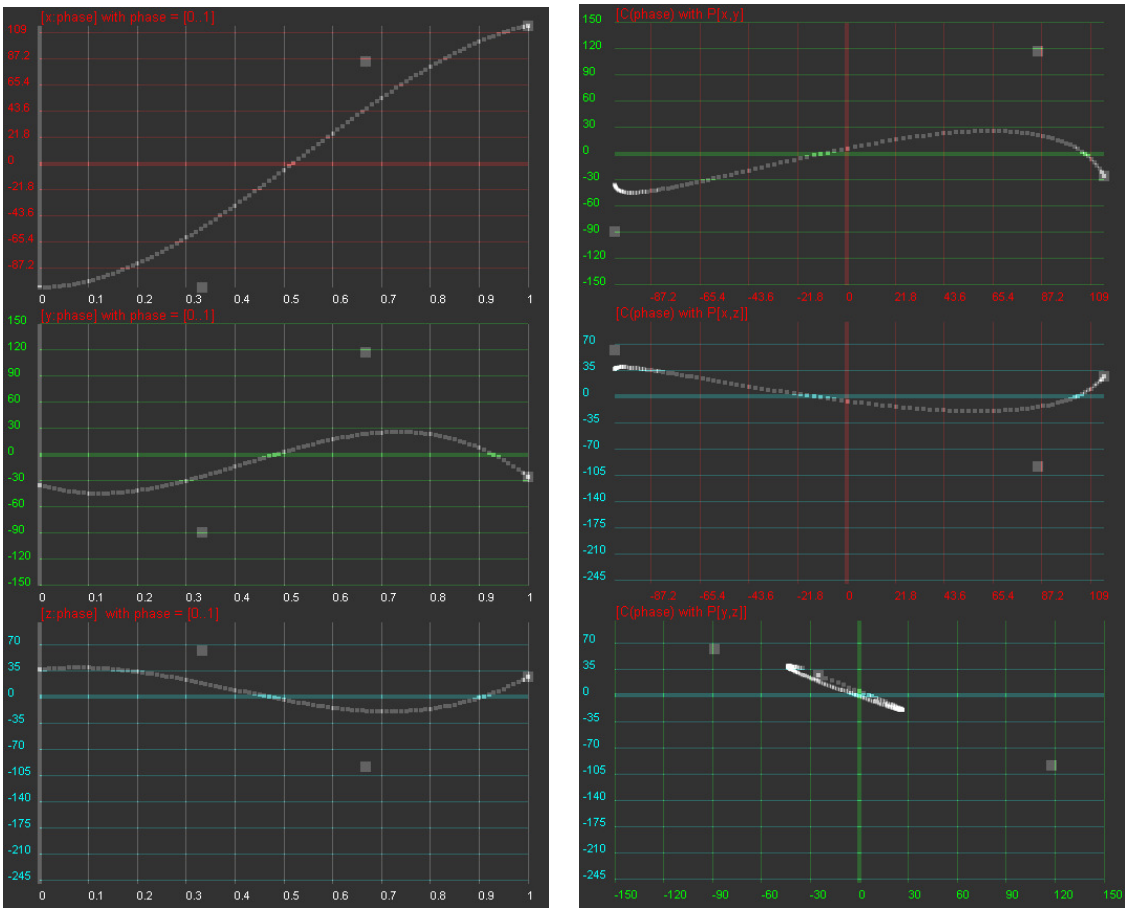


Figure 10.12: Left: 1-D sub views. Right: 2-D sub views

be activated. This can be done by clicking on one of the attached control points.

The 2-D view (cf. Fig. 10.12) 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 10.1 with $P_i = \binom{c_{x_i}}{c_{y_i}}$, $P_i = \binom{c_{x_i}}{c_{z_i}}$ or $P_i = \binom{c_{y_i}}{c_{z_i}}$.

The 1-D sub views (cf. Fig. 10.12) 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 10.1 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 can edit a displayed curve directly in any sub view by drag and drop.

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

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

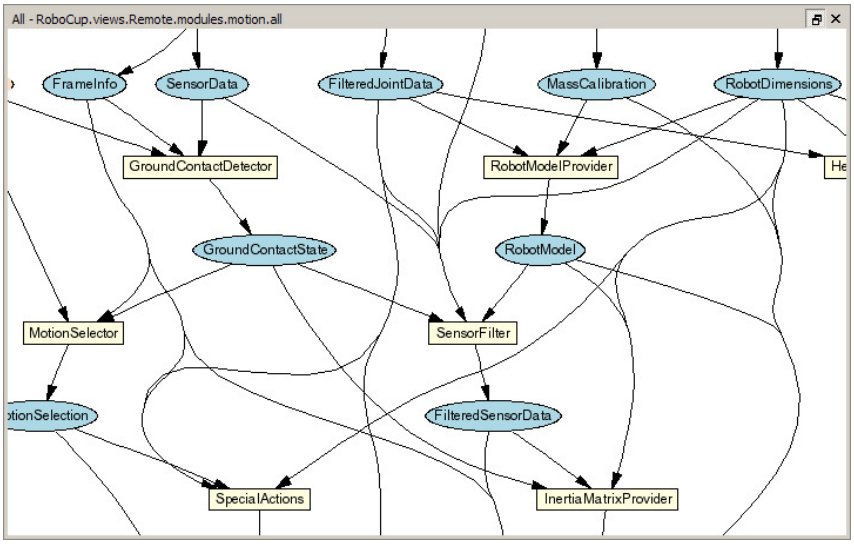


Figure 10.13: The module view shows a part of the modules in the process *Motion*.

To save or load a motion the kick view has to be the active view. Appropriate buttons will appear in the tool bar (cf. Fig. 10.11 at *G*).

10.1.4.5 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 10.13, are generated by the program *dot* from the *Graphviz* package [5]. 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*, *communication*, *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*, touch gestures, or the *page up/down* buttons. It can also be dragged around with the left mouse button or by touch gestures.

Plot Views

Plot views allow plotting data sent from the robot control program through the macro `PLOT` (cf. Fig. 10.14 left). They keep a history of the values sent, up to 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. 10.1.6.3).

For instance, the view on the left side of Figure 10.14 was defined as:

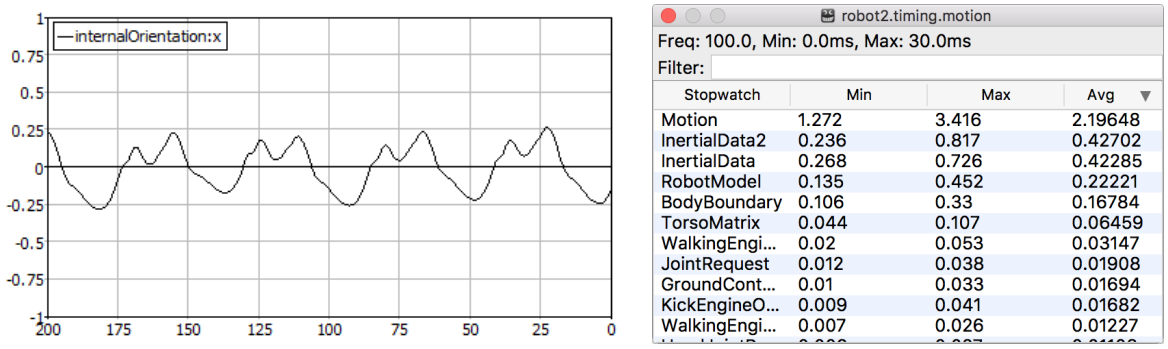


Figure 10.14: Plot view and timing view

```
vp orientationX 200 -1 1
vpd orientationX module:InertialDataProvider:internalOrientation:x
```

Timing View

The timing view displays statistics about all currently active stopwatches in a process (cf. Fig. 10.14 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 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. 10.15 left). Property browsers are well suited for displaying hierarchical data and should be well known from various editors such as 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.

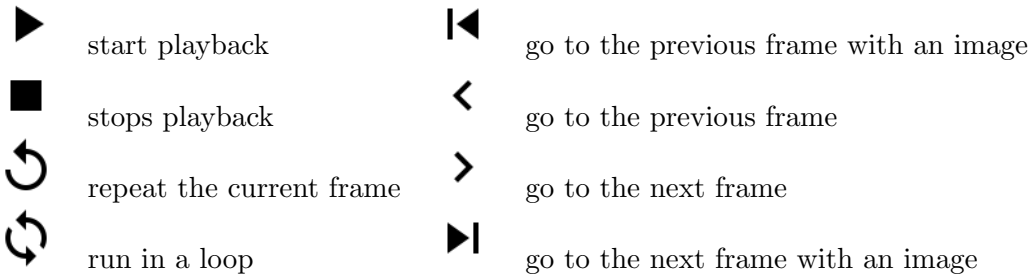
To modify data, just click on the desired field and start editing. The view will stop updating itself as soon as you start editing a field. The editing process is finished either by pressing enter or by deselecting the field. By default, modifications will be sent to the robot immediately. This feature is called the auto-set mode. It can be turned off using the context menu (cf. Fig. 10.15 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.

