B-Human

Team Report and Code Release 2011

Thomas Röfer[1], Tim Laue[1], Judith Müller[1], Alexander Fabisch[2], Fynn Feldpausch[2], Katharina Gillmann[2], Colin Graf[2], Thijs Jeffry de Haas[2], Alexander Härtl[2], Arne Humann[2], Daniel Honsel[2], Philipp Kastner[2], Tobias Kastner[2], Carsten Könemann[2], Benjamin Markowsky[2], Ole Jan Lars Riemann[2], Felix Wenk[2]

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

Revision: November 3, 2011

Contents

Chapter 1

Introduction

This document is a continuation of previous B-Human team reports, namely [30], [31], and [32] and still contains text provided by the authors of those reports. We do not list them as authors of the current report anymore, because this would result in a longer list of authors in every year.

1.1 About Us

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

The senior team members have also been part of a number of other successes, such as winning the RoboCup World Championship three times with the GermanTeam (2004, 2005, and 2008), winning the RoboCup German Open also three times (2007 and 2008 with the GermanTeam, 2008 with B-Smart), and winning the Four-Legged League Technical Challenge twice (2003 and 2007 with the GermanTeam).

In parallel to these activities, B-Human started as a part of the joint team BreDoBrothers that has been a cooperation of the Technische Universität Dortmund and the Universität Bremen. The team participated in the Humanoid League in RoboCup 2006. The software was based on previous works of the GermanTeam [14]. This team was split into two single Humanoid teams, because of difficulties in developing and maintaining a robust robot platform across two locations. The DoH!Bots from Dortmund as well as B-Human from Bremen participated in RoboCup 2007; B-Human reached the quarter finals and was undefeated during round robin. In addition to the participation in the Humanoid League at the RoboCup 2008, B-Human also attended a new cooperation with the Technische Universität Dortmund. Hence, B-Human took part in the Two-Legged Competition of the Standard Platform League as part of the team BreDoBrothers, who reached the quarter finals. After the RoboCup 2008, we concentrated our work exclusively on the Two-Legged SPL. By integrating all the students of the Humanoid League team B-Human, the BreDoBrothers would have had more than thirty members. Therefore we decided to end the cooperation by mutual agreement to facilitate a better workflow and work-sharing.

In 2009, we participated in the RoboCup German Open Standard Platform League and won the competition. We scored 27 goals and received none in five matches against different teams. Furthermore, B-Human took part in the RoboCup World Championship and won the compe-

Figure 1.1: The majority of the team members at the RoboCup 2011.

tition, achieving a goal ratio of 64:1. In addition, we could also win first place in the technical challenge, shared with Nao Team HTWK from Leipzig. We repeated our successes in 2010 and won the German Open with a goal ratio of 54:2 as well as the RoboCup with an overall goal ratio of 65:3. In 2011, we won both competitions again with the goal ratios 57:2 and 62:1.

The current team consists of the following persons:

Students. Alexander Fabisch, Arne Humann, Benjamin Markowsky, Carsten Könemann, Daniel Honsel, Felix Wenk, Fynn Feldpausch, Martin Ring, Ole Jan Lars Riemann, Philipp Kastner, Thomas Liebschwager, Tobias Kastner.

Senior Students. Alexander Härtl, Colin Graf, Katharina Gillmann, Thijs Jeffry de Haas.

Staff. Judith Müller, Thomas Röfer (team leader), Tim Laue.

1.2 About the Document

As we wanted to revive the tradition of an annual code release three years ago, it is obligatory for us to sustain it this year. This document, which is partially based on last year's code release [32], gives a survey about the evolved system we used at RoboCup 2011. The changes made to the code that was used at RoboCup 2010 are shortly enumerated in Section 1.3.

Chapter 2 starts with a short introduction to the software required, as well as an explanation of how to run the *Nao* with our software. Chapter 3 gives an introduction to the software framework. Chapter 4 deals with the cognition system and gives an overview of our perception

and modeling components. In Chapter 5, we describe our sensor reading processing and motion generation approaches. Chapter 6 gives an overview of the behavior that was used at the RoboCup 2011[1] and how to create a new behavior. Chapter 7 explains our presentation in the Open Challenge. Finally, Chapter 8 describes the usage of SimRobot, the program that is both used as simulator and as debugging front end when controlling real robots. After acknowledging people who are important for this work in Chapter 9, the appendices A and B provide details about B-Human's configuration files and SimRobot's scene description files.

1.3 Changes Since 2010

The major changes made since RoboCup 2010 are described in the following sections:

2.2.3.1 Support for Mac OS X
Developing on Mac OS X using Xcode 4 is now fully supported.

2.7 B-Human User Shell
We implemented a new tool that accelerates configuration and deployment.

3.4.3–3.4.5 Streaming and Configuration Files
All datatypes can now be streamed using a single macro. A new macro `ENUM` was introduced that automatically generates string representations for all enumeration values defined. In addition, all configuration files now use a more type-safe and readable format that makes use of the datatype specifications acquired by the streaming architecture. This format is also used by the *get*/*set* commands in the SimRobot console.

4.1.1.1 Camera Matrix
The process of camera calibration has been enhanced and simplified.

4.1.2 Using Both Cameras
We can now switch between both cameras while playing.

4.1.4 Color Table
The color table creation has been improved and a documentation of color tables has been added.

4.1.5.7 Foot Perceptor
We added a method to locate nearby robot feet. The perceptions are used to improve tackling behavior.

4.2.11 Ground Truth
The ground truth system can now be synchronized with the robot's estimated model.

4.2.4 Robot Tracking
The robot tracking algorithm is now based on an unscented Kalman filter.

4.2.9 Field Coverage
Patrol targets are now generated from a field observation module.

4.2.10 CombinedWorldModel
A combined word model consisting of an estimated global ball state and positions of all players (own and opponent) was developed, that is mainly used for path planning.

[1] The code release only contains a striker with a simple "go-to-ball-and-kick" behavior that is not described in this document.

5.2.1 Walking

The walking engine has been revised. Now, it uses a refined balancing mechanism and can perform minor in-walk kicks.

6.3 Behavior of 2011

The behavior was improved to be able to deal with four robots on the field (6.3.8). Additionally, we implemented a method to customize kickoff situations (6.3.7) and took further steps on improving intentional passes.

6.3.6 Path Planner

A path planner for faster obstacle avoidance was developed.

6.5 *b-script* Behavior Engine

A behavior engine based on *b-script* was implemented and used for the penalty shootout behavior.

8.2/B SimRobot Architecture

The architecture of SimRobot has changed and a new simulation core (SimRobotCore2) has been developed. As a result, the scene description language has changed significantly.

8.7 Logging

Logging has been enhanced and is now documented.

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: unpacking the source code, compiling the code using Visual Studio 2010, Xcode 4, or the Linux shell, setting up the Nao, copying the files to the robot and starting the software.

2.1 Unpacking

The code release package should be unpacked (for example with `tar -xkf bhuman11_coderelease.tar.bz2`) to a location, whose path must *not* contain any whitespaces. Using Windows, it is recommended to unpack the archive using the *tar* program that is delivered with Cygwin (cf. Sect. 2.2.2.1). To use the B-Human software on a Nao, some libraries and headers from the Aldebaran SDK are required. These libraries can be installed automatically with the *Install/alcommonInstall* script. It can be executed with Cygwin and the shells of Linux and Mac OS X, respectively (cf. Sect. 2.2.2.1, Sect. 2.2.3.1, and Sect. 2.2.4.1). Please ensure that the SDK package also is located in a path which does not contain any whitespaces.

After the unpacking process, the chosen location 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 source code. It is initially missing and will be created by the build system. If desired, the user can remove it manually.

Config contains configuration files used to configure the B-Human software. A more thorough description of the individual files can be found below in Sect. 2.9 and Appx. A.

Doc contains some further documentation resources and is the target directory for the compiled documentation of the simulator and the behavior, which can be created by using the *BehaviorDoc* and the *SimulatorDoc* components (cf. Sect. 2.8).

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

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

Src contains the source code of the B-Human software.

Util contains auxiliary and third party libraries and tools (cf. Chap. 9).

2.2 Building the Code

2.2.1 Project Generation

In the *Make/<OS/IDE>* directory there is a script called *generate*. This script generates the platform or IDE specific files which are needed to compile the components. It collects all the source files, headers and other resources if needed and packs them into a solution matching your system (i. e. Visual Studio 2010 projects and a solution file for Windows, make files for Linux, and an Xcode project for Mac OS X). *generate* has to be called, before any IDE can be opened or any build process can be started. It has to be called again whenever files are added or removed from the project.

2.2.2 Visual Studio 2010 on Windows

2.2.2.1 Required Software

- Visual Studio 2010 SP1
- Cygwin – 1.7 with the following additional packages: make, ruby, rsync, openssh, libxml2, libxslt. Add the *...\cygwin\bin* directory to the beginning of the PATH environment variable (before the Windows system directory, since there are some commands that have the same names but work differently). (`http://www.cygwin.com`)
- gcc, glibc – Linux cross compiler for Cygwin, download from the B-Human website[1], use a Cygwin shell to extract in order to keep symbolic links. The content of the file should be placed in the Cygwin root. This cross compiler package was built using crosstool-NG (`http://crosstool-ng.org/`). To use this cross compiler together with our software, we placed the needed boost and python include files into the include directory.
- alcommon – For the extraction of the required alcommon library and compatible boost headers from the *Nao SDK release v1.10.52 linux* (*naoqi-sdk-1.10.52-linux.tar.gz*) the script *Install/alcommonInstall* 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 `http://users.aldebaran-robotics.com` (account required). Please note that this package is only required to compile the code for the actual Nao robot.

2.2.2.2 Compiling

Generate the Visual Studio 2010 project files using the script *Make/VS2010/generate.bat* and open the solution *Make/VS2010/BHuman.sln* in Visual Studio. Visual Studio will then list all the components (cf. Sect. 2.8) needed to compile the source code in the "Solution Explorer". Select the desired configuration (cf. Sect. 2.8, for now *Develop* would be a good choice) and built 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. 2.7). You also may select *SimRobot* or *Utils/bush* as "StartUp Project".

[1] `http://www.b-human.de/downloads/crosscompiler2011.tar.bz2`

2.2.3 Xcode 4 on Mac OS X

2.2.3.1 Required Software

The following components are required and should be installed at their default locations:

- Xcode 4 or above from the Mac App Store
- Qt libraries 4.7.3 or above for Mac (`http://qt.nokia.com/downloads/qt-for-open-source-cpp-development-on-mac-os-x`)
- Cross compiler gcc 4.5.2 running on Mac OS X and compiling for Linux (`http://crossgcc.rts-software.org/download/gcc-4.5.2-for-linux32-linux64/gcc-4.5.2-for-linux32.dmg`). Please note that the binaries produced by this cross compiler are not compatible with the runtime library on the Nao. Therefore, the runtime library is linked statically, which results in relatively large binaries.
- Xcode-Plugin for Cross Compiler gcc 4.5.2 (`http://www.b-human.de/downloads/GCC%20Linux%204.5.2.xcplugin.zip`). It must be unpacked in */Developer/Library/Xcode/PrivatePlugIns/Xcode3Core.ideplugin/Contents/SharedSupport/Developer/Library/Xcode/Plug-ins*.
- GraphViz 2.26.3 or above (`http://www.graphviz.org/Download_macos.php`).
- Java 6 if running MacOS X 10.7 or later (`http://support.apple.com/kb/DL1421`).
- alcommon – For the extraction of the required alcommon library and compatible boost headers from the *Nao SDK release v1.10.52 linux* (*naoqi-sdk-1.10.52-linux.tar.gz*) the script *Install/alcommonInstall* 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 `http://users.aldebaran-robotics.com` (account required). Please note that this package is only required to compile the code for the actual Nao robot. Also note that *alcommonInstall* 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.

2.2.3.2 Compiling

Generate the Xcode project by double-clicking *Make/MacOS/generate*. Open the Xcode project *Make/MacOS/B-Human.xcodeproj*. If Xcode suggests to upgrade the project file by replacing the *gcc* compiler with the *llvm* compiler, reject the suggestion, because the *gcc* is the cross-compiler that is still needed. 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[2] in all four configurations (cf. Sect. 2.8). For both targets, *Develop* is a good choice. In addition, the B-Human User Shell *bush* can be built as well as the documentation for the behavior and the robot code in the simulator[3]. It is advisable to delete all the schemes that are automatically created by Xcode, i. e. all non-shared ones. When SimRobot is compiled on Mac OS X 10.7, Qt currently generates a lot of compatibility warnings. They are best suppressed by deleting the line that generates them from the file *qglobal.h* in the Qt installation.

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. 2.4).[4] If the *login* script was used before to login to a

[2]Note that the cross compiler builds 32 bit code, although the scheme says "My Mac 64-bit".
[3]Building the simulator documentation takes very long.
[4]Before you can do that, you have to setup the Nao first (cf. Sect. 2.3).

Nao, the IP address used will be provided as default. In addition, the option `-r` is provided by default that will restart the software on the Nao after it was deployed. Since *copyfiles* only copies changed files by default, it is advisable to initiate a full copy with the option `-d` when starting to work with a Nao that has been used by someone else before.

2.2.4 Linux Shell

2.2.4.1 Required Software

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

- g++-4.4, make
- libqt4-dev, qt4-dev-tools, libphonon-dev – 4.3 or above (`http://qt.nokia.com`)
- ruby – 1.8
- doxygen, graphviz, xsltproc – For compiling the documentation.
- openssh-client, rsync – For deploying compiled code to the Nao.
- alcommon – For the extraction of the required alcommon library and compatible boost headers from the *Nao SDK release v1.10.52 linux* (*naoqi-sdk-1.10.52-linux.tar.gz*) the script *Install/alcommonInstall* 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 `http://users.aldebaran-robotics.com` (account required). Please note that this package is only required to compile the code for the actual Nao robot.

On Ubuntu Natty Narwhal you just need this command line to install all needed dependencies:

```
sudo apt-get install g++-4.4 make libqt4-dev qt4-dev-tools libphonon-dev \
ruby doxygen graphviz xsltsproc openssh-client rsync
```

2.2.4.2 Compiling

Before you can use any of the Makefiles you have to execute *Make/Linux/generate*.

To compile one of the components described in Section 2.8 (except *Copyfiles*), simply select *Make* as the current working directory and type:

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

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

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

To clean up the whole solution use:

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

As an alternative, there is also support for the integrated development environment Code::Blocks that works similar to Visual Studio for Windows (cf. Sect. 2.2.2.2). To use it, execute *Make/CodeBlocks/generate* and open the *B-Human.workspace* afterwards.