Log Player View

The log player (cf. Fig. 10.1.4.5 left) allows to control the replay of log files using the following buttons:



In addition, the current frame can be directly selected using a slider.

Annotation View

The annotation view displays all annotations (cf. Sect. 3.7.6) contained in a log file (cf. Fig. 10.16 right). Double clicking an annotation will cause the log file to jump to the given frame number.

It is also possible to view annotations of *Motion* and *Cognition* modules when using the simulated NAO or a direct debug connection to a real NAO. This has to be activated by using the following debug request:

```
dr annotation
```

10.1.5 Scene Description Files

The language of scene description files is an extended version of RoSiML [12]. To use this new version and the new SimRobotCore2, the scene file has to end with *.ros2*, such as *BH2016.ros2*. In the following, the most important elements, which are necessary to add robots, dummies, and balls, are shortly described (based upon *BH2016.ros2*). For a more detailed documentation see Appendix A.

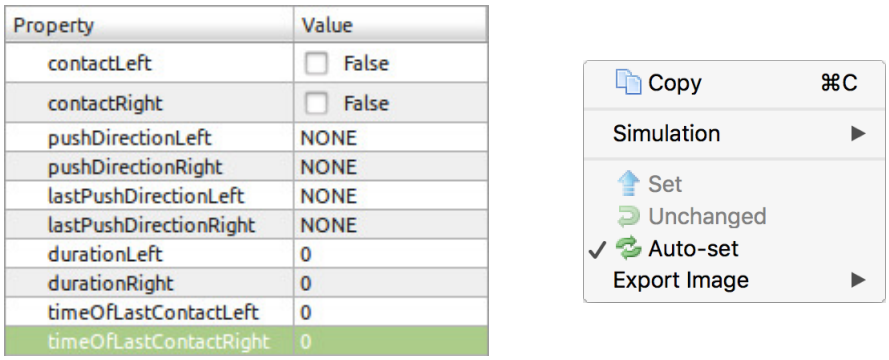


Figure 10.15: The data view can be used to remotely modify data on the robot.

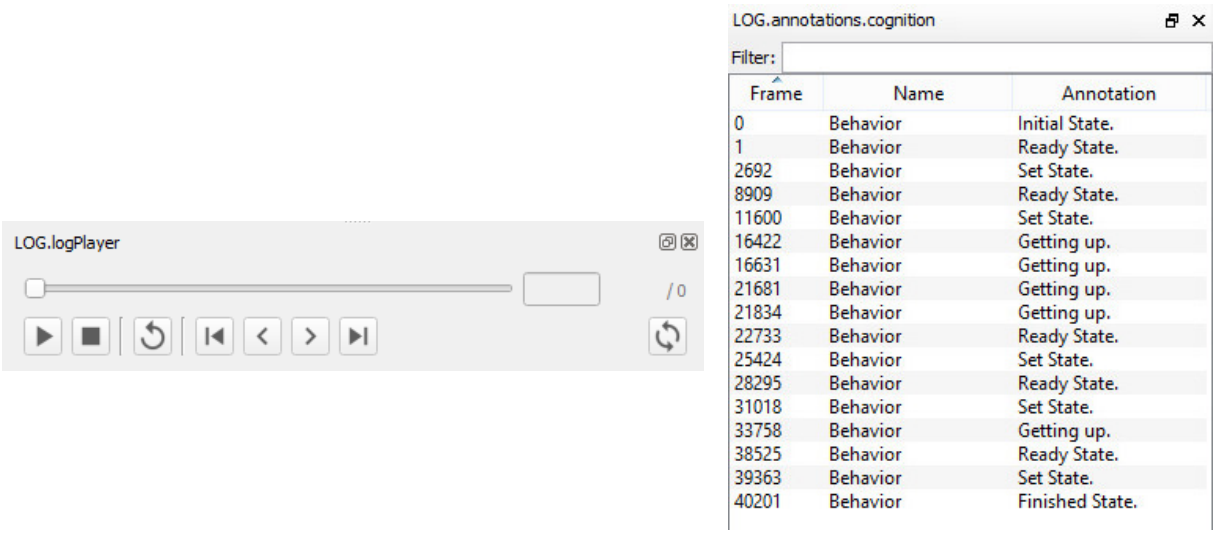


Figure 10.16: The log player view and the annotation view

<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".

Predefined Scenes

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.

Simulating multiple robots is expensive. To overcome this and enable decent frame rates, some scenes come in special variants. In scenes ending with *PerceptOracle* percepts are provided by the simulator. However, models, for example the *BallModel*, still have to be calculated. The suffix *Fast* further implies that even models are provided.

BH2016[PerceptOracle|Fast]: A single robot with five dummies.

Game2016[PerceptOracle|Fast]: A game five against five.

KickViewScene[Remote]: For working on kicks (cf. Sect. 10.1.4.4).

ReplayRobot(Cognition|Motion): Used to replay log files (cf. Sect. 3.7.5).

RemoteRobot: Connects to a remote robot and displays images and data.

10.1.6 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).

10.1.6.1 Initialization Commands

cl <location>
Changes the location of the scene. This command is special, because it only has an effect if run directly from the script that is executed when a scene is loaded. This command and the following one are always executed before any other command in the script, because the location must be set before any robot code is executed, as the location influences the search path of the configuration files loaded.

cs <scenario>
Changes the scenario of the scene. As for the command *cl*, this command only has an effect if run directly from the script that is executed when a scene is loaded. This command and the previous one are always executed before any other command in the script, because the scenario must be set before any robot code is executed, as the scenario influences the search path of the configuration files loaded.

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

When replaying a log file, the replay can be controlled by the *Log Player* (cf. Sect. 10.1.4.5) or the command *log* (see below). It is even possible to load a different log file during the replay.

10.1.6.2 Global Commands

ar off | on
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.

gc initial | ready | set | playing | finished | kickOffBlue | kickOffRed | outByBlue | outByRed | gameDropIn | gamePlayoff | gameRoundRobin
The command sets the current state of the GameController.

(help | ?) [<**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.

10.1.6.3 Robot Commands

ac both | upper | lower
Accept Camera: Change the process that provides drawings in the field and 3-D views.

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%.

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 $[-speed \dots speed]$. 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.

kick
Adds the *KickEngine* view to the *SceneGraph*.

log ? [<**pattern**>] | **mr** [**list**] | **start** | **stop** | **pause** | **forward** [**image**] | **backward** [**image**] | **fastForward** | **fastBackward** | **repeat** | **goto** <**number**> | **time** <**minutes**> <**seconds**> | **clear** | **(** **keep** | **remove** **)** <**message**> {<**message**>} | **keep** **(** **ballPercept** [**seen** | **guessed**] | **ballSpots** | **goalPostPercept** | **image** | **penaltyMarkPercept** **)** | **(** **load** | **save** | **saveImages** [**raw**]) <**file**> | **saveTiming** <**file**> | **cycle** | **once** | **full** | **jpeg** | **merge** | **saveAudio** <**file**> | **saveInertialSensorData** | **saveJointAngleData** | **saveFsrSensorData**
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. 10.1.6.1). The different parameters have the following meaning:

? [<**pattern**>]
Prints statistics on the messages contained in the current log file. The optional pattern limits to messages corresponding to the pattern.

mr [**list**]
Sets the provider of all representations from the log file to CognitionLogDataProvider or MotionLogDataProvider. 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**> | **time** <**minutes**> <**seconds**>
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. If the log file contains *GameInfo*s, the command *time* allows to jump to the first frame with a certain remaining game time.

fastForward | **fastBackward**
Jump 100 steps forward or backward.

clear | **(keep** | **remove)** <**message**> {<**message**>} | **keep (ballPercept [seen** | **guessed]** | **ballSpots** | **goalPostPercept** | **image** | **penaltyMarkPercept)**
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. *keep* also has a second form, in which a condition can be specified that is used to keep the frames (not just the messages) that meet the condition. The conditions are hardcoded. They check, e. g., whether an image was recorded or a certain percept was detected in a frame.

(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.

merge
Tries to find the counterpart log file to the currently loaded one, i. e. a *Cognition* log file if a *Motion* log file is loaded and vice versa, and merges it into the log data in memory.

saveAudio <**file**>
Creates an audio-file and saves a connected robot's audio intake to it.

saveInertialSensorData | **saveJointAngleData** | **saveFsrSensorData**
Saves logged sensor data of the corresponding sensors to a .csv-file next to the log.