2.3 Setting Up the Nao

2.3.1 Requirements

Setting up the Nao is only possible from a Linux OS. First of all, download the *OS image v1.10.52* that is available in the download area of `http://users.aldebaran-robotics.com`. Unpack this file and move the extracted image to *Install/images*. To save space it is possible to compress the image with bzip2. After that, there should be a file *nao-system-image-robocup-1.10.52.ext3* or *nao-system-image-robocup-1.10.52.ext3.bz2*. If your image file has a different name, you have the choice to change the *imageName* variable in line 8 of *flashAndInstall.sh* or to use the *-i* option with the name of your image file as argument when calling the install script. Note that the only supported compression is bzip2, which is only detected if the image file has the bz2 extension. All other file extensions are ignored and the image file is considered as uncompressed image file.

The only supported NaoQi and OS image version is 1.10.52.

To use the scripts in the directory *Install* the following tools are needed[5]:

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

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

2.3.2 Creating a Robot Configuration

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

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

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

createNewRobot.sh -t 3 -r 25 Penny

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

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

2.3.3 Wireless Configuration

To use the wireless interface of the Nao, you need to create a *wpa_supplicant.conf* file. The easiest way is to copy the file *Install/files/wpa_supplicant.conf_template* to *Install/-*

[5]The tools should be present on every Linux distribution. If not, the following command should help: *sudo apt-get install sed e2fsprogs util-linux mount grep gawk patch bzip2 tar coreutils dosfstools bash*

file/wpa_supplicant.conf_default and change it in the way needed for your wireless network. The name of the new configuration must be *wpa_supplicant.conf_<suffix>* in order to use this file with our install script. To use a suffix other than *default*, change the variable *wlanConfigSuffix* in line 16 of *Install/flashAndInstall.sh* to the chosen suffix or use the option *-W* of the install script with the suffix as argument. In order to manage the wireless configurations of your robots, for example to copy new or updated configurations or switching the active configuration, you can use the scripts *updateWirelessConfig.sh* and *switchActiveWirelessConfiguration.sh*. *updateWirelessConfig.sh* copies all *wpa_supplicant.conf* files found to all known and reachable robots. For *switchActiveWirelessConfiguration.sh*, help is available using the option *-h*. This script activates the configuration specified by the argument on all known and reachable robots. While switching the active wireless configuration you should be connected via a cable connection. Switching the wireless configuration via a wireless connection is untested.

A robot is known if an interfaces file exists for that robot. Both scripts use all IP addresses found in all interfaces files except the *_interfaces_template_* file to connect to the robots. If you are connected to a robot via a cable and a wireless connection, all changes are done twice. As an alternative to these scripts you can use the B-Human User Shell described in Sect. 2.7 to manage the wireless configurations.

2.3.4 Setup

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

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

The install script writes the OS image to the flash drive and formats the *userdata* partition of the drive. After that, the needed configuration files for the robot and all files needed to run *copyfiles* for this robot and to use it with the B-Human software are copied to the drive. If the script terminates without any error message you can remove the flash drive from your computer and plug it into the Nao. Notice that the script assumes that you have placed an OS image named *nao-system-image-robocup-1.10.52.ext3.bz2* and an appropriate partition table with the same name but a *.parttable* extension in *Install/images* if you do not specify an alternative image file with *-i*.

Start your Nao and wait until the boot finished. Connect via SSH to the Nao and log in as root with the password *cr2011*. Start *./phase2.sh* and reboot the Nao. After the reboot the Nao is ready to be used with the B-Human software.

2.4 Copying the Compiled Code

2.4.1 Using copyfiles

The tool *copyfiles* is used to copy compiled code and configuration files to the *Nao*.

Running Windows as well as on MacOS you have two possibilities to use *copyfiles*. On the one

hand, in Visual Studio you can do that by "building" the tool *copyfiles*. *copyfiles* can be built in all configurations. If the code is not up-to-date in the desired configuration, it will be built. After a successful build, you will be prompted for entering the parameters described below. On the Mac, a successful build for the Nao always ends with a dialog asking for *copyfiles*' command line options. On the other hand you can just execute the script *Make/VS2010/copyfiles.cmd* or *Make/MacOS/copyfiles* at the command prompt, respectively.

The second option is also the one Linux users have to use, in this case the script *Make/Linux/copyfiles*. Actually, also all other ways described to deploy files to the Nao finally result in executing this script.

copyfiles requires two mandatory parameters. First, the configuration the code was compiled with (*Debug*, *Optimized*, *Operate* or *Release*)[6], and second, the IP address of the robot. To adjust the desired settings, it is possible to set the following optional parameters:

Option	Description
-l <location>	Sets the location, replacing the value in the *settings.cfg*.
-t <color>	Sets the team color to *blue* or *red*, replacing the value in the *settings.cfg*.
-p <number>	Sets the player number, replacing the value in the *settings.cfg*.
-d	Deletes the target directory before the compiled code is copied.
-r	Restarts bhumand, and if -d is active, naoqid, too.
-m n <ip>	Copies to IP address <ip> and sets the player number to *n*.
-wc	Compiles also under Windows if the binaries are outdated.
-nc	Never compiles, even if the binaries are outdated.

Possible calls could be:

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

The destination directory on the robot is */media/userdata/Config*. Alternatively the B-Human User Shell (cf. Sect. 2.7) can be used to copy the compiled code to several robots at once.

2.5 Working with the Nao

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

To connect to the Nao, the subdirectories of *Make* contain a *login* script for each supported platform. The only parameter of those scripts is the IP address of the robot to login. The scripts automatically use the appropriate SSH key to login. In addition, they write the IP address specified to the files *Config/Scenes/connect.con* and *Config/Scenes/connectWithPuppet.con*. Thus a later use of the SimRobot scenes *RemoteRobot.ros2* and *BikeSceneRemoteWithPuppet.ros2* will automatically connect to the same robot (cf. next section). On the Mac, the IP address is also the default address for deployment in Xcode.

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

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

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

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

./naoqid start|stop|restart starts, stops or restarts *NaoQi*. After updating *libbhuman* with *copyfiles* *NaoQi* needs a restart. *copyfiles*' option *-r -d* accomplishes this automatically, but then, the deployment will take longer than it has to.

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

./bhumand start|stop|restart starts, stops or restarts the *bhuman* executable. After uploading files with *Copyfiles* *bhuman* must be restarted. *copyfiles*' option *-r* accomplishes this automatically, but prevents you from seeing any of *bhuman*'s console outputs.

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

To shutdown Nao, execute *halt* in Nao's command shell. If the B-Human software is running, this can also be done by pressing the chest button longer than three seconds.

2.6 Starting SimRobot

On Windows and Mac OS X, SimRobot can either be started from the development environment, or by starting a scene description file in *Config/Scenes*[7]. 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 Code::Blocks, 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 *information views*.

To connect to a real Nao, enter its IP address in the file *Config/Scenes/connect.con* on the PC if you have not used one of the login scripts described in the previous section. Afterwards, start the simulation scene *Config/Scenes/RemoteRobot.ros2* (cf. Sect. 8.9.3). In a remote connection, the simulation scene is usually empty. Therefore, it is not necessary to open a scene view. A remote connection to the Nao is only possible if the code running on the Nao was compiled in a configuration other than *Operate*.

See Chapter 8 for more detailed information about SimRobot.

[7] 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 Mac OS X, starting a scene description file bears the risk of executing a different version of SimRobot than the one that was just compiled.

Entry	Description	
number	The team number.	
port	The port which is used for team communication messages.	*Optional*. If this value is omitted, the port is generated from the team number.
location	The location which should be used by the software (cf. section 2.9).	*Optional*. It is set to `Default` if it is omitted
color	The team color in the first half.	*optional* It is only required if no game controller is running which overwrites the team color.
wlanConfig	The name of the configuration file which should be used to configure the wireless interface of the robots.	*Optional*. If it is omitted, it is set to `projektraum` as specified in *Config/bush.cfg*
buildConfig	The name of the configuration which should be used to build the Nao code (cf. section 2.8).	*Optional*. If it is omitted, it is set to `Develop` as specified in *Config/bush.cfg*
players	The list of robots the team consists of. Each of the names has to be defined in a file *robot.cfg*.	

Table 2.1: Configuration variables in the file *Config/teams.cfg*

2.7 B-Human User Shell

The B-Human User Shell (bush) accelerates and simplifies the deployment of the 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.

2.7.1 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>/robot.cfg* which defines the name of the robot and how it can be reached by the bush.[8] 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 2.1.

The bush implements some useful commands which perform tasks which were implemented in several scripts before. The most important ones are listed and explained below. Some of these commands require that one or more robots are selected in the upper half of the bush window. The robots can either be selected by the mouse or with the keys *F1* to *F8*.

[8] The configuration file is created by *createNewRobot.sh* described in Sect. 2.3.2.

2.7.2 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. It does not keep the project files up-to-date, i. e. call one of the *generate* scripts. Instead, you have to do it manually (cf. Sect. 2.2).

After all the files required by the Nao are copied, the *deploy* command calls the *updateSettings* command, which generates a new *settings.cfg* according to the configuration, tracked by the bush for every of the selected robots. Of course the *updateSettings* command can also be called without the *deploy* in order to reconfigure several robots without the need of updating the binaries. After updating the *settings.cfg* file, the bhuman software has to be restarted for changes to take effect. This can easily be done with the command *restart*. If it is called without any parameter, it restarts only the bhuman software but it can also be used to restart naoqi, 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 robots you can use the command *show*, which knows most of the files located on the robots and can help you to find the desired files with tab completion.

2.7.3 Managing Multiple Wireless Configurations

Since the robot soccer competition generally takes place on more than just a single field and normally each field has its own WLAN access point, the robots have to deal with multiple configuration files for their wireless interfaces. The bush helps to manage these various files with the commands *updateWireless* and *changeWireless*. The first command can be used to copy all new configuration files to the Nao robot, while the latter activates a specific one on the robot. Which configuration is used can be specified in the first argument of *changeWireless*. If the argument is omitted, the wireless configuration of the selected team is used.

2.7.4 Substituting Damaged 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 drap&drop at the appropriate position. Since this view only supports eight 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.

2.7.5 Creating Color Tables

The bush has some commands that simplify the creation of color tables:

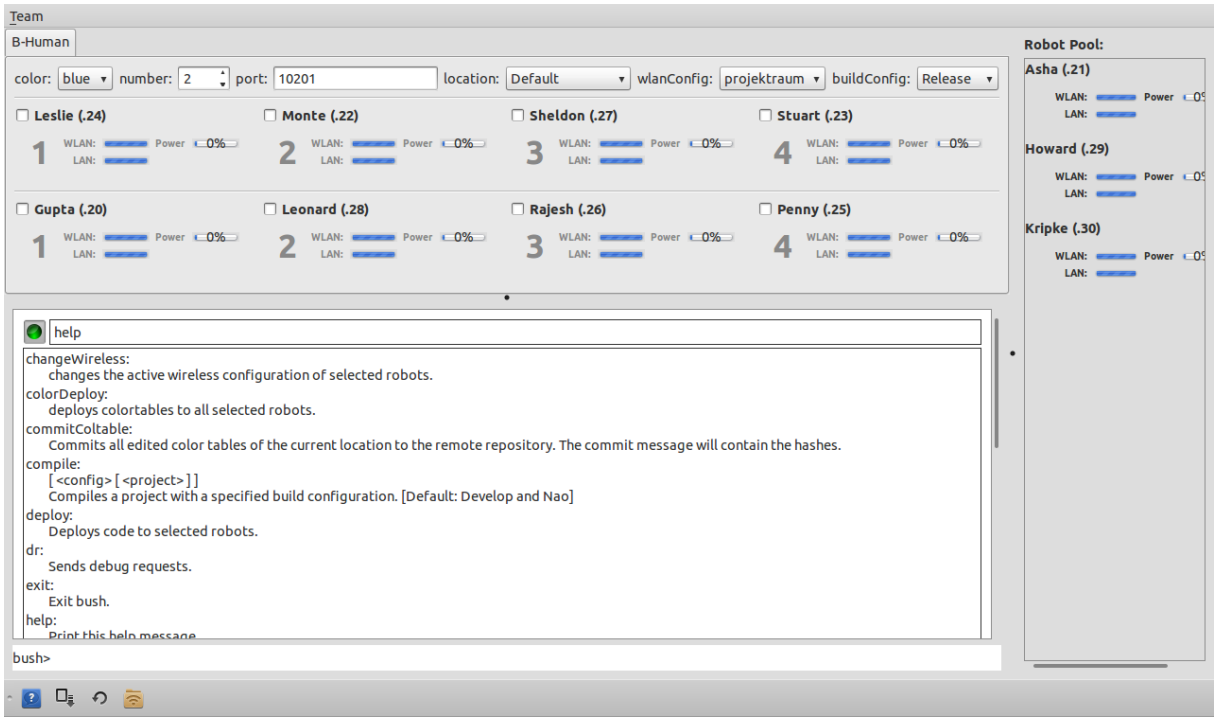


Figure 2.1: bush on Gnome.

- In order to deploy the B-Human code to the selected Naos, to restart bhumand on the Naos, and to connect to the Naos with the simulator, you just have to type *prepareColtable*. Training will automatically be enabled in the simulator scene loaded (cf. Sect. 4.1.4).

- You can copy all color tables and training sets to the selected Naos with the command *colorDeploy <location>*. Remember to load the new color table, e. g. by restarting bhumand with the command *restart*.

- You can push your locally modified color tables to your remote git repository by executing *commitColtable*. If you use another version control system, you will have to adapt this command.

2.7.6 Monitoring Robots

The bush displays some information about the robots' states as you can see in Figure 2.1: wireless connection pings, wired connection pings, and remaining battery. But you cannot always rely on this information, because it is only collected properly if the robots are reachable and the *bhuman* software is running on the robot.

The maximal representable ping is 2000 ms. So, if the connection to the robot is good, the bar will almost not be visible. If the connection is completely lost, the bar will not be updated.

The power bar shows the remaining battery. bush reads this information from the team communication (representation *RobotHealth*). The robot has to be at least in the state *initial* in order to send team communication messages, i. e. it must stand. The power bar will freeze until it receives the next message.

2.7.7 Adding New Commands

The architecture of the bush was designed to make the implementation of new commands as simple as possible. All current command implementations are located in *Src/Utils/bush/cmds*. There you will find *CommandNameCmd.h.template* and *CommandNameCmd.cpp.template*. In order to implement a new command <CommandName>, just follow these instructions from *CommandNameCmd.cpp.template*:

1. Copy *CommandNameCmd.h.template* to *<CommandName>Cmd.h* and *CommandNameCmd.cpp.template* to *<CommandName>Cmd.cpp*.
2. Replace all occurrences of `CommandName` by <`CommandName`> in both files.
3. Implement `getName`, which is used to identify the command. The returned string has to be unique and is not allowed to contain whitespaces.
4. Implement `getDescription` and `complete` if you want, or remove the declarations and definitions and
5. Choose one of the following options:
   (a) In order to implement a command without a robot involved: let the command inherit from `CommandAdapter` and
      - implement `execute` and
      - delete declarations and definitions for `preExecution`, `perRobotExecution` and `postExecution`.

   (b) In order to implement a command for every activated and reachable robot: let the command inherit from `RobotCommand` and
      - implement `preExecution`, `perTask` and `postExecution` and
      - delete declarations and definitions for `execute`.

2.8 Components and Configurations

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

SimRobot is the executable simulator (cf. Chapter 8) for running and controlling the B-Human robot code. It dynamically links against the components *SimRobotCore2*, *SimRobotEditor*, *SimRobotHelp*, *SimulatedNao*, and some third-party libraries. It also depends on the components *Behavior*, *b-scriptBehaviorSim*[9], and *SpecialActions* the results of which are loaded by the robot code. SimRobot is compilable in *Release*, *Develop* and *Debug* configurations. All these configurations contain debug code but *Release* performs some optimizations and strips debug symbols (Linux). *Develop* produces debuggable robot code while linking against non-debuggable *Release* libraries.

SimRobotCore2 is a shared library that contains the simulation engine of SimRobot. It is compilable with or without debug symbols (configurations *Release* and *Debug*).

[9] Not on Windows.

SimRobotEditor is a shared library that contains the editor widget of the simulator. It is compilable with or without debug symbols (configurations *Release* and *Debug*).

SimRobotHelp is a shared library that contains the help widget of the Simulator. It is compilable with or without debug symbols (configurations *Release* and *Debug*).

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. The library is available in the configurations *Release* and *Debug*.

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

Nao cross-compiles the B-Human executable for the Nao. It is available in *Operate*, *Release*, *Develop*, and *Debug* configurations, where *Operate* produces "game code" without any support for debugging. The configuration *Develop* produces optimized code, but still supports all debugging techniques described in Sect. 3.6. If you want to disable assertions as in *Operate* but enable debugging support, *Release* can be used.

SimulatedNao compiles the B-Human code for the simulator as shared library. It uses the same code as *Nao*. It is statically linked against the *Controller*.

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

	without assertions (NDEBUG)	without debug macros (RELEASE)	debug symbols (compiler flags)	debug libs[1] (_DEBUG, compiler flags)	optimizations (compiler flags)	assertion tracing (WITH_TRACE ASSERTIONS)
Operate [2] [3]						
> Nao	✓	✓	×	×	✓	×
Release						
> Nao	✓	×	×	×	✓	×
> SimulatedNao	✓	×	×	×	✓	×
Develop [3]						
> Nao	×	×	×	×	✓	×
> SimulatedNao	×	×	✓	×	×	×
Debug						
> Nao	×	×	✓	✓	×	×
> SimulatedNao	×	×	✓	✓	×	×

[1] - on Windows - http://msdn.microsoft.com/en-us/library/0b98s6w8%28VS.80%29.aspx

[2] - for SimulatedNao "Release" will be used

[3] - for SimRobot and Controller "Release" will be used

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

bhbsc, b-script2cpp, b-scriptBehavior, b-scriptBehaviorSim [10] are the components which are used to embed the behavior written in b-script (*.bs* files) into our system (cf. Sect. 6.5).

SpecialActions compiles motion patterns (*.mof* files) into an internal format (cf. Sect. 5.2.3) using the *URC*.

URC stands for *Universal Resource Compiler* and is a small tool for automatic generation of some *.xabsl* files (cf. Sect. 6.1) and for compiling *special actions* (cf. Sect. 5.2.3).

copyfiles is a tool for copying compiled code to the robot. For a more detailed explanation see Sect. 2.4.

BehaviorDoc is a tool for creating the documentation of the behavior. The results will be located in *Doc/Reference/BH2011BehaviorControl*.

[10] Only available on Mac OS X and Linux. On Windows, the behavior is interpreted at runtime instead.

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

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. All these files, which are used by the software[11] are located below the directory *Config*.

Besides the global configuration files there are files which are specific for a robot. These files are located in *Config/Robots/<robotName>* where *<robotName>* is the name of a specific robot. They are only taken into account if the name of the directory matches the name of the robot where the code is executed on.

In addition, there are configuration files that depend on the current *location*, e. g. "RoboCup2011" or "OurLab". Locations can also be used to configure robots for a different task, e. g. 'ÒpenChallenge". The current location is set in the file *Config/settings.cfg*.

To handle all these different configuration files there are fall-back rules that are applied if a requested configuration file is not found. If a configuration file is robot-specific, its path is expanded using the method `Settings::expandRobotFilename`. In that case, the search sequence for the file is:

- *Robots/<robotName>/<filename>*
- *Robots/Default/<filename>*

The name of a location-specific file is expanded using `Settings::expandLocationFilename`. In that case, the following paths are tried:

- *Locations/<locationName>/<filename>*
- *Locations/Default/<filename>*

Finally, if a configuration file depends both on location and robot, its name is expanded with `Settings::expandRobotLocationFilename`, and the search path is:

- *Locations/<locationName>/Robots/<robotName>/<filename>*
- *Locations/Default/Robots/<robotName>/<filename>*

For a complete list of the directories and files below the directory *Config* see Appendix A.

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

Chapter 3

Architecture

The B-Human architecture is based on the framework of the GermanTeam 2007 [29], adapted to the Nao. This chapter summarizes the major features of the architecture: binding, processes, modules and representations, communication, and debugging support.

3.1 Binding

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

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

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

- Both frameworks use their own address space without losing their real-time capabilities and without a noticeable reduction of performance. Thus, a malfunction of the process *bhuman* cannot affect *NaoQi* and vice versa.
- Whenever *bhuman* crashes, *libbhuman* is still able to display this malfunction using red blinking eye LEDs and to make the Nao sit down slowly. Therefore, the *bhuman* process uses its own watchdog that can be activated using the -w flag[1] when starting the *bhuman* process. When this flag is set, the process forks itself at the beginning where one instance

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

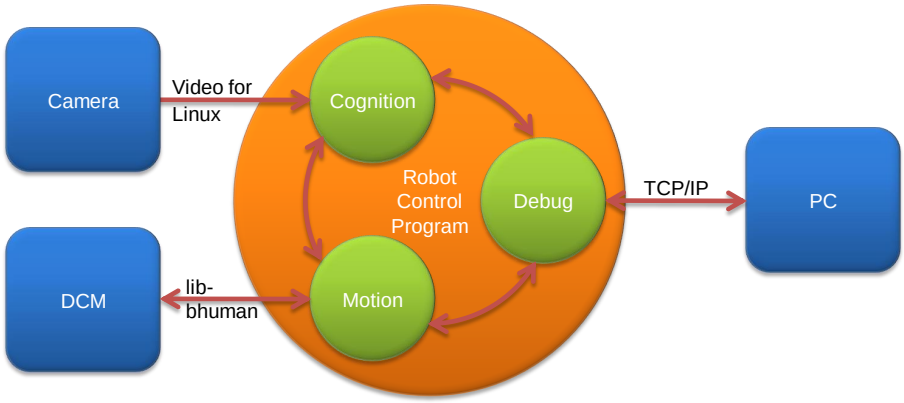


Figure 3.1: The processes used on the Nao

waits for a regular or irregular exit of the other. On an irregular exit the exit code can be written into the shared memory block. The *libbhuman* monitors whether sensor values were handled by the *bhuman* process using the counter of the semaphore. When this counter exceeds a predefined value the error handling code will be initiated. When using release code (cf. Sect. 2.8), the watchdog automatically restarts the *bhuman* process after an irregular exit.

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

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

3.2 Processes

Most robot control programs use concurrent processes. The number of parallel processes is best dictated by external requirements coming from the robot itself or its operating system. The Nao provides images at a frequency of 30 Hz and accepts new joint angles at 100 Hz. Therefore, it makes sense to have two processes running at these frequencies. In addition, the TCP communication with a host PC (for the purpose of debugging) may block while sending data, so it also has to reside in its own process. This results in the three processes *Cognition*, *Motion*, and *Debug* used in the B-Human system (cf. Fig. 3.1). *Cognition* receives camera images from *Video for Linux*, as well as sensor data from the process *Motion*. It processes this data and sends high-level motion commands back to the process *Motion*. This process actually executes these commands by generating the target angles for the 21 joints of the Nao. It sends these target angles through the *libbhuman* to Nao's *Device Communication Manager*, and it receives sensor readings such as the actual joint angles, acceleration and gyro measurements, etc. In addition, *Motion* reports about the motion of the robot, e. g., by providing the results of dead reckoning. The process *Debug* communicates with the host PC. It distributes the data received from it to the other two processes, and it collects the data provided by them and forwards it back to the host machine. It is inactive during actual games.

Processes in the sense of the architecture described can be implemented as actual operating system processes, or as threads. On the Nao and in the simulator, threads are used. In contrast,

in B-Human's team in the Humanoid League, framework processes were mapped to actual processes of the operating system (i. e. Windows CE).

3.3 Modules and Representations

A robot control program usually consists of several modules, each performing a certain task, e. g. image processing, self-localization or walking. Modules require a certain input and produce a certain output (i. e. so-called *representations*). Therefore, they have to be executed in a specific order to make the whole system work. The module framework introduced in [29] simplifies the definition of the interfaces of modules, and automatically determines the sequence in which the modules are executed. It consists of the *blackboard*, the *module definition*, and a visualization component (cf. Sect. 8.4.9).

3.3.1 Blackboard

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

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

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

3.3.2 Module Definition

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

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

class SimpleBallLocator : public SimpleBallLocatorBase
{
  void update(BallModel& ballModel)
  {
    if(theBallPercept.wasSeen)
    {
```

```
      ballModel.position = theBallPercept.position;
      ballModel.wasLastSeen = theFrameInfo.frameTime;
    }
  }
}

MAKE_MODULE(SimpleBallLocator, World Modeling)
```

The module interface defines the name of the module (e. g. `MODULE(SimpleBallLocator)`), the representations that are required to perform its task, and the representations provided by the module. The interface basically creates a base class for the actual module following the naming scheme <`ModuleName`>`Base`. The actual implementation of the module is a class that is derived from that base class. It has read-only access to all the required representations in the blackboard (and only to those), and it must define an `update` method for each representation that is provided. As will be described in Section 3.3.3, modules can expect that all their required representations have been updated before any of their provider methods is called. Finally, the `MAKE_MODULE` statement allows the module to be instantiated. It has a second parameter that defines a category that is used for a more structured visualization of the module configuration (cf. Sect. 8.4.9).

The module definition actually provides a lot of hidden functionality. Each `PROVIDES` statement makes sure that the representation provided can be constructed and deconstructed (remember, the blackboard only contains references), and will be available before it is first used. In addition, representations provided can be sent to other processes, and representations required can be received from other processes. The information that a module has certain requirements and provides certain representations is not only used to generate a base class for that module, but is also available for sorting the providers, and can be requested by a host PC. There it can be used to change the configuration, for visualization (cf. Sect. 8.4.9), and to determine which representations have to be transferred from one process to the other. Please note that the latter information cannot be derived by the processes themselves, because they only know about their own modules, not about the modules defined in other processes. Last but not least, the execution time of each module can be determined (cf. Sect. 3.6.7) and the representations provided can be sent to a host PC or even altered by it.

The latter functionality is achieved by variants of the macro `PROVIDES` that add support for `MODIFY` (cf. Sect. 3.6.6), support for streaming the representation to be recorded in a log file (`OUTPUT`, requires a *message id* with the same name as the representation, cf. Sect. 3.5), and drawing based on a parameterless method *draw* implemented by the representation itself. The maximum version of the macro is `PROVIDES_WITH_MODIFY_AND_OUTPUT_AND_DRAW`. For a reduced functionality, the sections of the name that are not required or not supported can be left out.

Besides the macro `REQUIRES`, there also is the macro `USES(`<`representation`>`)`. `USES` simply gives access to a certain representation, without defining any dependencies. Thereby, a module can access a representation that will be updated later, accessing its state from the previous frame. Hence, `USES` can be used to model cyclic relations in a single process. Since `USES` is actually ignored by the whole framework, it should only be used for this single purpose. In contrast to `REQUIRES`, `USES` does not guarantee that the representation used actually exists. Therefore, the representation should only be accessed when its address is not zero. As a further drawback, the module view (cf. Sect. 8.4.9) does not display `USES` connections.

3.3.3 Configuring Providers

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

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

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

3.3.4 Pseudo-Module *default*

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

3.4 Streams

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

There are reasons not to use the C++ iostreams library. The C++ iostreams library does

not guarantee that data is streamed in a way that it can be read back without any special handling, especially when streaming into and from text files. Another reason not to use the C++ iostreams library is that the structure of the streamed data is only explicitly known in the streaming operators themselves. Hence, exactly those operators have to be used on both sides of a communication, which results in problems regarding different program versions or even the use of different programming languages. Note that there is no support for streaming the data types `long`, `unsigned long`, and `size_t`, because their binary representations have different sizes on different platforms (32/64 bits).

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

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

3.4.1 Streams Available

Currently, the following classes are implemented:

PhysicalOutStream. Abstract class

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

StreamWriter. Abstract class

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

Out. Abstract class

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

OutBinaryMessage. Writing binary into a MessageQueue

OutTextMessage. Writing into a MessageQueue as text

OutTextRawMessage. Writing into a MessageQueue as raw text

OutConfigMap. Writing into a `ConfigMap` (cf. Sect. 3.4.5). This actually is a helper for writing files in `ConfigMap` format with `serialize` methods (cf. Sect. 3.4.3). This stream is not meant to be used directly.

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

InConfigMap. Reading from a `ConfigMap` (cf. Sect. 3.4.5). This actually is a helper for reading from files in `ConfigMap` format with `serialize` methods (cf. Sect. 3.4.3). This stream is not meant to be used directly.

3.4.2 Streaming Data

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

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

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

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

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

```
#include "Tools/Streams/InStreams.h"
// ...
InTextFile stream("MyFile.txt");
int a, d;
double b;
std::string c;
stream >> a >> b >> c >> d;
```

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

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

```
13.14000Hello Dolly
42
```

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

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

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

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

3.4.3 Making Classes Streamable

A class is made streamable by deriving it from the class *Streamable* and implementing the abstract method *serialize(In*, Out*)*. For data types derived from *Streamable* streaming operators are provided, meaning they may be used as any other data type with standard streaming operators implemented. To 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. 3.4.4) 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();
}
```

There are three more macros that are intended to be used for team communication, where a low memory footprint is desirable and a loss in precision is acceptable:

STREAM_COMPRESSED_POSITION(<attribute>) streams a `Vector2<float>` as `Vector2<short>`.