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 | **illegalBallContact** | **playerPushing** | **illegalMotionInSet** | **inactivePlayer** | **illegalDefender** | **leavingTheField** | **kickOffGoal** | **requestForPickup** | **manual**
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. 10.1.4.1). 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. 10.1.4.1). A field view is the means for displaying debug drawings in field coordinates. The parameter defines the *name* of the view.

vfd ? [<**pattern**>] | **(** <**name**> | **all** **)** **(** **?** [<**pattern**>] | <**drawing**> **(** **on** | **off** **)** **)** | **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) or *all* to add the drawing to all field views. 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. *vfd off* will turn off all debug drawings except for the background in all field views.

vi ? [<**pattern**>] | <**image**> [**jpeg**] [**segmented**] [**upperCam**] [<**name**>] [**gain** <**value**>]
Adds an image view (cf. Sect. 10.1.4.1). 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**> | **all** **)** **(** **?** [<**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) or *all* to add the drawing to all image views. 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

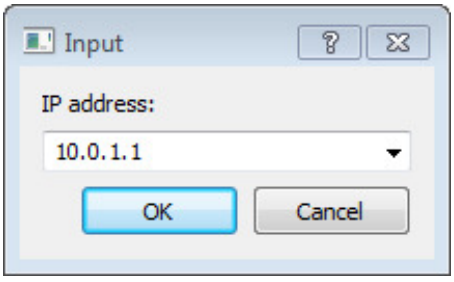


Figure 10.17: A dialog for selecting an IP address

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. *vid off* will turn off all debug drawings in all image views.

vp <name> <numOfValues> <minValue> <maxValue> [<yUnit> <xUnit> <xScale>]
Adds a plot view (cf. Sect. 10.1.4.5). 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. *black*, *red*, *green*, *blue*, *yellow*, *cyan*, *magenta*, *orange*, *violet*, *gray*, and six-digit hexadecimal RGB color codes are supported. 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.

10.1.6.4 Selection Dialogs

Selection Dialogs may be used in scripting-files and allow for user input. The dialogs will be opened before their respective script-lines are executed and the user selected value handled as usual script, e.g. a command or parameter.

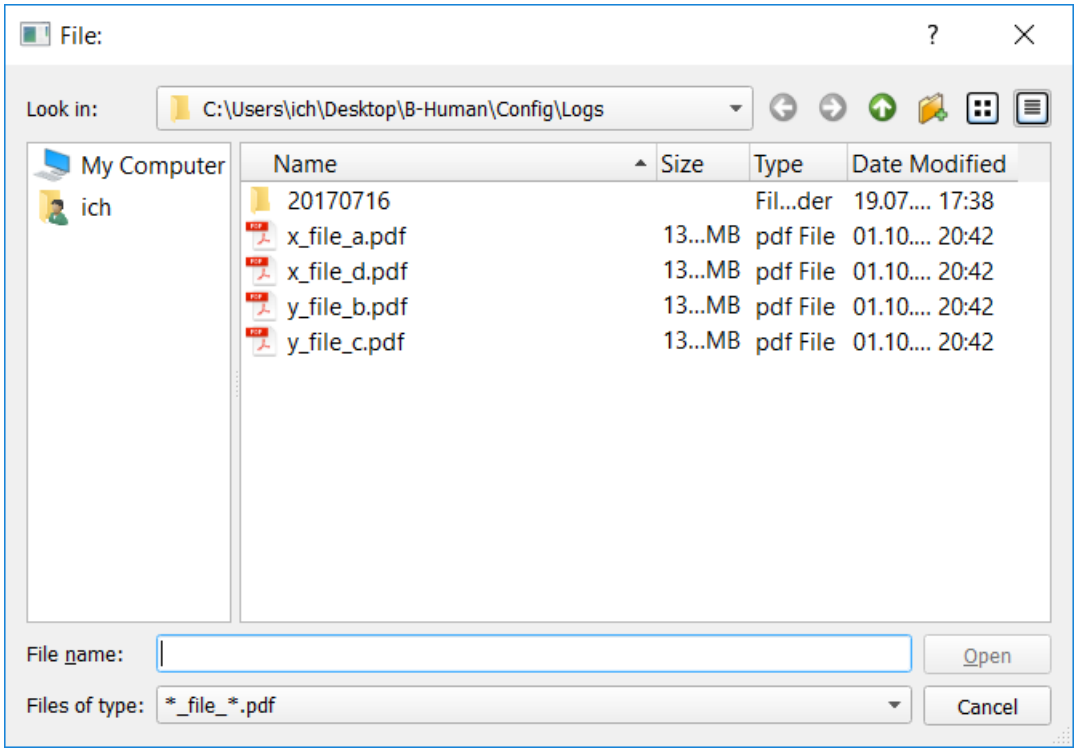


Figure 10.18: A dialog for selecting a file

${<label>,<value-1>,<value-2>{,<value-x>}}[2]
This expression opens a dropdown list with the respective name and value options. An example of *${IP address:,10.0.1.1,192.168.1.1}* is shown in figure 10.17.

${<label>,<relativePath>}[2]
This expression opens a file selection dialog at the relative path. The path may also specify legal file names via the asterisk-character (*). An example of *${File:,../Logs/*_file_*.pdf}* is shown in figure 10.18.

10.1.7 Recording a Remote 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:JointAngles
dr representation:InertialSensorData
```

[2]The outer curly braces are literals.

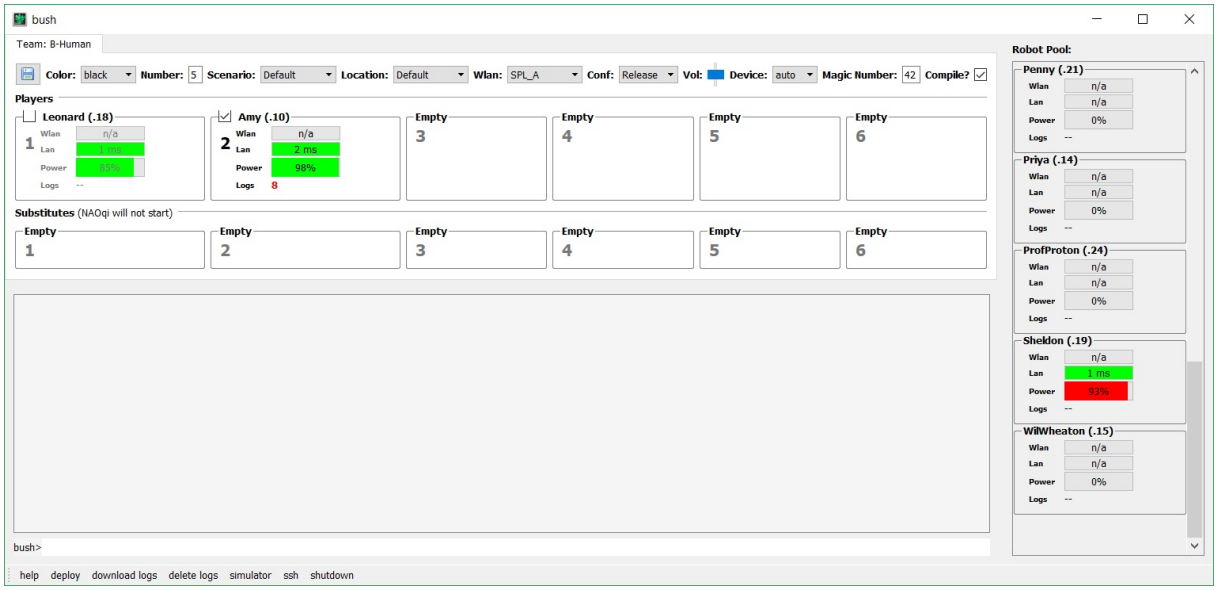


Figure 10.19: An example screenshot of the *bush*