STREAM_COMPRESSED_ANGLE(<attribute>) normalizes an angle to $[-\pi, \pi[$ and streams it as a `char`.

STREAM_AS_UCHAR(<attribute>) streams an attribute as an `unsigned char`. This is particularly useful for small enums.

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

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

3.4.4 Enumerations

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

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

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

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

3.4.5 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 exactly 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:

```
file     ::= map
value    ::= plain
           | '[' list ']'
           | '{' map '}'
```

```
plain    ::= VALUE
list     ::= [ comments value { ',' comments value } [ ',' ] ]
map      ::= { comments key '=' value ';' }
comments ::= { COMMENT }
```

As in C, comments can be denoted either by `//` for a single line or by `/* ... */` for multiple lines. Here is an example:

```
// The pose during kickoff
pose = {
  /* straight
     ahead */
  rotation = 0;
  translation = {x = -100; y = 0;};
};
"kick directions" = ["straight ahead", left, right, /* more often left */ left
    ];
```

There are two possible ways to use configuration maps. On the one hand, they can be used together with the `serialize` methods (cf. Sect. 3.4.3). If a `Streamable` object is read from an `InConfigMap` stream or written to an `OutConfigMap` stream, it is actually read from or stored to a configuration map file, which is far more readable than the flat serialization used for configuration files in previous versions of the B-Human system. Note that files which are read and then written again by a configuration map may change, because the configuration map tries formatting the output as good as possible. If you use `InConfigMap` and `OutConfigMap` to read and write a file, the comments will be lost as well, because comments are not supported by the streaming interface.

On the other hand, configuration maps can be used directly, which is described in the remainder of this section. To construct a configuration map, you have to construct a `ConfigMap` object as the root of the map. Then you can attach values to keys with the `[]`-operator. The following example attaches `1337` to the key `key` and `1338` to the key `first_level.second_level`:

```
ConfigMap map;
map["key"] << 1337;
map["first_level"]["second_level"] << 1338;
```

As you can see, configuration maps support the use of multiple `[]`-operators, but you can also use the key `first_level.second_level` to access the same value, because the period is used as a separator for a complex key. In this example, because of the use of this complex key, the value of `first_level` has to be a complex value too. If such a value does not exist yet, the configuration map will construct one automatically. Entries for plain values will be constructed automatically as well if the key the value is assigned to does not exist. As the `[]`-operator is overloaded for `size_t` as well, you can access values by an index (only supported for `list`-type values, where `map["l"][0]` would access the first element of the `l` value, which has to be of the type `list`). This is also used to detect list indices in complex keys (e. g. `"first.second.1.2"`). As a consequence, keys must not start with a digit.

The more common use of configuration maps is to load one from a file and read the values from the map in the desired order afterwards. Thereto the same `[]`-operator can be used to access the keys, but the `>>`-operator is needed to read the data. For example:

```
ConfigMap map;
if(map.read("file.cfg") == 0)
{
  int i;
  map["i"] >> i;
```

```
}
else // error handling
```

As the desired C++-type is unknown, when the configuration map is read from a file, values are stored as strings internally. Conversion to or from a concrete type is done via streaming operators (`>>` and `<<`), which are implemented for all base types. There also is a special implementation for lists, which can stream their values to vectors of base types.

If you would like to add support for more complex or self-defined data types, you have to define appropriate streaming operators for the configuration map. These are: `ConfigValueType& operator<<(ConfigValueType&, const YourClass&)` and `const ConfigValueType& operator>>(const ConfigValueType&, YourClass&)` where `YourClass` is the data type class the operators shall be implemented for and `ConfigValueType` can be either `ConfigMap`, `ListConfigValue` or `PlainConfigValue`, depending on which type of configuration map value your self-defined data type needs. In order to tell the configuration map which types your operator is using, you have to use the `CONFIGMAP_STREAM_IN_DELCARE`- and the `CONFIGMAP_STREAM_IN`-macro or, analog, the `CONFIGMAP_STREAM_OUT_DELCARE`- and the `CONFIGMAP_STREAM_OUT`-macro.

3.5 Communication

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

3.5.1 Message Queues

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

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

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

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

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

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

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

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

3.5.2 Inter-process Communication

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

All inter-process communication is triple-buffered. Thus, processes never block each other, because they never access the same memory blocks at the same time. In addition, a receiving process always gets the most current version of a packet sent by another process.

3.5.3 Debug Communication

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

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

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

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

The process *Debug* manages the communication of the robot control program with the tools on the PC. For each of the other processes (*Cognition* and *Motion*), it has a sender and a receiver for their debug message queues (cf. Fig. 3.1). Messages that arrive via WLAN from the PC are stored in `debugIn`. The method `Debug::handleMessage(InMessage&)` distributes all messages in `debugIn` to the other processes. The messages received from *Cognition* and *Motion* are stored in `debugOut`. When a WLAN or Ethernet connection is established, they are sent to the PC via TCP/IP. To avoid communication jams, it is possible to send a *QueueFillRequest* to the process *Debug*. The command *qfr* to do so is explained in Section 8.8.3.

3.5.4 Team Communication

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

Similar to debug communication, data can be written to the team communication message queue using the macro `TEAM_OUTPUT(<id>, <format>, <sequence>)`. The macro can only be used in process *Cognition*. In contrast to the debug message queues, the one for team communication is rather small (1384 bytes). So the amount of data written should be kept to a minimum. In addition, team packages are only broadcasted approximately every 100 ms. Hence, and due to the use of UDP in general, data is not guaranteed to reach its intended receivers. The representation *TeamMateData* contains a flag that states whether a team communication package will be sent out in the current frame or not. There also exist a macro `TEAM_OUTPUT_FAST(<id>, <format>, <sequence>)` that uses the information from the blackboard entry `theTeamMateData` to even suppress the actual streaming to the message queue in all frames in which the queue will not be sent anyway. This saves computation time. However consequently, at the location where the macro is used, `theTeamMateData` must be accessible.

3.6 Debugging Support

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

3.6.1 Debug Requests

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

DEBUG_RESPONSE(<id>, <statements>) executes the statements if the debug request

with the name `id` is enabled.

DEBUG_RESPONSE_ONCE(<id>, <statements>) executes the statements *once* when the debug request with the name `id` was just enabled. It is also possible to emulate this behavior with a normal `DEBUG_RESPONSE` by specifying in the debug request that only a single execution is desired.

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

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

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

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

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

This statement sends the text "Hello" if the debug request with the name `"sayHello"` is activated. Please note that only those debug requests are usable that are in the current path of execution. This means that only debug request in those modules can be activated that are currently executed. To determine which debug requests are currently available, a method called *polling* is employed. It asks all debug responses to report the name of the debug request that would activate it. This information is collected and sent to the PC (cf. command *poll* in Sect. 8.8.3).

3.6.2 Debug Images

Debug images are used for low level visualization of image processing debug data. They can either be displayed as background image of an image view (cf. Sect. 8.4.1) or in a color space view (cf. Sect. 8.4.2). Each debug image has an associated textual identifier that allows referring to it during image manipulation, as well as for requesting its creation from the PC. The identifier can be used in a number of macros that are defined in file *Src/Tools/Debugging/DebugImages.h*, and that facilitate the manipulation of the debug image. In contrast to all other debugging features, the textual identifier used is not enclosed in double quotes. It must only be comprised of characters that are legal in C++ identifiers.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

3.6.3 Debug Drawings

Debug drawings provide a virtual 2-D drawing paper and a number of drawing primitives, as well as mechanisms for requesting, sending, and drawing these primitives to the screen of the PC. In contrast to debug images, which are raster-based, debug drawings are vector-based, i. e., they store drawing instructions instead of a rasterized image. Each drawing has an identifier and an associated type that enables the application on the PC to render the drawing to the right kind of drawing paper. In the B-Human system, two standard drawing papers are provided, called `drawingOnImage` and `drawingOnField`. This refers to the two standard applications of

debug drawings, namely drawing in the system of coordinates of an image and drawing in the system of coordinates of the field. Hence, all debug drawings of the type `drawingOnImage` can be displayed in an image view (cf. Sect. 8.4.1) and all drawings of type `drawingOnField` can be rendered into a field view (cf. Sect. 8.4.3).

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

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

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

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

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

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

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

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

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

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

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

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

3.6.4 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 initialized 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 (cf. Appx. B). Note that drawings directly attached to hinges will be drawn relative to the base of the hinge, not relative to the moving part. Drawings of the type `field` are drawn relative to the center of the field, whereas drawings of the type `robot` are drawn relative to the origin of the robot according to Aldebaran's documentation. B-Human uses a different position in the robot as origin, i. e. the middle between the two hip joints. An object called "origin" has been added to the Nao simulation model at that position. It is often used as reference frame for 3-D debug drawings in the current code.

The 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:

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

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

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

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

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

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

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

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

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

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

Watch the result in the scene view of SimRobot (cf. Sect. 8.3) by sending a debug request (cf. Sect. 8.8.3) for a 3-D debug drawing with the ID of the desired drawing prefixed by "debugDrawing3d:" as its only parameter. If you wanted to see the example described above, you would type "dr debugDrawing3d:test3D" into the SimRobot console. Note that you might have to activate 3-D debug drawings in SimRobot first. You can do this by simply right-clicking on the scene view and selecting the desired type of rendering in the "Drawings Rendering" submenu.

3.6.5 Plots

The macro `PLOT(<id>, <number>)` allows plotting data over time. The plot view (cf. Sect. 8.4.7) will keep a history of predefined size of the values sent by the macro `PLOT` and plot them in different colors. Hence, the previous development of certain values can be observed as a time series. Each plot has an identifier that is used to separate the different plots from each other. A plot view can be created with the console commands *vp* and *vpd* (cf. Sect. 8.8.3).

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

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

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

3.6.6 Modify

The macro `MODIFY(<id>, <object>)` allows reading and modifying of data on the actual robot during runtime. Every streamable data type (cf. Sect. 3.4.3) can be manipulated and read, because its inner structure is gathered while it is streamed. This allows generic manipulation of runtime data using the console commands *get* and *set* (cf. Sect. 8.8.3). The first parameter of `MODIFY` specifies the identifier that is used to refer to the object from the PC, the second parameter is the object to be manipulated itself. When an object is modified using the console command *set*, it will be overridden each time the `MODIFY` macro is executed.

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

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

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

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

```
    MODIFY_ENUM("x", x);
  }
};

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

3.6.7 Stopwatches

Stopwatches allow the measurement of the execution time of parts of the code. The statements the runtime of which should be measured have to be placed into the macro `STOP_TIME_ON_REQUEST(<id>, <statements>)` (declared in *Src/Tools/Debugging/Stopwatch.h*) as second parameter. The first parameter is a string used to identify the time measurement. To activate a certain time measurement, e. g., `Foo`, a debug request `stopwatch:Foo` has to be sent. The measured time can be seen in the timing view (cf. Sect. 8.4.8). By default, a stopwatch is already defined for each representation that is currently provided. In the release configuration of the code, all stopwatches in process *Cognition* can be activated by sending the release option *stopwatches* (cf. command *ro* in Sect. 8.8.3).

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

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

Often rudimentary statistics (cf. Sect. 8.4.8) of the execution time of a certain part of the code are not sufficient, but the actual progress of the run time is needed. For this purpose there is another macro `STOP_TIME_ON_REQUEST_WITH_PLOT(<id>, <statements>)` that enables measuring *and* plotting the execution time of some code. This macro is used exactly as the one without `_WITH_PLOT`, but it additionally creates a plot `stopwatch:myCode` (assuming the example above) (cf. Sect. 3.6.5 for the usage of plots).

Chapter 4

Cognition

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

4.1 Perception

B-Human uses a scan line-based perception system. The *YUV422* images provided by the Nao camera have a resolution of 640×480 pixels. They are interpreted as *YUV444* images with a resolution of 320×240 pixels by ignoring every second row and the second Y channel of each *YUV422* pixel pair. The images are scanned on vertical scan lines (cf. Fig. 4.10). Thereby, the actual amount of scanned pixels is much smaller than the image size. The perception modules provide representations for the different features. The *BallPercept* contains information about the ball if it was seen in the current image. The *LinePercept* contains all field lines, intersections, and the center-circle seen in the current image, and the *GoalPercept* contains information about the goals seen in the image. All information provided by the perception modules is relative to the robot's position.

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

4.1.1 Definition of Coordinate Systems

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

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

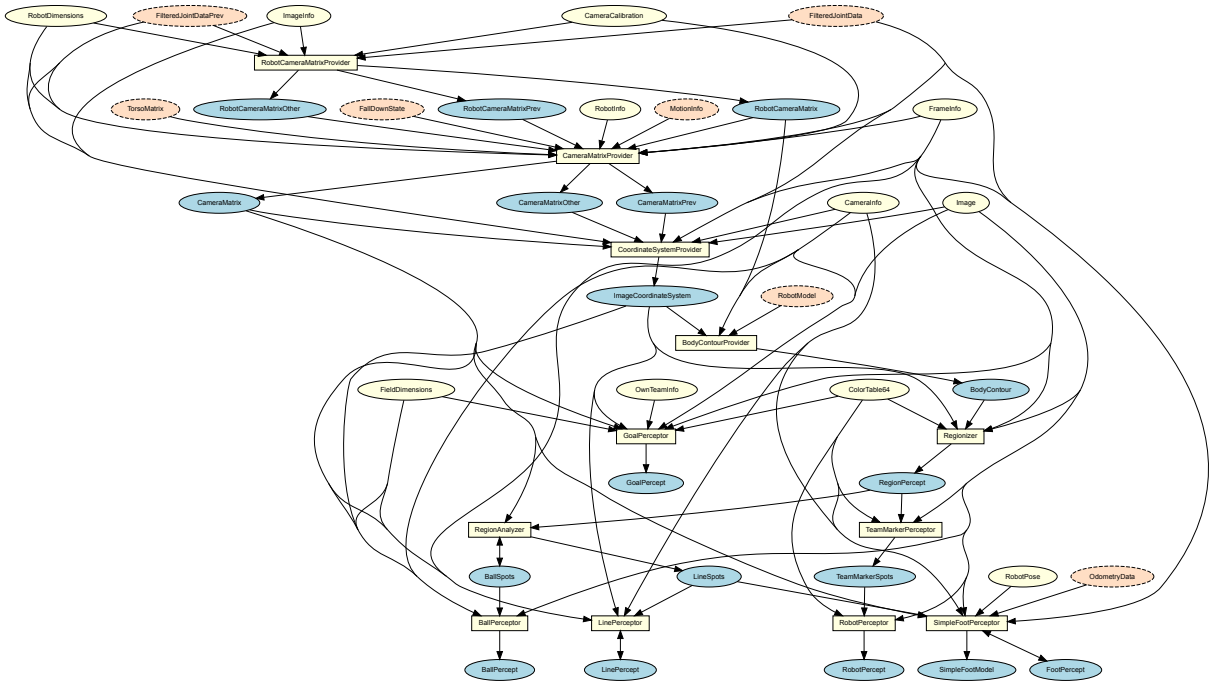


Figure 4.1: Perception module graph. Blue ellipses mark representations provided by cognition modules, which are marked as white squares. White ellipses mark required representations from the cognition process. In contrast, red ellipses with a dashed outline mark required representations of the motion process.

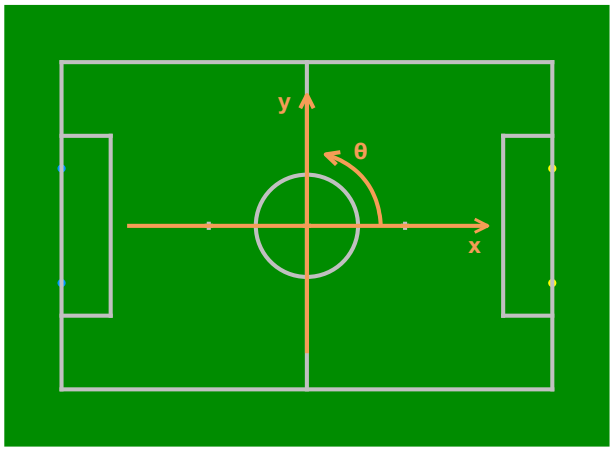


Figure 4.2: Visualization of the global coordinate system

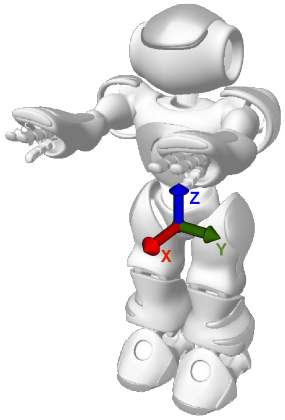


Figure 4.3: Visualization of the robot-relative coordinate system

4.1.1.1 Camera Matrix

The *CameraMatrix* is a representation containing the transformation matrix of the active camera of the Nao (cf. Sect. 4.1.2) that is provided by the CameraMatrixProvider. It is computed based on the *TorsoMatrix* that represents the orientation and position of a specific point within the robot's torso relative to the ground (cf. Sect. 5.1.6). Using the *RobotDimensions* and the current position of the joints the transformation of the camera matrix relative to the torso matrix is computed. This relative transformation matrix is also provided as *RobotCameraMatrix*, which is used to compute the *BodyContour* (cf. Sect. 4.1.3). In addition to these fixed parameters 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 *CameraMatrix* is used for projecting objects onto the field as well as the creation of the *ImageCoordinateSystem* (cf. Sect. 4.1.1.2).

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. Especially during competitions when robots are often repaired on-site and therefore require recalibration this is an annoying necessity.

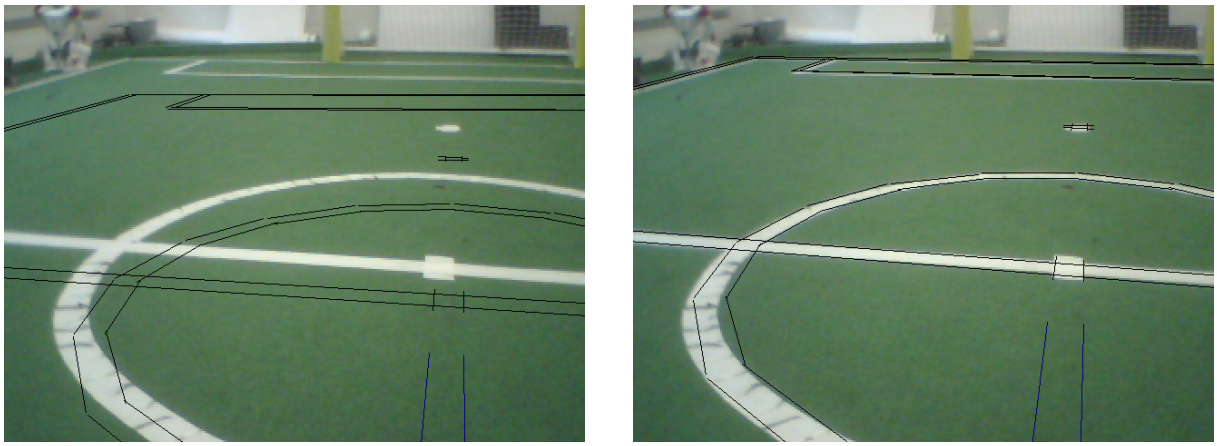


Figure 4.4: Projected lines before (left) and after (right) the calibration procedure

In order to overcome this problem, we developed a semi-automatic calibration module. It is based on the calibration module that has been introduced last year. This year, we developed an improved version of the camera calibration module to further simplify the calibration process. It differs in the following points:

- Instead of marking defined points on the field, the user can mark arbitrary points on field lines. This is especially useful for the operation during competitions because it is now possible to calibrate the camera if parts of the field lines are covered (e.g. by robots or other team members).
- Since we use both cameras this year, the calibration module is able to calibrate the parameters of the lower as well as the upper camera. Therefore, the user simply has to mark additional reference points in the image of the upper camera.
- In order to optimize the parameters, the Gauss-Newton algorithm is used[1] instead of hill climbing. Since this algorithm is especially designed for non-linear least squares problems like this, the time to converge is drastically reduced to typically 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.

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

- The error function takes the distance of a point to the next line in image coordinates instead of field coordinates into account. This is a more accurate error approximation because the parameters and the error are in angular space.

With these improvements the module typically produces a parameter set that requires only little manual adjustments, if any. The calibration procedure using the above mentioned `CameraCalibrator` module is as follows:

1. Connect the simulator to a robot on the field and place it on a defined spot.

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

3. Set the robot's *RobotPose* according to the robot's position on the field (cf. Sect. 4.1.1).

4. Start collecting points using *dr module:cameraCalibrator:collectPoints*. Move the head to collect points for different rotations of the head. The number of collected points is shown in the top left corner of the image view.

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

The calibration module allows to arbitrarily switch between upper and lower camera during the point collection phase. Both cameras should be considered for a good result. For the purpose of refinement calibration of the mentioned robot-specific parameters, there is a debug drawing that projects the field lines into the camera image. To activate this drawing, type *vid raw module:CameraMatrixProvider:calibrationHelper* in the simulator console. This drawing is helpful for calibrating, because the real lines and the projected lines only match if the camera matrix and hence the camera calibration is correct (assuming that the real position corresponds to the self-localization of the robot). Modify the parameters of *CameraCalibration* so that the projected lines match the field lines in the image (see Fig. 4.4 for a desirable calibration).

4.1.1.2 Image Coordinate System

Based on the camera transformation matrix, another coordinate system is provided which applies to the camera image. The *ImageCoordinateSystem* is provided by the module CoordinateSystemProvider. Its origin of the y-coordinate lies on the horizon within the image (even if it is not visible in the image). The x-axis points right along the horizon whereas the y-axis points downwards orthogonal to the horizon (cf. Fig. 4.5). For more information see also [35]. To get a good estimate of the current motion of the camera, the current *TorsoMatrix* provided by the process *Motion* to the process *Cognition* not only contains the current pose of the torso, but also the offset to the previous *TorsoMatrix* from the last *Motion* frame 10ms ago. Therefrom, the *CameraMatrixPrev* is computed that enables the CoordinateSystemProvider to determine the rotation speed of the camera and thereby interpolate its orientation when recording each image row. As a result, the representation *ImageCoordinateSystem* provides a mechanism to compensate for different recording times of images and joint angles as well as for image distortion caused by the rolling shutter. For a detailed description of this method applied to the Sony AIBO see [28].

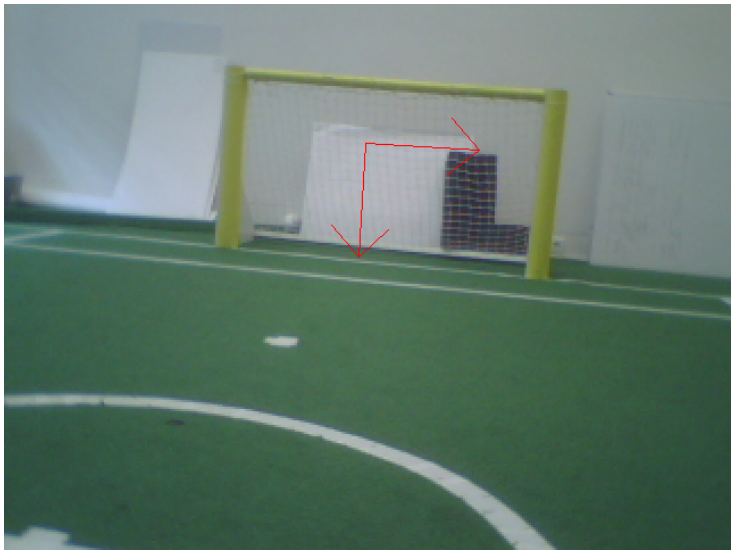


Figure 4.5: Origin of the *ImageCoordinateSystem*

4.1.2 Using Both Cameras

As already mentioned in [33], the Nao robot is equipped with two video cameras, both mounted in the head of the robot. The first camera is mounted in the middle of the robots forehead, the second camera is installed about $4cm$ below the first one and tilted by 40° with respect to the upper camera. Since the vertical opening angle of the cameras is only 34.8°, stereo vision is impossible, and to make things worse, the cameras of the Nao cannot deliver images simultaneously. This is why we previously used only the lower camera and disabled the upper one completely.