```
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
```

10.2 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.

10.2.1 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*.[3] 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 10.1.

[3]The configuration file is created by the script *createRobot* described in Sect. 2.4.3.

Entry	Description
name	The name of the team.
number	The team number.
port	The port, which is used for team communication messages.
color	The team color in the first half.
scenario	The scenario, which should be used by the software (cf. Sect. 2.9).
location	The location, which should be used by the software (cf. Sect. 2.9).
compile	Should the code be compiled before it is deployed?
buildConfig	The name of the configuration, which should be used to deploy the NAO code (cf. Sect. 2.2).
wlanConfig	The name of the configuration file, which should be used to configure the wireless interface of the robots.
volume	The audio volume, to which the robots should be set to.
deployDevice	The device, which should be used to connect to the robots. Either a Ethernet or a WiFi connection can be established. The default entry is auto. This chooses the best device, depending on ping times.
magicNumber	The magic number for the team communication. Robots with different numbers will ignore each other. The default number is -1, which will set a random number between 0 and 255.
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 10.1: Configuration variables in the file *Config/teams.cfg*

10.2.2 Commands

The *bush* supports two types of commands. There are local commands (cf. Tab. 10.2) and commands that interact with selected robot(s) (cf. Tab. 10.3). Robots can be selected by checking their checkbox or with the keys *F1* to *F10*.

10.2.3 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 independently from the *deploy* command. Depending on the platform, the *compile* command uses *make*, *xcodebuild*, or *MSBuild* to compile the binaries required.

After all the files required by the NAO are copied, the *deploy* command generates a new *settings.cfg* according to the configuration tracked by the *bush* for each of the selected robots. 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,

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 10.2: General *bush* commands.

Command	Parameter(s) Description
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.
deleteLogs	 Deletes all logs from 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 an command via ssh or opens a ssh session

Table 10.3: *Bush* commands that need at least one selected robot.

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.

10.2.4 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 WiFi access point, the robots have to deal with multiple configuration files for their wireless interfaces. When a robot is deployed the selected wireless configuration in the drop down menu will be used.

10.2.5 Locations and Scenarios

To be able to easily change config files which are tailored for a specific physical location or use case they are saved in different directories. They can be selected in the drop down menus for location and scenario, where only the selected one will be deployed.

10.2.6 Substituting Robots

The robots known to the *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 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. If a robot is deployed in the second row, NAOqi will be shut down. This is important because the robots should be operational, to be replaced as fast as possible, but are not allowed to send and receive packages.

10.2.7 Monitoring Robots

The *bush* displays some information about the robots' states as you can see in Figure 10.19: wireless connection pings, wired connection pings, remaining battery charge level, and number of logfiles stored on the robot. The battery charge level is shown by the power bar. If the power bar is colored green, a power source is detected and if it's red no power source is detected. Besides the color change of the power bar the robot's head LEDs will display a rotating pattern to indicate a connected external power source.

10.3 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*. The official *GameController* is the one we created and is written in Java 1.7.

10.3.1 Architecture

The architecture (cf. Fig. 10.20) is based on a combination of the model-view-controller (MVC) and the command pattern.

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 broadcast via UDP several times per seconds. Robots may answer using the `RoboCupGameControlReturnData` C data structure.

Both C data structures were translated to Java for the GameController.[4] 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

[4] Their names leave out the prefix "RoboCup"

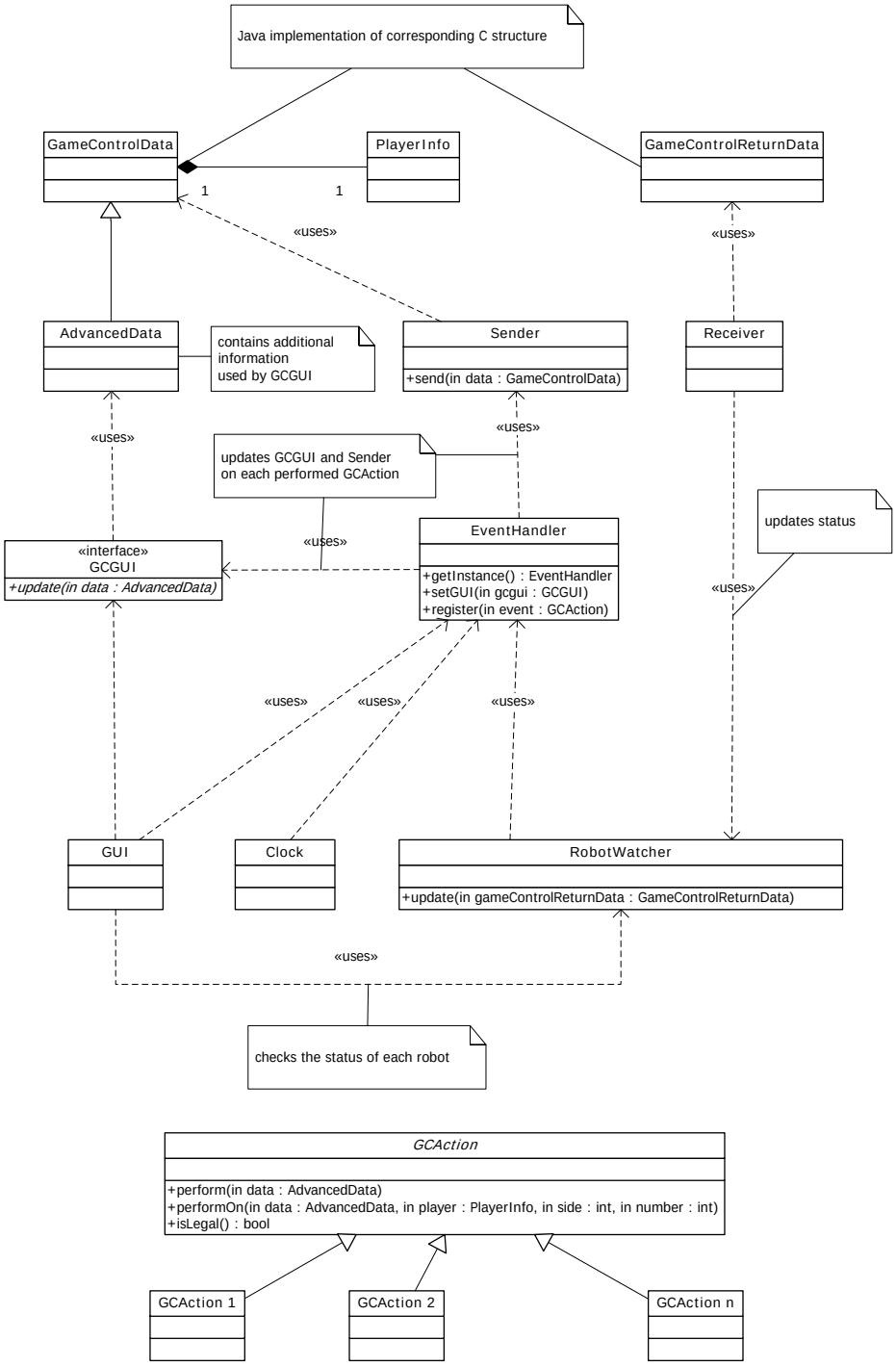


Figure 10.20: The architecture of the *GameController*

`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

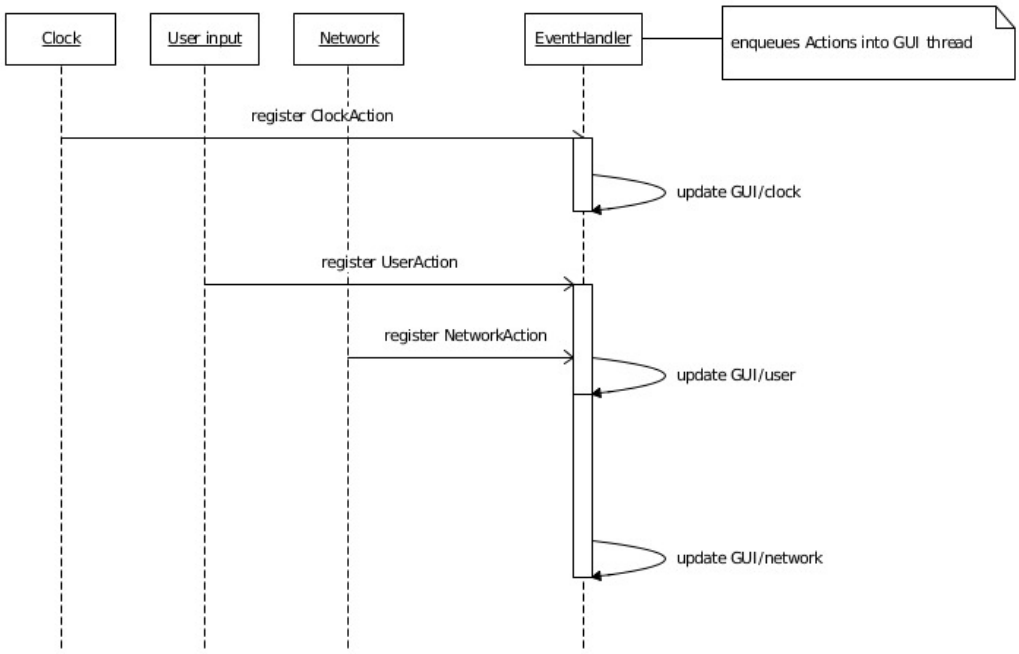


Figure 10.21: The sequences between some threads

`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. 10.21).

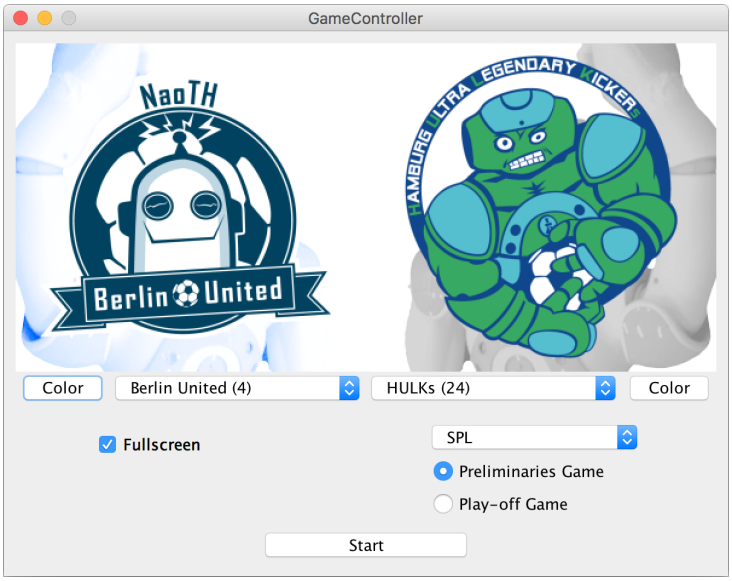


Figure 10.22: Start screen of the *GameController*

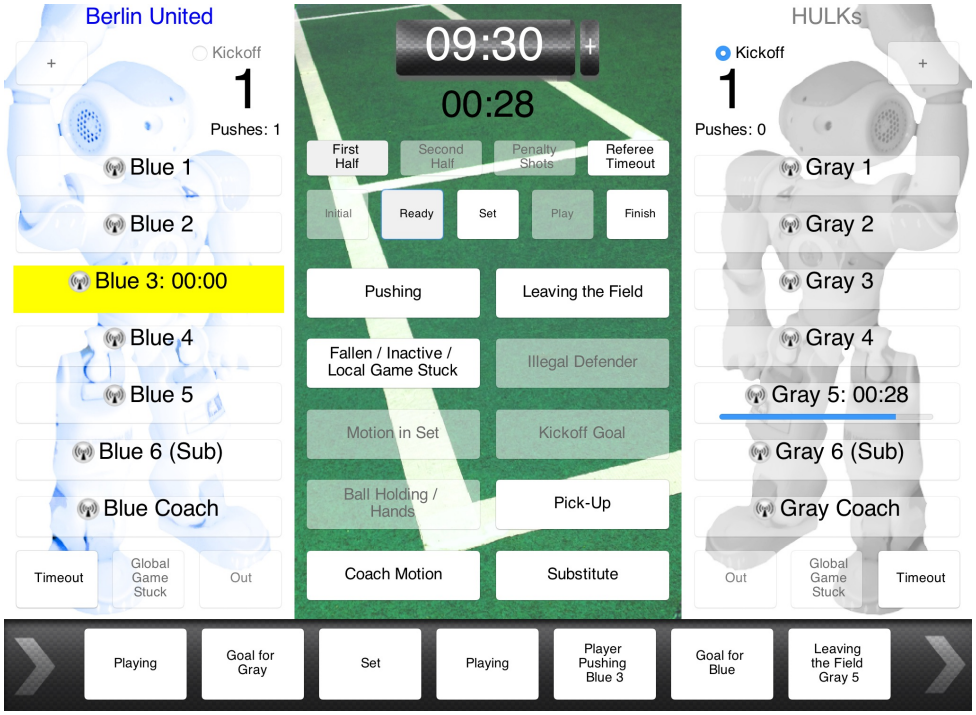


Figure 10.23: The main screen of the *GameController*

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.

10.3.2 UI Design