Using only the lower camera is linked with some disadvantages. Switching quickly between close and far targets requires enormous head acceleration. An even bigger problem is caused by the shoulder pads in conjunction with the lower camera. These are blocking huge parts of the camera's field of view when the robot looks alongside itself.

To avoid such situations, we implemented the ability to switch between the cameras during gameplay. First of all, we added the representation *CameraMatrixOther* which is the camera matrix for the currently switched-off camera. In addition, the module CameraProvider, which is responsible for getting images from the enabled camera and switching cameras, provides the representation *ImageInfo* telling other modules which camera is enabled and which camera has been enabled in the previous frame. To switch cameras, the CameraProvider takes the representation *ImageRequest*, which contains information about which camera should provide the next image.

The *ImageRequest* is provided by the CameraControlEngine, which is the module that has been added to use two cameras. This module takes the *HeadMotionRequest*, provided by the BH2011HeadControlEngine. The *HeadMotionRequest* contains the target at which the robot is supposed to look. This target is either specified as head joint angles for the lower camera, a point relative to the ground or a point relative to the robot's torso. The CameraControlEngine then determines the head tilt joint angles both for the case that the current camera will be used to look at the target and for the case that the other camera will be used to look at the target.

With the head tilt joint angle determined the camera is selected. In addition to the target, the *HeadJointRequest* can force the use of a certain camera or let the CameraControlEngine determine

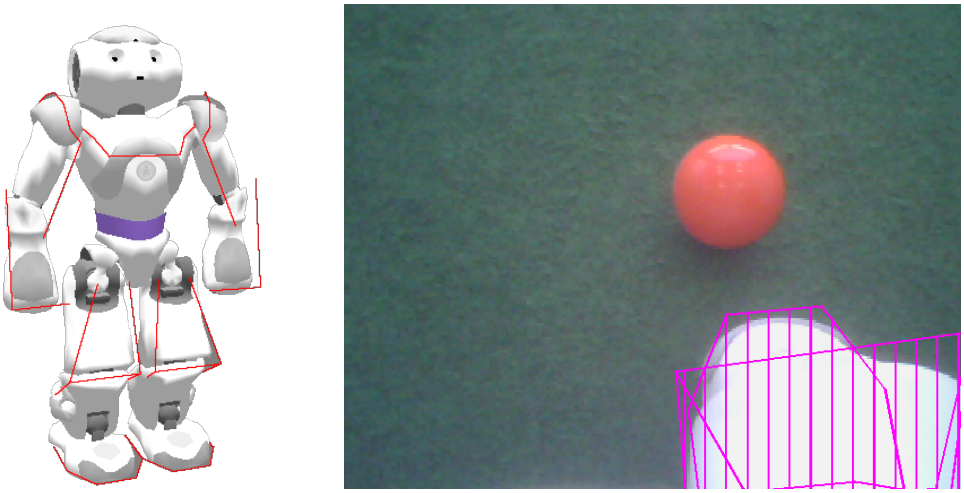


Figure 4.6: Body contour in 3-D (left) and projected to the camera image (right).

which camera to use. In the latter case, the upper camera is used at the default, but the tilt angle is checked against a table of useful tilt angles. That table contains for each camera and for each pan angle the minimum useful tilt angle, such that the tilt angle in question is physically reachable and that the camera in question wouldn't be obstructed by the robots own body – for instance the shoulder pads – if the tilt angle was reached.

The camera selection is then written into the *ImageRequest*, and the corresponding pan and tilt angles are written into the *HeadAngleRequest* representation which is to be used by the motion module HeadMotionEngine (cf. Sect. 5.2.4) to move the head accordingly.

4.1.3 Body Contour

If the robot sees parts of its body, it might confuse white areas with field lines. However, by using forward kinematics, the robot can actually know where its body is visible in the camera image and exclude these areas from image processing. This is achieved by modeling the boundaries of body parts that are potentially visible in 3-D (cf. Fig. 4.6 left) and projecting them back to the camera image (cf. Fig. 4.6 right). The part of that projection that intersects with the camera image or is above it is provided in the representation *BodyContour*. It is used by the image processor as lower clipping boundary. The projection relies on *ImageCoordinateSystem*, i. e., the linear interpolation of the joint angles to match the time when the image was taken.

4.1.4 Color Table

A fast identification of color classes is achieved by precomputing the class of each color and store it in a *ColorTable64*. To save space the dimensions of this color table are reduced from 256x256x256 down to 64x64x64. This is still precise enough, but saves resources.

The class `ColorTableHandler` provides methods to modify color table data. Most of them are accessible by console commands (see command *ct* in Sect. 8.8.3). Currently there are two ways of creating such a color table. First it can be created completely by hand, training every color manually. To speed up this process, colors in an appropriate range to the selected color are assigned the same color class. This process has some weaknesses. On the one hand it needs a lot of training colors to cover the whole color class. On the other hand color classes tend to blur into another, which makes it difficult to decide what color class a color belongs to.

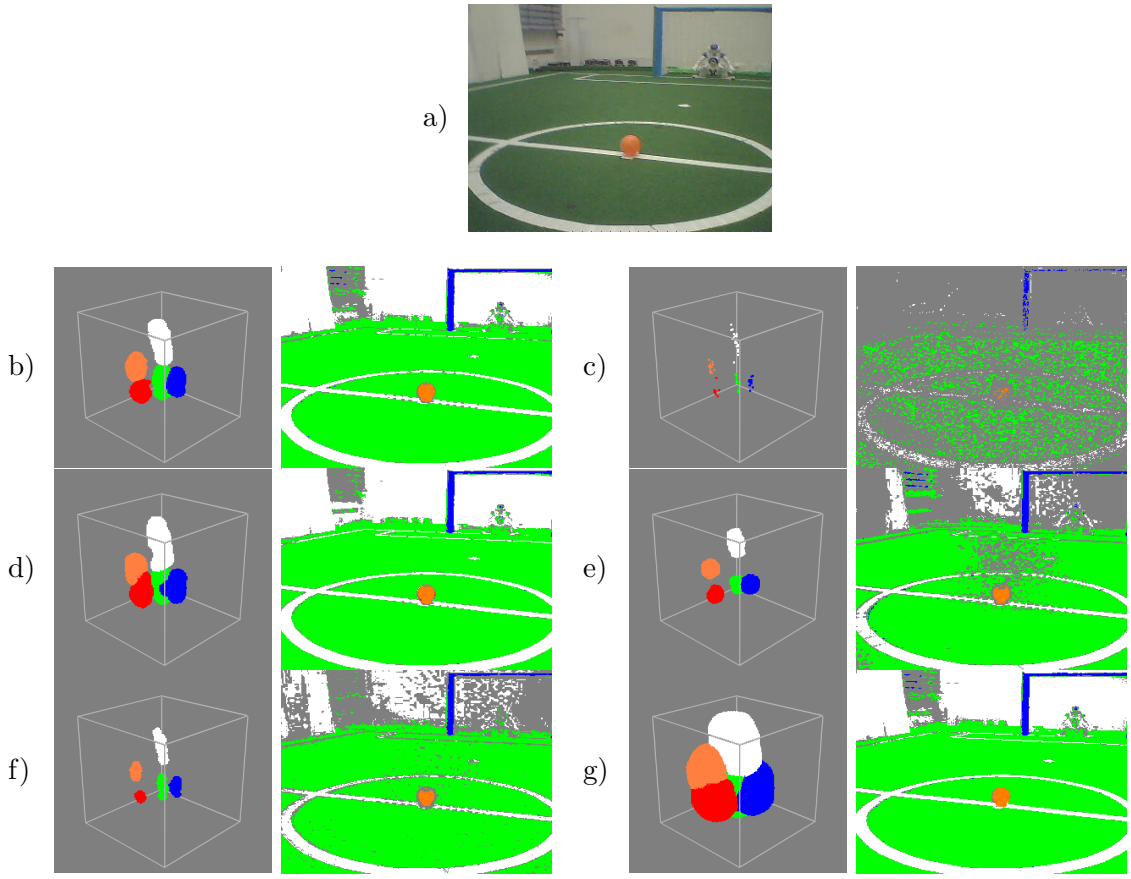


Figure 4.7: This figure shows different configurations of the 3D-tree's *neighbor* and *range* parameters, with the color table on the left and the corresponding segmented image on the right. a) is the raw image, the robot is looking at. b) shows the standard configuration with $neighbor = 3$ and $range = 20$, whereas c) shows the training data itself. The third row shows two different configurations of the *neighbor* parameter with a fix *range* value of 20. d) uses $neighbor = 1$ whereby small deviations affect the result more than in b). e) uses $neighbor = 13$ which results in small clusters, because only the core of a cluster in the training set has enough neighbors. f) and g) show different variations of the *range* parameter with fix $neighbor = 3$. f) uses a *range* of 10, so only colors nearby the training data are classified. In contrast g) uses $range = 50$ which results in huge blobs.

To speed up the process, a nearest neighbor approach based on a 3D-tree was added [1]. It searches the nearest neighbors of a color in the training data to decide what color class the color belongs to. Instead of adding classifications directly to the color table, this approach gathers the training data in a vector, so it can be processed more easily. After completing the training, the training data is used to build the 3D-tree. This tree allows fast computation of nearest neighbors in the training data for a given color. This is possible due to the structure of the 3D-tree. Every node in the tree corresponds to a classification in the training data. To insert a node the training data is sorted according to the values of one dimension. The median is selected and inserted into the tree. The dimension that is used depends on the level at which the node is to be inserted. So the training data can be split at the median node's value, wherefore the next insertion only considers half the data of the previous step. When searching the nearest neighbors this can be used to reduce the search space as the search descends in the tree (cf. Fig. 4.8).

To handle the training data the `ColorTableHandler` is extended by the `ColorTableCreator`. It provides methods to modify the training data and generate the 3D-tree. The actual 3D-tree is implemented in the `Color3DTree` class. This builds the tree and computes the nearest

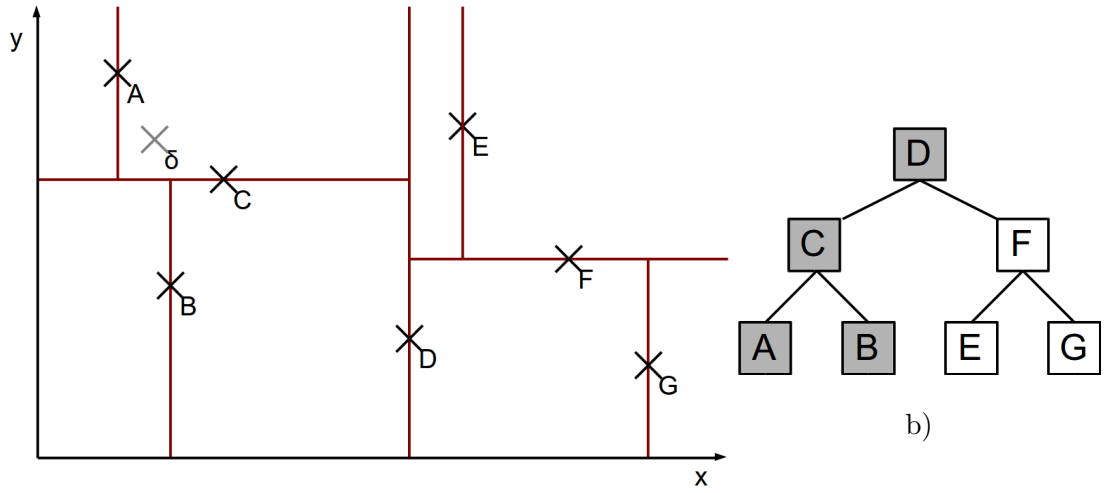


Figure 4.8: This figure shows a two dimensional example of a kD-tree. a) shows a coordinate system with dimensions x and y and the points $A \dots G$ and δ. $A \dots G$ are used as training data to build up the 2D-tree b), whereas δ is a point witch neighbors are searched. The red lines mark how the space is split when inserting $A \dots G$ in the tree. In this case only the grey nodes of b) have to be visit to ensure that the three nearest neighbors of δ have been found.

neighbors of a given color. Although the 3D-tree is quite fast the runtime and the result strongly depend on two parameters. The number of neighbors considered in the color classification and the maximum distance of training data, that is considered as neighbor (cf. commands *ct kd neighbor* and *ct kd range*). The number of neighbors affects the influence of small clusters in the training data to the color classification. The maximum neighborhood distance controls the size of the classified cluster in the final color table. This can be used to adjust the cluster size according to lighting to prevent blur (cf. Fig. 4.7). In both cases increasing the value increases the time needed to classify all colors in the color table. Even if the search is quite fast it is to slow to be used in real time on the Nao. So the final classifications are inserted in the color table. This way the Nao can use it in the same way as a color table generated without the nearest neighbor approach.

Since both approaches produce a color table it is possible to modify a nearest neighbor based color table by hand later on. When generating a new color table using the nearest neighbor approach the color table will be overwritten and previous data, that is not included in the training data of the 3D-tree, will be lost. Therefore it is possible but not recommended to cross the classic and the tree based approach.

Color tables as well as 3D-tree training data can be stored to files (cf. commands *ct save/load* and *ct kd save/load*). Tree training data is stored in **.kdt*-files using the configuration map format (cf. Sect. 3.4.5). Color tables are saved to **.c64*-files, if training data is available it will be stored automatically, too. The same also applies to loading a color table, if a **.kdt*-file is available the training data will be loaded, too.

4.1.5 Image Processing

The image processing is split into the three steps segmentation and region-building (cf. Sect. 4.1.5.1), region classification (cf. Sect. 4.1.5.2) and feature extraction (cf. Sect. 4.1.5.3, 4.1.5.4, 4.1.5.5). Since the goals are the only features that are above the horizon and the field

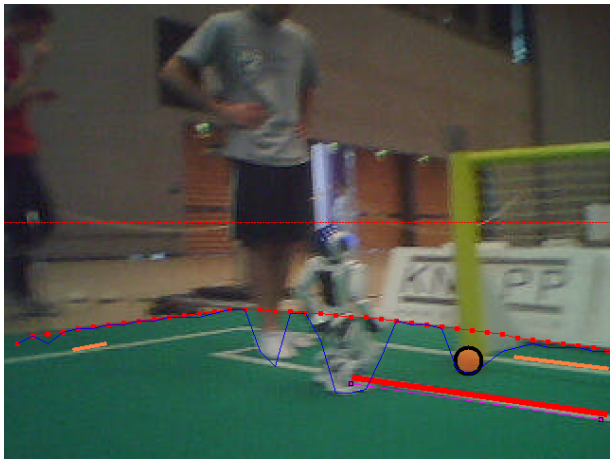