After launching the *GameController*, a small window will appear to select the basic settings of the game (cf. Fig. 10.22). The most basic decision is the league. You can choose between SPL, SPL Drop-in 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. For SPL teams, it can also be switched between the primary and the secondary jersey color. After pressing the start button, the main GUI will appear.

The look of the GUI (cf. Fig. 10.23) 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

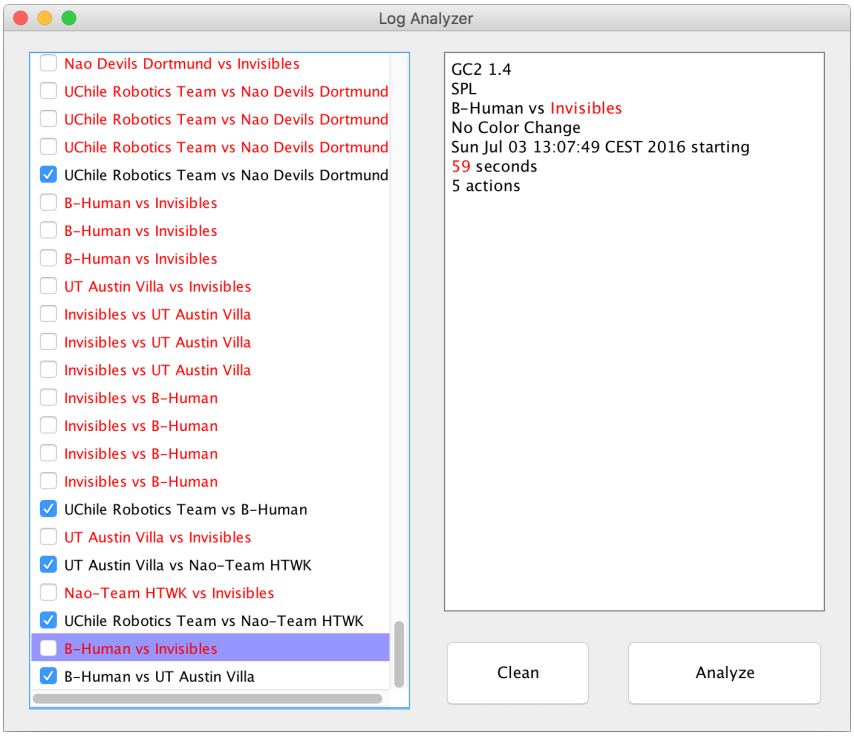


Figure 10.24: The *LogAnalyzer*

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.

Since 2017, the current penalty taker and penalty keeper can be selected in each round during a penalty shootout. All other robots will be set as substitutes.

10.3.3 Logging

The *GameController* writes some status information and all decisions entered to a log file with a unique name. Besides, it also includes the logging mechanism of the Team Communication Monitor (see 10.4), logging all messages sent by robots during a game so all games of a tournament can be replayed in the *TCM* afterwards. As long as a GameController is running on the same machine, the *TCM* does not log any data by itself.

10.3.4 Log Analyzer

As 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. 10.24). 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

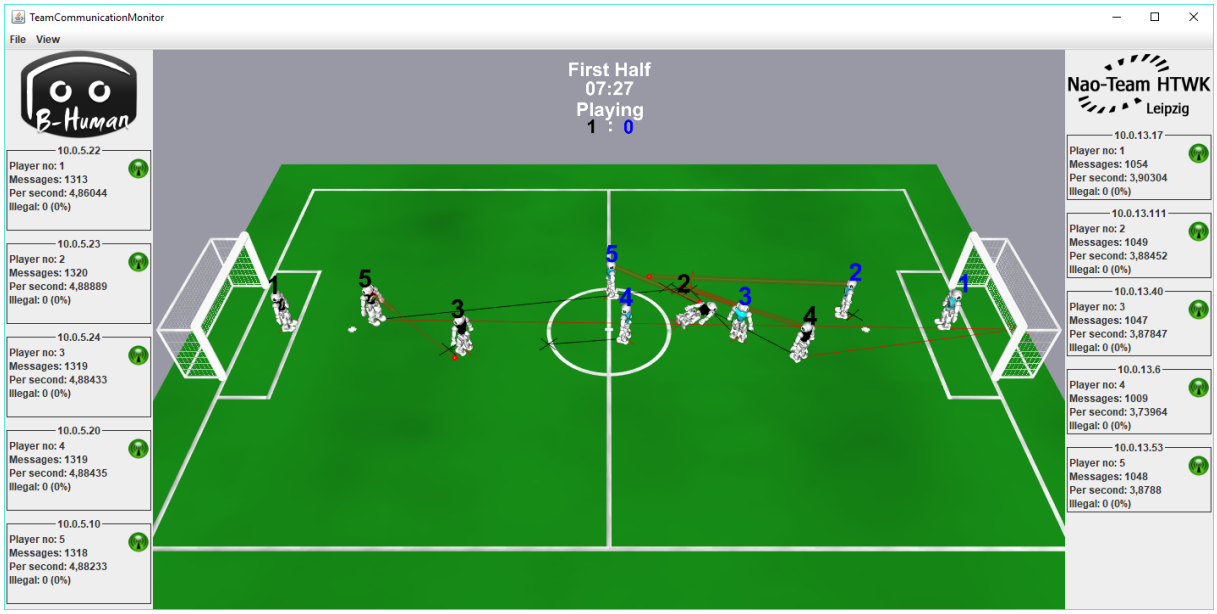


Figure 10.25: Display of the *TeamCommunicationMonitor* during a game between the teams Nao-Team HTWK and B-Human

the form of comma separated values can be created.

10.4 Team Communication Monitor

With the start of the Drop-in Player Competition in 2014, a standard communication protocol, i. e. the `SPLStandardMessage`, was introduced to allow robots of different teams to exchange information. While implementing the standard protocol correctly is a necessity for the Drop-in Player Competition to work, it is also an opportunity for teams to lower the amount of coding they have to do to develop debugging tools, because when all teams use the same communication protocol, the tools they wrote could also be shared more easily. Therefore, B-Human developed the *TeamCommunicationMonitor* (TCM) as a standard tool to monitor the network traffic during SPL games. This project has been partially funded by the RoboCup Federation in 2015 and 2017.

10.4.1 Functionality

The *TCM* visualizes all data currently sent by the robots and highlights illegal messages. Furthermore, it uses knowledge about the contents of `SPLStandardMessage`s in order to visualize them in a 3-D view of the playing field. This makes it also suitable for teams to debug their network code. Visualized properties of robots are their position and orientation, their fallen and penalty states (the latter is received from the *GameController*), where they last saw the ball, and their player number. A screenshot of the *TCM* can be seen in Fig. 10.25.

In order to display all messages of robots sending packets, the *TCM* binds sockets to all UDP ports that may be used by a team as their team number, i. e. the ports 10000 to 10099, and listens for any incoming packets. When a packet is received, it is parsed as an `SPLStandardMessage`, marking all fields as invalid that do not contain legal values according to the definition of the `SPLStandardMessage`. As the *TCM* only listens and does not send anything, it may run on

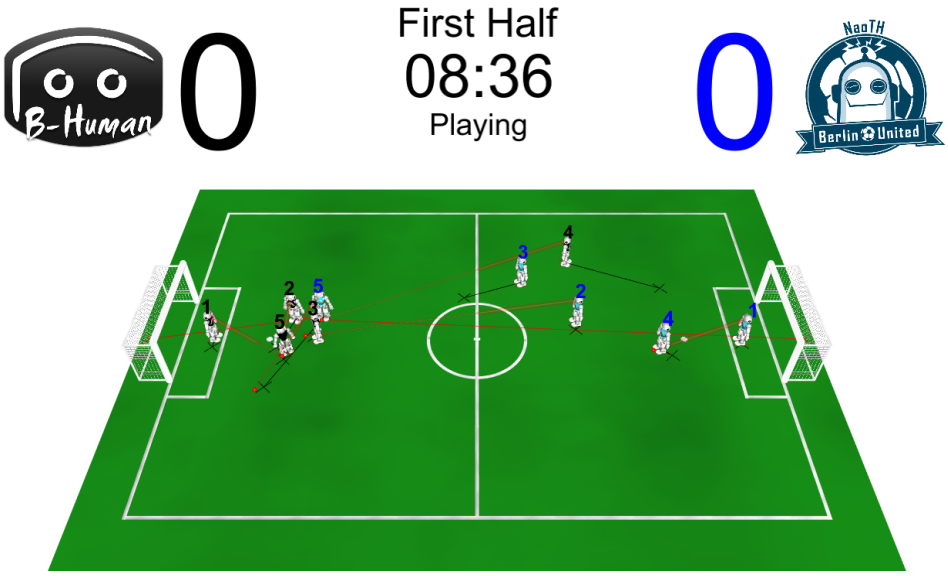


Figure 10.26: The *GameStateVisualizer* mode of the *TCM*

multiple computers in the same network at once without any interferences.

The *TCM* identifies robots using their IP address and assumes them to be sending on the port that matches their team number. The team number sent by the robots as part of the `SPLStandardMessage` is marked as invalid if it does not match the port on which the messages are received. For each robot, the *TCM* holds an internal state containing the last received message as well as the time stamps of the most recently received messages in order to calculate the number of sent messages per second. A robot's state is discarded if no message was received from the robot for at least ten seconds.

Displaying the 3-D view is done with OpenGL using the JOGL library [1]. The visualization subsystem is also extensible to enable teams to write plug-ins containing drawings that visualize data from the non-standardized part of their messages. We have written a plugin to display perceptions of field features, obstacles, and whistles as well as basic status information which our team communicates via the non-standardized message part.

Besides just visualizing received messages, the *TCM* also stores all received messages both from robots and the *GameController* in log files. These can be replayed later, allowing teams and organizers to check the communication of robots after the game is over.

The *TCM* is developed as part of the repository of the *GameController* and it shares parts of its code.

After having been successfully used at the RoboCup German Open 2015 to ensure valid messages from all teams during the Drop-in games, the *TCM* was publicly released in early June 2015 and was installed on the referee PC on each SPL field at all RoboCups since 2015.

10.4.2 Game State Visualizer

The GameController had always been bundled with another tool, the *GameStateVisualizer*, which had the purpose of displaying the current state of a game to the audience. For this year's competitions, this functionality has been included into the *TCM* as a separate mode which can be entered either by passing the command line parameter `--gsv` or via the menu bar of the *TCM*'s main window.

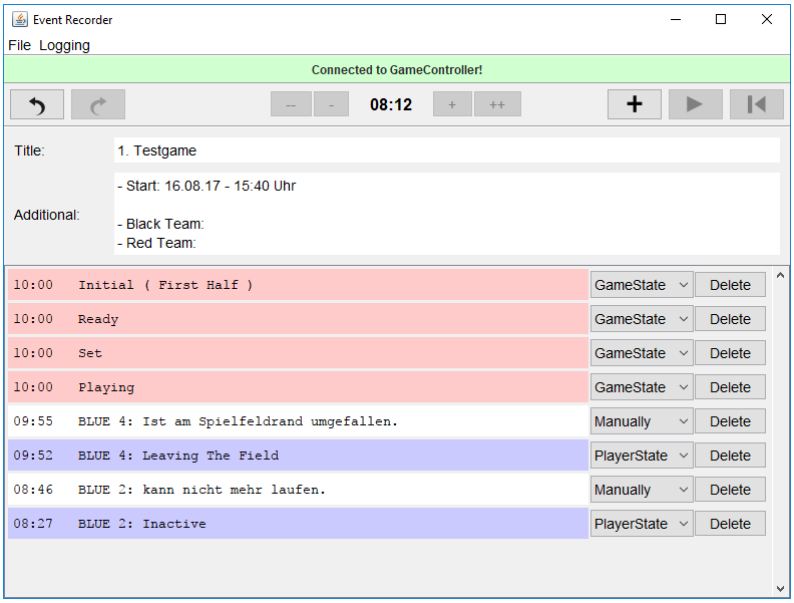


Figure 10.27: The EventRecorder

In addition to visualizing the states of the robots on the field like the normal mode of the *TCM*, the *GameStateVisualizer* also displays the teams playing, the current score, the time remaining, and some other information in such a way that it can be easily read by the audience (cf. Fig. 10.26).

In the past, for the purpose of testing the network or packages the robots should receive, the *GameStateVisualizer* always came with a test mode that showed everything that was contained in the *GameController* packages and how it should be interpreted. This functionality has not been integrated into the *TCM*. Instead, there is now a separate program, the *GameControllerTester*, containing this functionality.

10.5 Event Recorder

The *EventRecorder* is an independent semi-automatic logging software for real robot games, which is written in Java and is integrated in the *GameController* project in 2017. For automatic logging of game state changes, penalties, and the time, it communicates with the *GameController*. If the *GameController* is not on the same network as the *EventRecorder*, the time can be synchronized manually.

There are three types of events:

GameState: The game states *Initial*, *Ready*, *Set*, *Play*, and *Finish*, which are automatically logged if the GameController is on the same network.

PlayerState: The penalties of individual robots, which are automatically logged if they are activated in the menu *Logging*, see Fig. 10.28.

Manually: Events which are not triggered by the GameController and can't be automatically logged, e. g. "Black 2 walks too slow and falls often."

For a higher usability, there is an undo and redo functionality. It is decoupled from automatic events and just undoes the manually executed actions such as changes. For example: There

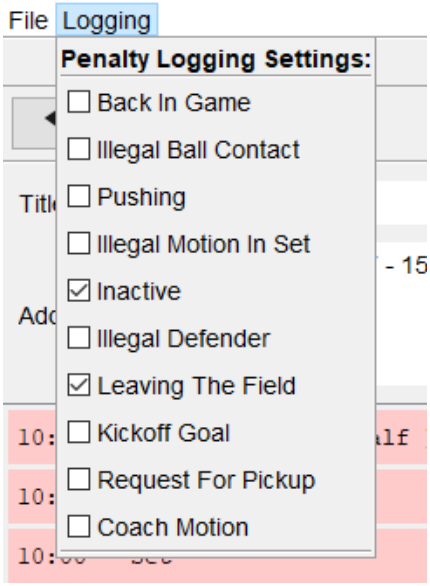


Figure 10.28: Penalties which can be automatically logged

are no entries in the *EventRecorder* in the beginning and while the game takes place, there are entries by the GameController automatically added, entries added manually by the user, and some entries automatically added are changed by the user. If the user would undo all actions afterwards, the entries added automatically were the only ones left.

There are the following shortcuts:

- Create a new entry (Ctrl + Enter)
- Undo (Ctrl + Z)
- Redo (Ctrl + Y)
- Delete the last word (Ctrl + W)
- Increase time by 5 seconds (Ctrl+Plus) or 60 seconds (Ctrl+Shift+Plus)
- Decrease time by 5 seconds (Ctrl+Minus) or 60 seconds (Ctrl+Shift+Minus)

The implementation is based on the model-view-controller pattern. The main class `EventRecorder` initializes the three following components:

DataModel model: This object contains all the informations in the EventRecorder.

ActionHistory history: This object controls the executions of user actions and offers the interface of the undo-functionality.

MainFrame gui: This object represents the view and inherits from Java's `JFrame`.

The components of the model (`eventrecorder.data`), of the control (`eventrecorder.action`), and of the view (`eventrecorder.gui`) are located in individual packages and named as the packages of the *GameController* project.

In the package `eventrecorder.export` is a class which implements a markdown export feature. The communication with the *GameController* is done in the main class `EventRecorder`.

The command pattern is used for the undo and redo functionality. All possible user actions are implemented as a class and inherit from the class `Action` and implement the abstract methods `executeAction()` and `undoAction()`.

Chapter 11

Acknowledgements

We gratefully acknowledge the support given by SoftBank Robotics. We also would like to thank our team sponsor CONTACT Software and our other sponsors Wirtschaftsförderung Bremen, TME, IGUS, Portescap, Alumni of the University of Bremen, neuland, MLP, and AVM for funding parts of our project. Since B-Human 2017 did not start its software from scratch, we also want to thank the previous team members as well as the members of the GermanTeam 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:

Artistic Style: Source code formatting in the *AStyle for B-Human* text service on macOS. (`http://astyle.sourceforge.net`)

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`)

FFTW: For performing the Fourier transform when recognizing the sounds of whistles.
(`http://www.fftw.org`)

getModKey: For checking whether the shift key is pressed in the Deploy target on macOS.
(`http://allancraig.net/index.php?option=com_docman&Itemid=100`, not available anymore)

gtest: A very powerful test framework.
(`https://code.google.com/p/googletest/`)

ld: The GNU linker is used for cross linking on Windows and macOS.
(`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.
(`https://bitbucket.org/libqxt/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://www.qt.io`)

qtpropertybrowser: Extends the Qt framework with a property browser.
(`https://github.com/qtproject/qt-solutions/tree/master/qtpropertybrowser`)

snappy: Used for the compression of log files.
(`http://google.github.io/snappy`)

Walk2014Generator: The module `Walk2014Generator` is based on the class of the same name released by the team UNSW Australia as part of their code release. The team kindly gave us the permission to release our derived module under our license.
(`https://github.com/UNSWComputing/rUNSWift-2016-release`)

Bibliography