Figure 4.9: Field borders: the small red-dotted line is the horizon, the blue line connects the green points found, the red line is the upper part of the convex hull with a dot for each scan line, the ball in front of the post would be missed without the convex hull.

border, the goal detection is based on special segments and is not integrated in the region classification.

4.1.5.1 Segmentation and Region-Building

The Regionizer works on the camera image and creates regions based on segments found on scan lines. It provides the *RegionPercept* containing the segments, the regions, the field borders and the horizontal and vertical goal segments.

First the field borders are detected. This is done by running scan lines, starting from the horizon, downwards until a green segment of a minimum length is found. From these points the upper half of the convex hull is used as field border. Using the convex hull ensures that we cannot miss a ball that is in front of a goal post or a robot and therefore has no green above (cf. Fig. 4.9). Since all features that are detected based on regions are on the field (ball and field lines) further segmentation starts from these field border points. Because we do not want to accidentally detect features inside the body of the robot, the *BodyContour* is used to determine the end of each scan line. Now scan lines running from the field border to the bottom of the image, clipped by the *BodyContour*, are used to create segments. To be tolerant against noise in the segmentation, a certain number of different colored pixels in a row may be skipped without starting a new segment. Since we want to create regions based on the segments and our scan lines are not very close to each other, non-green segments are explored. This means we are trying to find equally colored runs between the current scan line and the last one which touches the segment and starts before or ends after the current segment. This is necessary since a field line that is far away might create segments that do not touch each other because of the distance of the scan lines. However, we still want to be able to unite them to a region (cf. Fig. 4.10). Next to the regular scanlines there are used some scanlines between those in case of the ball was not seen for a certain time. These scanlines only recognize orange regions. In this way we also can find balls even if the distance from the robot is that large that balls would disappear between usual scanlines.

Based on these segments regions are created using the algorithm shown in Algorithm 1 and 2. It iterates over all segments (sorted by y- and x-coordinates) and connects the current segment to the regions already created or creates a new one in each iteration. To connect each segment to

the regions already found, a function is called that takes as input the segment and a pointer to a segment in the previous column that is the first segment that might touch the segment to be connected. The function returns a pointer to the segment in the last column that might touch the next segment. This pointer is passed again to the function in the next iteration for the next segment. The algorithm is capable of handling overlapping segments, which is needed because the region-building is done on the explored segments. While building the regions, information about neighboring regions is collected and stored within the regions.

Algorithm 1 Region-building

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

We do not create green regions, since green is treated as background and is only needed to determine the amount of green next to white regions, which can be determined based on the segments.

Two segments of the same color touching each other need to fulfill certain criteria to be united to a region:

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

These criteria (and all other thresholds mentioned before) are configurable through the file *regionizer.cfg*. For some colors these features are turned off in the configuration file. These restrictions are especially needed for white regions, since we do not want to have a single big region containing all field lines and robots. The result of these restrictions is that we most likely get small straight white regions (cf. Fig. 4.10). For example, at a corner the change in direction of the white segments should exceed the maximum change in direction, and therefore they are not united or if a robot stands on a field line, the length ratio of the two segments is most likely too big and the regions are not united.

Algorithm 2 ConnectToRegions(s, lastColumnPointer)

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

4.1.5.2 Region Classification

The region classification is done by the RegionAnalyzer. It classifies the regions within the *RegionPercept* whether they could be parts of a line or the ball and discards all others. It provides the *LineSpots* and *BallSpots*.

The RegionAnalyzer iterates over all regions in the *RegionPercept* and checks for all white regions whether they could be part of a line and for all orange regions whether they could be a ball.

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

- The region must consists of a certain number of segments, and it must have a certain size. If a region does not have enough children but instead has a size that is bigger than a second threshold it is also accepted as a line. This case occurs for example when a vertical

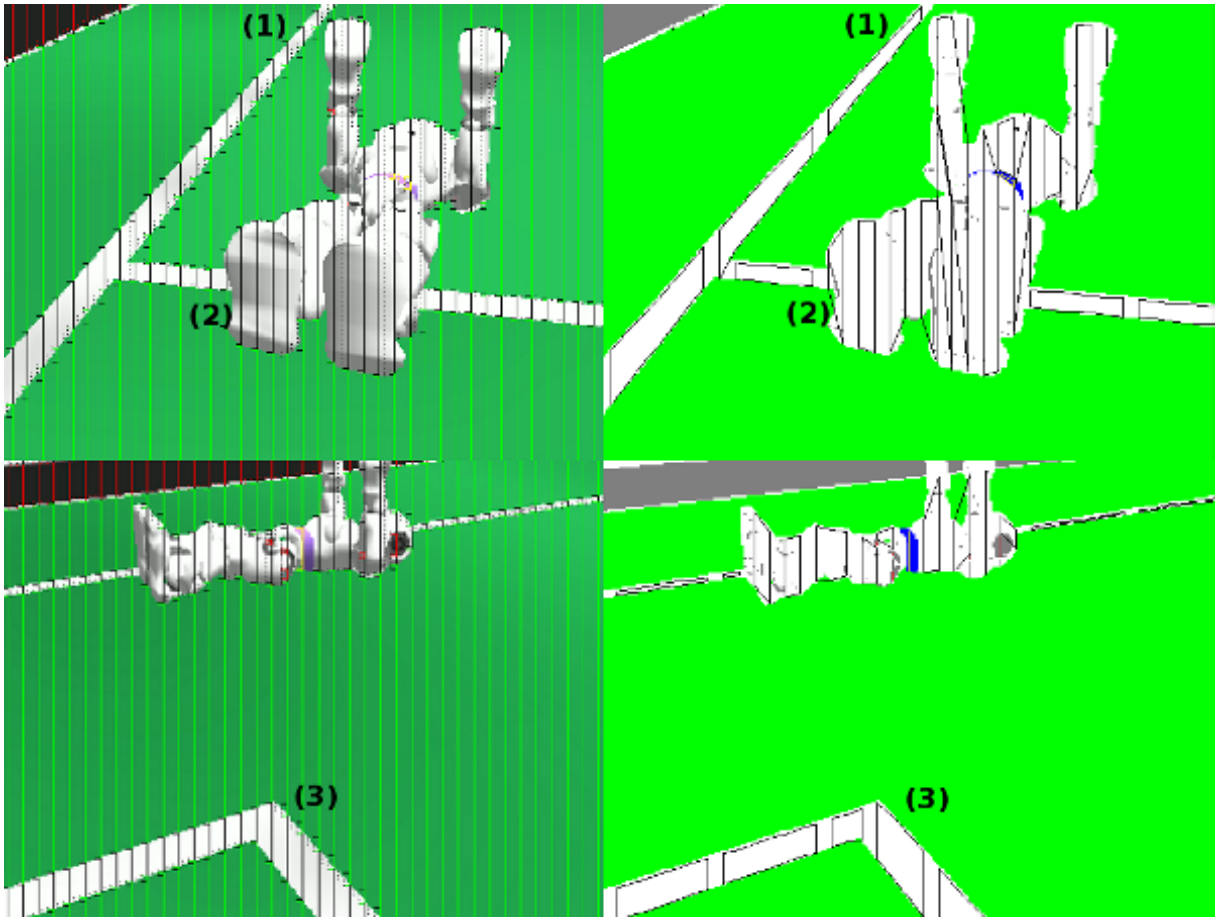


Figure 4.10: Segmentation and region-building: (1) segment is not touching the other, but the explored runs (dotted lines) are touching the other segment, (2) not connected because of length ratio, (3) not connected because of change in direction.

line only creates a single very long segment.

- The axis of orientation must be determinable (since this is the base information passed to further modules).
- The size of neighboring uncolored regions must not exceed a certain size and the ratio of the size of the white region and the neighboring uncolored regions must not exceed a certain ratio (this is because robot parts are most likely classified as uncolored regions).
- A horizontally oriented white region must have a certain amount of green above and below, while a vertically oriented white region must have a certain amount of green on its left and right side.

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

- Does the region have a certain size?

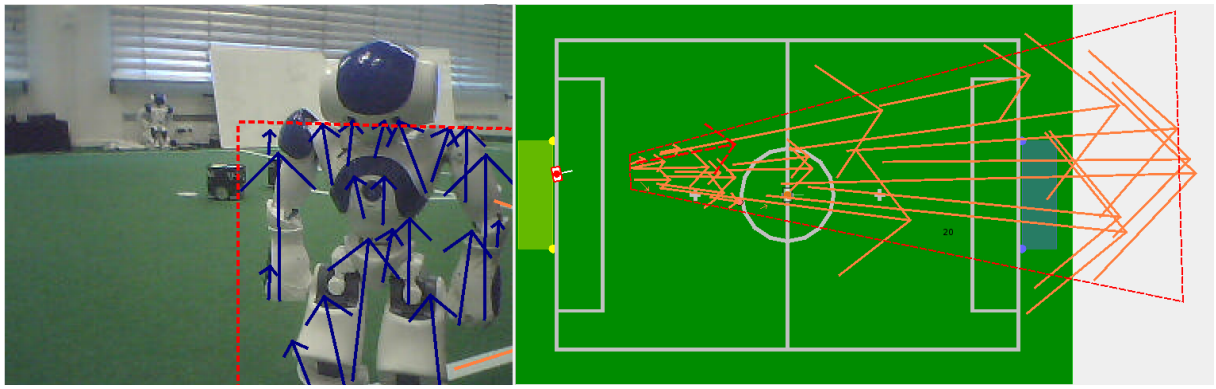


Figure 4.11: Ban sector built from *NonLineSpots*. The red arrow is a false line spot detected in the robot and discarded because of the ban sector.

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

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

Orange regions need to have a certain eccentricity to be accepted as BallSpots. For orange regions that are accepted as possible balls the center of mass and the eccentricity is stored as *BallSpot* in the *BallSpots*.

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

4.1.5.3 Detecting Lines

The LinePerceptor works on the *LineSpots* provided by the RegionAnalyzer. It clusters the *LineSpots* to lines and tries to find a circle within the line spots not clustered to lines. Afterwards the intersections of the lines and the circle are computed. The results are stored in the *LinePercept*.

To avoid detecting false lines in robots first of all the *NonLineSpots* within the *LineSpots* are clustered to so called *ban sectors*. The *NonLineSpots* are transformed to field coordinates using the 3-D camera equation. Since the *NonLineSpots* are most likely parts of robots that are above the field, this creates a very characteristic scheme in field coordinates (cf. Fig. 4.11).

After creating the ban sectors, the LinePerceptor creates line segments from the line spots. For each line spot that has a certain width/height ratio the start and end point of the spot is transformed to field coordinates using the 3-D camera equation. If one of the two points is farther away than a certain threshold the segment is discarded, since we can assume no line can be farther away than the diagonal of the field size. The threshold is chosen a little bit smaller than the diagonal of the field size because a line which has almost this distance is too small to be recognized by our vision system. If the spot is similar to the regions that are created by robots (certain width/height ratio, vertical, and a certain length), the LinePerceptor checks whether that spot is inside a ban sector. If that is the case the segment is discarded. For all segments created the Hesse normal form is calculated.

From these segments the lines are built. This is done by clustering the line segments. The clustering is done by Algorithm 3. The basic idea of the algorithm is similar to the quality threshold clustering algorithm introduced by Heyer et al. in [8], but it ensures that it runs in the worst-case-scenario in $\mathcal{O}(n^2)$ runtime. Therefore it is not guaranteed to find optimal clusters. Since the number of line spots is limited by the field setup, practical usage showed that the algorithm has an acceptable runtime and delivers satisfiable results. The difference of the directions and distances of the Hesse normal form of two segments need to be less than a certain threshold to accept the two segments as parts of the same line. Each cluster of segments also needs a segment with a length bigger than a certain threshold. This is necessary to avoid creating lines from small pieces, for example a cross and a part of the circle. The lines are also represented as Hesse normal form.

Algorithm 3 Clustering LineSegments