[1] JOGL - Java binding for the OpenGL API. http://jogamp.org/jogl/www/.

[2] RoboCup Technical Committee. RoboCup Standard Platform League (NAO) rule book, 2016. Only available online: http://www.informatik.uni-bremen.de/spl/pub/Website/Downloads/Rules2016.pdf.

[3] 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.

[4] Matteo Frigo and Steven G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005. Special issue on "Program Generation, Optimization, and Platform Adaptation".

[5] 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.

[6] Google. Snappy – a fast compressor/decompressor. Online: http://code.google.com/p/snappy, September 2013.

[7] Bernhard Hengst. rUNSWift Walk2014 report. Technical report, School of Computer Science & Engineering University of New South Wales, Sydney 2052, Australia, 2014. http://cgi.cse.unsw.edu.au/~robocup/2014ChampionTeamPaperReports/20140930-Bernhard.Hengst-Walk2014Report.pdf.

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

[9] 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.

[10] 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.

[11] Tim Laue, Thomas Röfer, Katharina Gillmann, Felix Wenk, Colin Graf, and Tobias Kastner. B-Human 2011 – eliminating game delays. In Thomas Röfer, N. Michael Mayer, Jesus Savage, and Uluç Saranlı, editors, *RoboCup 2011: Robot Soccer World Cup XV*, volume 7416 of *Lecture Notes in Artificial Intelligence*, pages 25–36. Springer, 2012.

[12] 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.

[13] 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.

[14] Scott Lenser and Manuela Veloso. Sensor resetting localization for poorly modelled mobile robots. In *Proceedings of the 2000 IEEE International Conference on Robotics and Automation (ICRA 2000)*, volume 2, pages 1225–1232, San Francisco, CA, USA, 2000.

[15] Judith Müller, Udo Frese, and Thomas Röfer. Grab a mug - object detection and grasp motion planning with the nao robot. In *Proceedings of the IEEE-RAS International Conference on Humanoid Robots (HUMANOIDS 2012), Osaka, Japan*, pages 349–356. IEEE, 2012.

[16] 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.

[17] Nobuyuki Otsu. A threshold selection method from grey level histograms. *IEEE Transactions on Systems, Man, and Cybernetics*, 9(1):62–66, January 1979.

[18] Thomas Reinhardt. Kalibrierungsfreie Bildverarbeitungsalgorithmen zur echtzeitfähigen Objekterkennung im Roboterfußball. Master's thesis, HTWK Leipzig, 2011.

[19] 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.

[20] 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.

[21] Thomas Röfer and Tim Laue. On B-Human's code releases in the Standard Platform League – software architecture and impact. In Sven Behnke, Manuela Veloso, Arnoud Visser, and Rong Xiong, editors, *RoboCup 2013: Robot World Cup XVII*, volume 8371 of *Lecture Notes in Artificial Intelligence*, pages 648–656. Springer, 2014.

[22] Thomas Röfer, Tim Laue, Jonas Kuball, Andre Lübken, Florian Maaß, Judith Müller, Lukas Post, Jesse Richter-Klug, Peter Schulz, Andreas Stolpmann, Alexander Stöwing, and Felix Thielke. B-Human team report and code release 2016, 2016. Only available online: `http://www.b-human.de/downloads/publications/2016/coderelease2016.pdf`.

[23] Thomas Röfer, Tim Laue, and Andre Mühlenbrock. B-Human team description for RoboCup 2017. In *RoboCup 2017: Robot World Cup XXI Preproceedings*, Nagoya, Japan, 2017. RoboCup Federation.

[24] 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`.

[25] Thomas Röfer, Tim Laue, Jesse Richter-Klug, Jonas Stiensmeier, Maik Schünemann, Andreas Stolpmann, Alexander Stöwing, and Felix Thielke. B-Human team description for RoboCup 2015. In *RoboCup 2015: Robot World Cup XIX Preproceedings*, Hefei, China, 2015. RoboCup Federation.

[26] 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`.

[27] Thomas Röfer. CABSL – C-based agent behavior specification language. In *RoboCup 2017: Robot World Cup XXI*, Lecture Notes in Artificial Intelligence. Springer, 2018. To appear.

[28] Sebastian Thrun, Wolfram Burgard, and Dieter Fox. *Probabilistic Robotics*. MIT Press, Cambridge, 2005.

[29] Max Trocha. Werkzeug zur taktischen Auswertung von Spielsituationen. Bachelor's thesis, University of Bremen, 2010.

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;
normalsClass         = Normals;
primitiveGroupClass  = Quads | Triangles;
rotationClass        = Rotation;
sceneClass           = Scene;
setClass             = Set;
solverClass          = Quicksolver;
texCoordsClass       = TexCoords;
translationClass     = Translation;
userInputClass       = UserInput;
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,
                       normalsClass, 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>";

UserInput             = "<UserInput></UserInput>" | "<UserInput/>";

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/>";

    Normals             = "<Normals> Normals Definition "</Normals>";
    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 library *SimulatedNao* is loaded by SimRobot). It is recommended to include other specifications per *Include* before the scene description starts (compare with *BH2016.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 a NAO in its definition file (*NaoV4H21.rsi2*), but NAOs with different jersey colors are needed on a field. For each NAO on the field, there is a reference to the original model. The positioning of the NAOs is done by *Translation* and *Rotation* elements. The color is set by a *Set* element, which is described below.

```
<Body name="Nao">
  <Set name="NaoColor" value="blue"/>
⋮
</Body>
⋮
<Body ref="Nao" name="BlueNao">
  <Translation x="-2" y = "0.4" z="320mm"/>
</Body>
<Body ref="Nao" name="RedNao">
  <Translation x="-1.5" y="-0.9" z="320mm"/>
  <Rotation z="180degree"/>
  <Set name="NaoColor" value="red"/>
</Body>
⋮
```

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 NAO's jersey is set by a *Set* element. Within the definition of the body *Nao* named *RedNao*, the *Set* element sets the placeholder color to the value *red*. The placeholder named *NaoColor* of *Nao*, which is defined in the general definition of a NAO, is replaced by *red* in all elements of the model, also in the ones that are just referenced, such as the appearances of individual body parts. So the *Surface* elements reference a *Surface* named *nao-red*.

```
⋮
<ComplexAppearance name="CHEST_COORD_MACROCOL">
  <Surface ref="nao-$NaoColor"/>
  <Vertices ref="CHEST_COORD"/>
⋮
</ComplexAppearance>
⋮
<Surface name="nao-red" diffuseColor="rgb(100%, 0%, 100%)" ambientColor="rgb
    (20%, 12%, 12%)"/>
⋮
```

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.4.23.
  - 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
    - **Use:** required
    - **Range:** $[0, MAXFLOAT]$
  - width The width of the box.
    - **Units:** mm, cm, dm, m, km
    - **Use:** required
    - **Range:** $[-MAXFLOAT, MAXFLOAT]$
  - height The height of the box.
    - **Units:** mm, cm, dm, m, km
    - **Use:** required
    - **Range:** $[-MAXFLOAT, MAXFLOAT]$
  - depth The depth of the box.
    - **Units:** mm, cm, dm, m, km
    - **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
    - **Use:** required
    - **Range:** $[0, MAXFLOAT]$
  - radius The radius of the sphere.
    - **Units:** mm, cm, dm, m, km
    - **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
    - **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:** $g * mm^2$, $kg * m^2$
    - **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:** $g * mm^2$, $kg * m^2$
    - **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:** $g * mm^2$, $kg * m^2$
    - **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.4.23
  - name
    - **Use:** optional
    - **Range:** String
  - width The width of the box.
    - **Units:** mm, cm, dm, m, km
    - **Use:** required
    - **Range:** $[-MAXFLOAT, MAXFLOAT]$
  - height The height of the box.
    - **Units:** mm, cm, dm, m, km
    - **Use:** required
    - **Range:** $[-MAXFLOAT, MAXFLOAT]$
  - depth The depths of the box.
    - **Units:** mm, cm, dm, m, km
    - **Use:** required
    - **Range:** $[-MAXFLOAT, MAXFLOAT]$
- **SphereGeometry**
  - color A color definition, see A.4.23
  - name
    - **Use:** optional
    - **Range:** String
  - radius The radius of the sphere.
    - **Units:** mm, cm, dm, m, km
    - **Use:** required
    - **Range:** $[-MAXFLOAT, MAXFLOAT]$
- **CylinderGeometry**
  - color A color definition, see A.4.23
  - name
    - **Use:** optional
    - **Range:** String
  - radius The radius of the cylinder.
    - **Units:** mm, cm, dm, m, km

        * **Use:** required
        * **Range:** $[-MAXFLOAT, MAXFLOAT]$
    - height The height of the cylinder.
        * **Units:** mm, cm, dm, m, km
        * **Use:** required
        * **Range:** $[-MAXFLOAT, MAXFLOAT]$

- **CapsuleGeometry**
    - color A color definition, see A.4.23
    - name
        * **Use:** optional
        * **Range:** String
    - radius The radius of the capsule.
        * **Units:** mm, cm, dm, m, km
        * **Use:** required
        * **Range:** $[-MAXFLOAT, MAXFLOAT]$
    - height The height of the capsule.
        * **Units:** mm, cm, dm, m, km
        * **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.
        * **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.
    - ∗ **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
    - ∗ **Use:** required
    - ∗ **Range:** $[-MAXFLOAT, MAXFLOAT]$
  - height The height of the box.
    - ∗ **Units:** mm, cm, dm, m, km
    - ∗ **Use:** required
    - ∗ **Range:** $[-MAXFLOAT, MAXFLOAT]$
  - depth The depth of the box.
    - ∗ **Units:** mm, cm, dm, m, km
    - ∗ **Use:** required
    - ∗ **Range:** $[-MAXFLOAT, MAXFLOAT]$
- **SphereAppearance**
  - name
    - ∗ **Use:** optional
    - ∗ **Range:** String
  - radius The radius of the sphere.
    - ∗ **Units:** mm, cm, dm, m, km
    - ∗ **Use:** required
    - ∗ **Range:** $[-MAXFLOAT, MAXFLOAT]$
- **CylinderAppearance**

  - name
    - **Use:** optional
    - **Range:** String
  - height The height of the cylinder.
    - **Units:** mm, cm, dm, m, km
    - **Use:** required
    - **Range:** $[-MAXFLOAT, MAXFLOAT]$
  - radius The radius of the cylinder.
    - **Units:** mm, cm, dm, m, km
    - **Use:** required
    - **Range:** $[-MAXFLOAT, MAXFLOAT]$

- **CapsuleAppearance**
  - name
    - **Use:** optional
    - **Range:** String
  - height The height of the capsule.
    - **Units:** mm, cm, dm, m, km
    - **Use:** required
    - **Range:** $[-MAXFLOAT, MAXFLOAT]$
  - radius The radius of the capsule.
    - **Units:** mm, cm, dm, m, km
    - **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.
    - **Use:** required
    - **Range:** $[-MAXFLOAT, MAXFLOAT]$
  - max The maximal deflection.
    - **Use:** required
    - **Range:** $[-MAXFLOAT, MAXFLOAT]$
  - init The initial deflection.
    - **Default:** 0
    - **Use:** optional
    - **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
    - **Use:** required
    - **Range:** $[-MAXFLOAT, MAXFLOAT]$
  - maxForce The maximum force of this motor.
    - **Units:** N
    - **Use:** required
    - **Range:** $[-MAXFLOAT, MAXFLOAT]$
  - p The p value of the motor's pid interface.
    - **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
    * **Use:** required
    * **Range:** $[-MAXFLOAT, MAXFLOAT]$
  - maxForce The maximum force of this motor.
    * **Units:** N
    * **Use:** required
    * **Range:** $[-MAXFLOAT, MAXFLOAT]$

A.4.18 surfaceClass

- **Surface** Defines the appearance of a surface.
  - diffuseColor The diffuse color, see A.4.23.
  - ambientColor The ambient color of the light, see A.4.23.
  - specularColor The specular color, see A.4.23.
  - emissionColor The emitted color of the light, see A.4.23.
  - shininess The shininess value.
    * **Default:** $0.f$
    * **Use:** optional
    * **Range:** $[0.f, 128.f]$
  - diffuseTexture A texture.
    * **Use:** optional
    * **Range:** String

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.
    - **Use:** required
    - **Range:** Integer > 0
  - imageHeight The height of the camera image.
    - **Use:** required
    - **Range:** Integer > 0
  - angleX Opening angle in x.
    - **Units:** degree, radian
    - **Use:** required
    - **Range:** Float > 0
  - angleY Opening angle in y.
    - **Units:** degree, radian
    - **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
    - **Use:** required
    - **Range:** Float > 0
  - angleY The maximum angle in y-direction the ray of the sensor can spread.
    - **Units:** degree, radian
    - **Use:** required
    - **Range:** Float > 0

- **DepthImageSensor**
  - name
    - **Use:** optional
    - **Range:** String
  - imageWidth The width of the image.
    - **Use:** required
    - **Range:** Integer > 0
  - imageHeight The height of the image.
    - **Default:** 1
    - **Use:** optional
    - **Range:** Integer > 0
  - angleX
    - **Units:** degree, radian
    - **Use:** required
    - **Range:** Float > 0
  - angleY
    - **Units:** degree, radian
    - **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:** perspective
    - **Use:** optional
    - **Range:** perspective, spheric

A.4.21 userInputClass

- **UserInput** Combines an actuator and a sensor. The values set for the actuator are directly returned by the sensor. Using the actuator view, this allows to feed user input to the controller.
  - name Name of the user input.
    - **Use:** optional
    - **Range:** String
  - type The kind of data for which user input is provided.
    - **Default:** length
    - **Use:** optional
    - **Range:** angle, angularVelocity, length, velocity, acceleration
  - min The minimum value that can be set.
    - **Units:** Units matching the type
    - **Use:** required
    - **Range:** $[-MAXFLOAT, MAXFLOAT]$
  - max The maximum value that can be set.
    - **Units:** Units matching the type
    - **Use:** required
    - **Range:** $[-MAXFLOAT, MAXFLOAT]$
  - default The value returned by the sensor if no value is set in the actuator view.
    - **Units:** Units matching the type
    - **Default:** 0
    - **Use:** optional
    - **Range:** $[-MAXFLOAT, MAXFLOAT]$

A.4.22 lightClass

- **Light** Definition of a light source.
  - diffuseColor Diffuse color definition, see A.4.23
  - ambientColor Ambient color definition, see A.4.23
  - specularColor Specular color definition, see A.4.23
  - 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.4.23 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

Camera Kernel Module

The default kernel module for the NAO cameras bundled by SoftBank Robotics has some disadvantages. In low light situations the camera image can turn dark. This happens frequently in duelling situations for the lower camera, when the robots stand close to one another and produce a lot of shadows. Additionally, when playing with auto exposure, the power line frequency needs to be known to the camera in order to avoid flicker from lights. Apart from those problem fixes two new features where added to the driver last year. A gamma correction setting; and most importantly a one time auto white balance. This years new features include improvements to the auto white balance. For a detailed description on all settings, and on how to build the module, please read the *README.md* of the BKernel repository on GitHub at `https://github.com/bhuman/BKernel`.