```
while lineSegments ≠ ∅ do
  s ← lineSegments.pop()
  supporters ← ∅
  maxSegmentLength ← 0
  for all s' ∈ lineSegments do
    if similarity(s, s') < similarityThreshold then
      supporters.add(s')
      if length(s') > maxSegmentLength then
        maxSegmentLength = length(s')
      end if
    end if
  end for
  if supporters.size() > supporterThreshold and maxSegmentLength >
  segmentLengthThreshold then
    createLine({s} ∪ supporters)
    lineSegments ← lineSegments\supporters
  end if
end while
```

All remaining line segments are taken into account for the circle detection. For each pair of segments with a distance smaller than a threshold the intersection of the perpendicular from the middle of the segments is calculated. If the distance of this intersection is close to the real circle radius, for each segment a spot is generated which has the distance of the radius to the segment. After the spots are created the same clustering algorithm used for the lines is used to find a cluster for the circle. As soon as a cluster is found which fulfills the criteria to be a circle it is assumed to be the circle (cf. Fig. 4.12). For all remaining line segments which have a certain length additional lines are created.

Since it might happen that a circle is detected but single segments on the circle were not recognized as part of it, all lines which are aligned on the circle are deleted. It might also happen that a single line in the image created multiple lines (because the line was not clustered but the single segments were long enough to create lines on their own) therefore lines which are similar (with respect to the Hesse normal form) are merged together. Since it might happen that single segments where not clustered to a line (for example because the line spot is not perfectly aligned with the line, what can be caused by an inaccurate segmentation) the remaining single segments are merge to the lines, if the distance of their start and end point is close to a line. For each resulting line the summed length of the segments must cover a certain percentage of the length of the line. Otherwise the line will be discarded. This avoids creating lines for segments

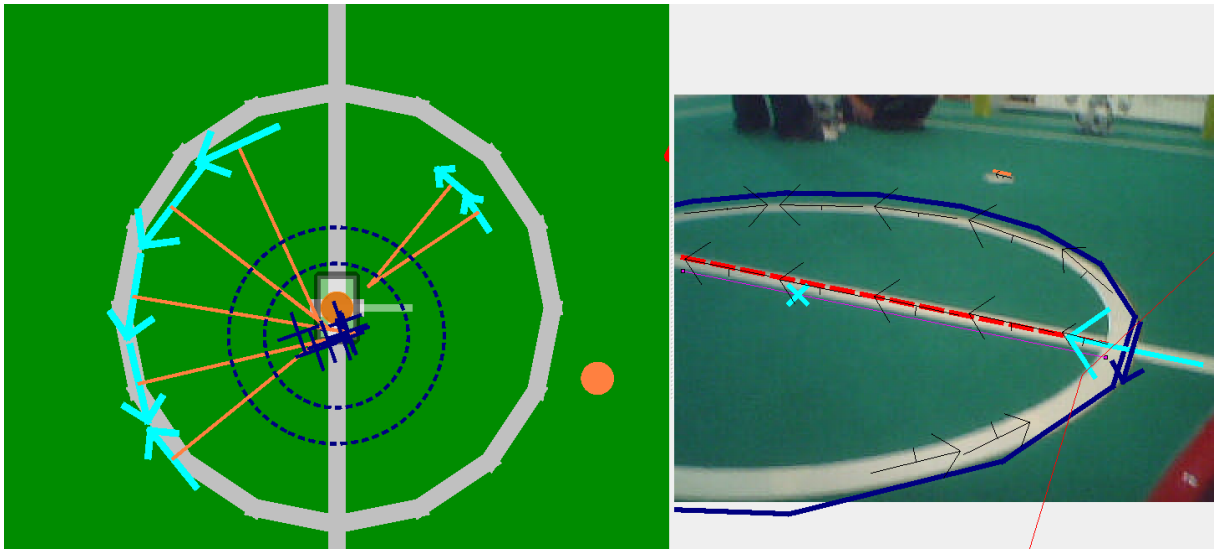
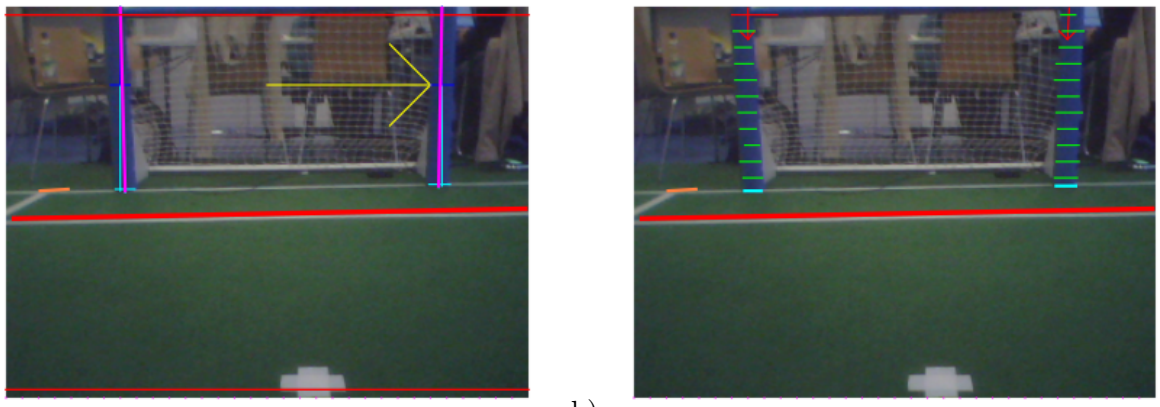


Figure 4.12: Circle detection: blue crosses: circleSpots, blue circles: threshold area, orange lines: perpendiculars of segments



a) b)

Figure 4.13: The debug drawings of the GoalPerceptor. Image a) shows the fist scans (blue and light blue) that find the foot and head of a potential post. The arrow marks the horizon. Image b) shows the runs orthogonal to the vector for determining the width of the potential post.

which are far away from each other, for example the cross and a part of the circle.

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

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

4.1.5.4 Detecting the Goal

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

Since the foot of a post must be below the horizon and the head of a post must be above (or not visible) the post must cross the projection of the horizon to the image. Therefore it is sufficient

to scan the projection of the horizon in the image for blue or yellow segments to detect points of interest for further processing. If the horizon is above the upper image border, a line a few pixels below the upper image border will be scanned.

From these points of interest, the GoalPerceptor tries to find the foot and head of a potential post. This is done by running up and down from a point of interest and continuing the run a few pixels left or right if a border was found (Fig. 4.13).

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

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

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

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

4.1.5.5 Detecting the Ball

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

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

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

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

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

4. Calculating center and radius of the ball.

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

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

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

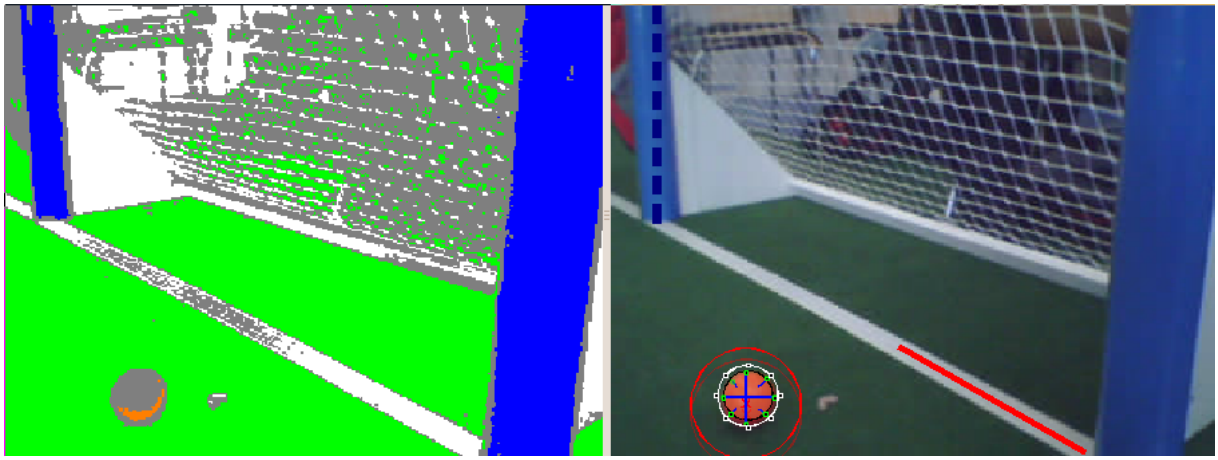


Figure 4.14: The debug drawing of the BallPerceptor. The green dots mark the end of the scan lines for ball points. The white circle should be around the ball, i. e. the color differs too much from the ball spot.

4.1.5.6 Detecting Other Robots

Our robot recognition modules TeamMarkerPerceptor and RobotPerceptor are described in detail in [2]. However, we modified some aspects during the last months. We will give a brief overview and highlight the changes.

The TeamMarkerPerceptor requires the representation *RegionPercept* and extracts regions that are classified as blue or robotBlue (team markers of the blue team) or red (team markers of the red team). We grow these regions based on distances in the YCbCr color space and calculate the orientation of the principal axis of inertia. We perform some sanity checks on the area, orientation and growth. We do not check the shape of the team marker any longer.

The RobotPerceptor processes the output of the TeamMarkerPerceptor. It scans the environment of the team marker. We added another sanity check here: we check the variances of the scan lines. The variances of the robot body's Cb and Cr components should be low while the variance of the Y component is high. All these parameters are manually tuned and should be adjusted when the lightning conditions change. You can do this by modifying the configuration files *Config/robotPerceptor.cfg* and *Config/teamMarkerPerceptor.cfg*. In order to locate the robot on the field we need a pixel below its feet. We search for a large green area.

The *RobotPercept* is the input of the RobotLocatorUKF.

4.1.5.7 Detecting Robot Feet

Since the sonar measurements have a quite poor angular precision and the robot detection depends on the team markers, recognizing nearby opponent robots is inaccurate. Therefore, we put some effort in recognizing robot feet. The approach we introduce here works generally on *NonLineSpots* provided by the RegionAnalyzer. As you can see in Fig. 4.11, *NonLineSpots* appear in body parts of close robots in the range of view. The perception process is initialized only if there are enough *NonLineSpots* in the image and if the head pitch angle is between $0°$ and $-35°$. This angle restriction exists, because we only want to find feet that are in the vicinity and, of course, to spare computational resources.

A preselection of the *NonLineSpots* ignores all spots that are too short and also have green or orange pixel above the top point. If there are enough spots left, the following step is scanning for white runs only in the upper horizontal row of the image. If at least one sufficiently long

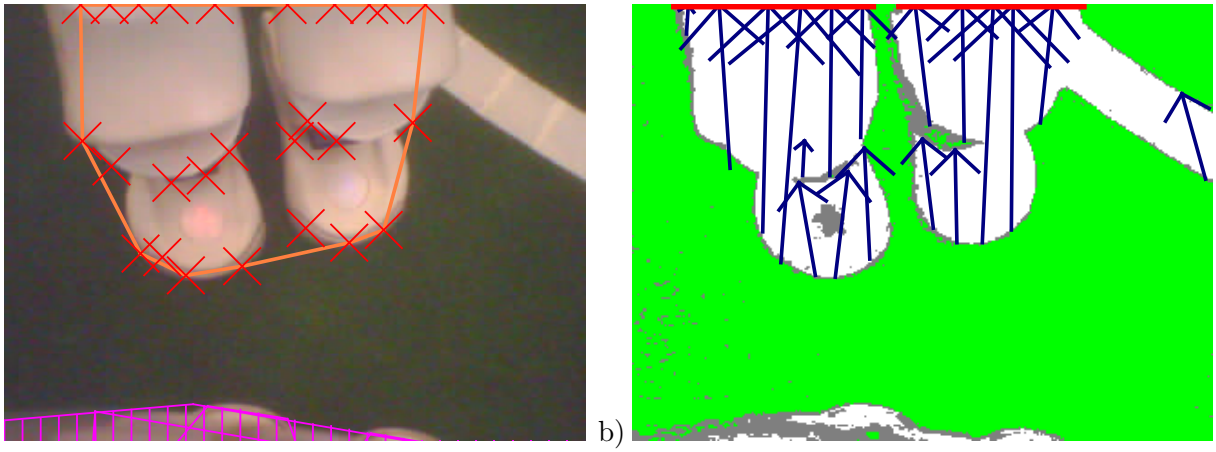


Figure 4.15: The SimpleFootPerceptor in action. Image a) shows the convex hull of the processed *NonLineSpots*. Image b) shows the unprocessed *NonLineSpots* (blue arrows). The two red lines on the top are found runs. The rightmost spot will be dismissed, as you can see on image a).

run was found, the *NonLineSpots* are associated with the run if the x component of the top and bottom points of a spot are within the range of the calculated runs. Subsequently, runs and their associated spots are dismissed if the count of associations (per run) is insufficient. At the end, the closest, leftmost, and rightmost points of the remaining spots are determined and – based on their relative distance to the robot – projected onto the field.

These information are passed to the *FootPercept*, which is used afterwards to determine the *FootModel*, which is very simple. After observing a *FootPercept* in four consecutive frames, the following perceptions are used as the model itself. If further perceptions stay away, the last valid perception remains for few seconds as the model. If that time has elapsed without new perceptions, the model becomes invalid.

4.2 Modeling

To compute an estimate of the world state – including the robot's position, the ball's position and velocity, and the presence of obstacles – given the noisy and incomplete data provided by the perception layer, a set of modeling modules is necessary. The modules and their dependencies are depicted in Fig. 4.16.

4.2.1 Self-Localization

For self-localization, B-Human uses a particle filter based on the Monte Carlo method [3] as it is a proven approach to provide accurate results in such an environment [34]. Additionally, it is able to deal with the kidnapped robot problem that often occurs in RoboCup scenarios. For a faster reestablishment of a reasonable position estimate after a kidnapping, the *Augmented MCL* approach by [7] has been implemented. A comprehensive description of our general state estimation implementation – applied to a Humanoid League scenario – is given in [15]. An overview of subsequently added features and the necessary perceptive components was published in [13].

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

B-Human's current vision system (cf. Sect. 4.1.5) provides a variety of perceptions that have all been integrated into the sensor model: goal posts (ambiguous as well as unambiguous ones),

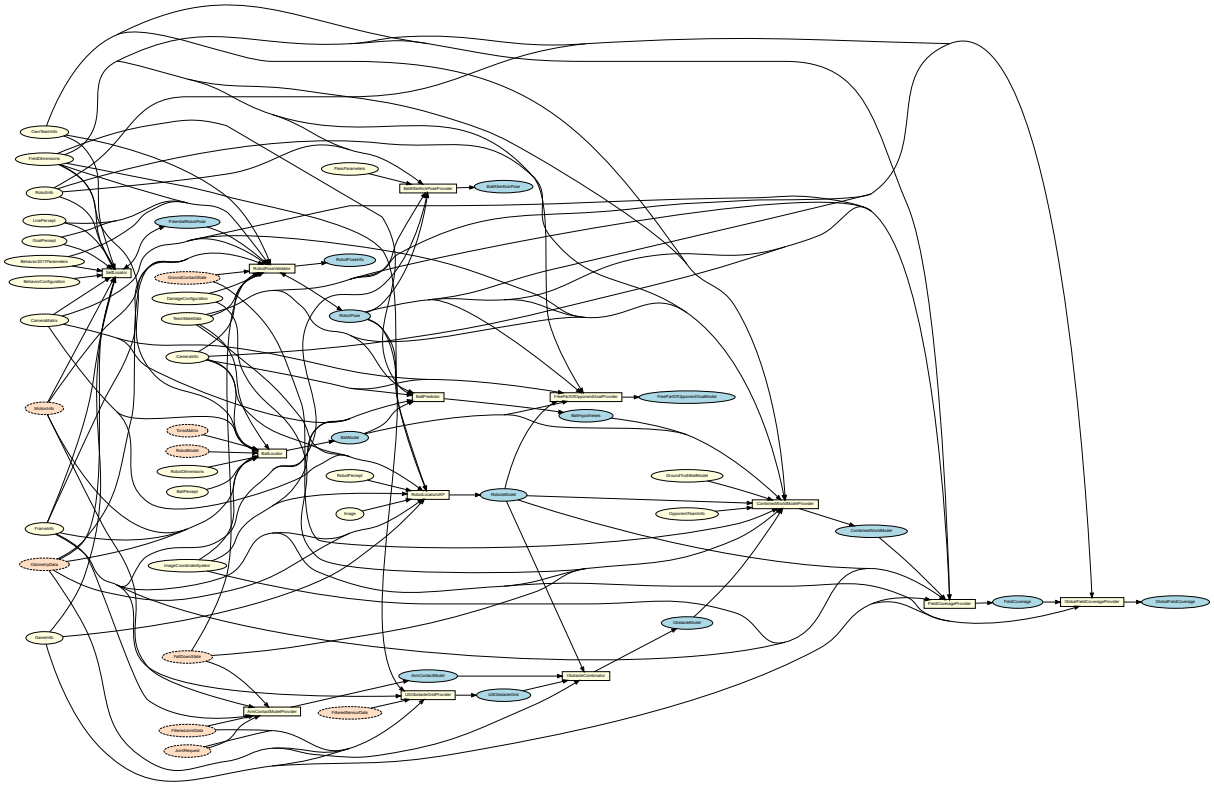


Figure 4.16: Modeling module graph. Blue ellipses mark representations provided by cognition modules, which are marked as white squares. White ellipses mark required representations from the cognition process. In contrast, red ellipses with a dashed outline mark required representations of the motion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

line segments (of which only the endpoints are matched to the field model), line crossings (of three different types: L, T, and X), and the center circle. During each sensor update of the particle filter, always a fixed number (currently six) of randomly selected perceptions is used, independently of the total number of perceptions; only goal post percepts are always preferred over other ones.

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

Through the process of sensor resetting [21] (cf. Fig. 4.17b), which is used to overcome kidnapping problems, the probability distribution of the sample set often becomes multimodal (cf. Fig. 4.17a). To robustly and computationally efficiently track different sample clusters without any discretization drawbacks as, for instance, in the popular binning approach, the approach presented in [17] is applied.

4.2.2 Robot Pose Validation

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

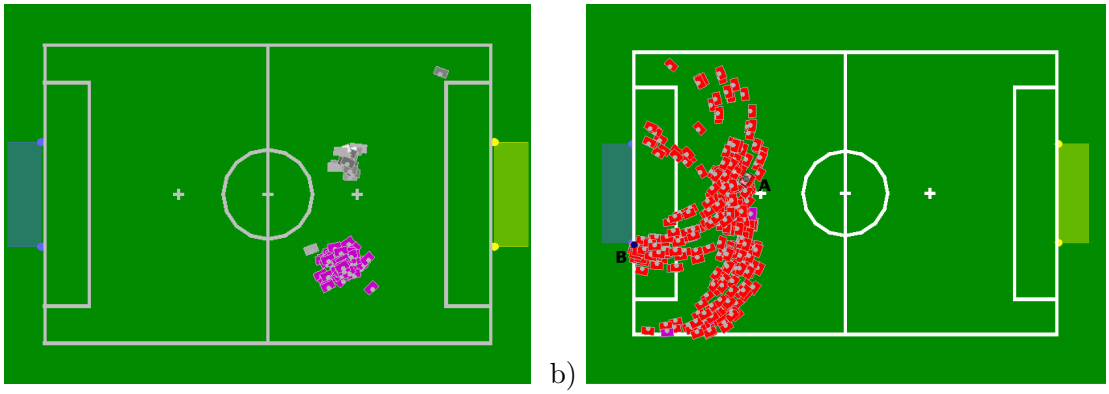


Figure 4.17: a) Multimodal probability distribution: The samples have clustered at two different positions. The cluster selected for pose computation is drawn in magenta. b) Sensor resetting: A robot standing at position A observes the non-unique goalpost at position B. The red boxes denote a sample set that has been fully generated using this single observation. All samples are situated on circles around the two possible goal posts. The different distances to the posts are a result of the sensor model's uncertainty.

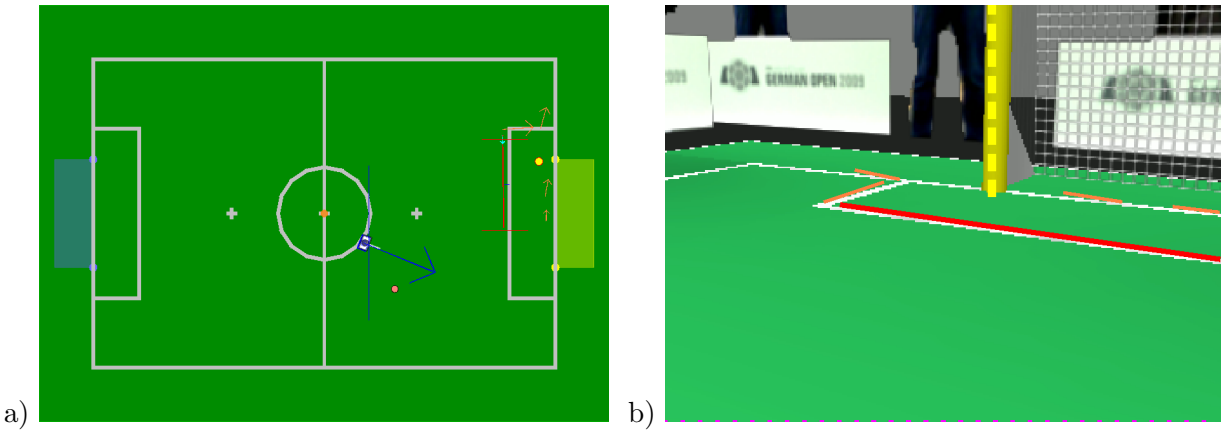


Figure 4.18: a) Robot pose components calculated from a single image. b) The vertical blue line marks a x-component calculated from line percept (large red line) and the two almost identical blue arrows mark calculated rotations using the line percept and the goal percept.

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

To address these problems the RobotPoseValidator, which provides the *RobotPose*, was introduced. It uses the *PotentialRobotPose* from the SelfLocator to relate line, goal and center circle percepts to their counter parts on the field. This way each line percept can be used to calculate a translational component (cf. Fig. 4.18) and the rotation of the robot pose. The results are filtered using a two-dimensional Kalman filter for the translational part and a one-dimensional Kalman filter for the rotation. The goal percepts are used to calculate additional rotations by assuming that the estimated position is correct. The center circle is used for additional translational components by assuming that the estimated rotation is correct. The resulting pose is

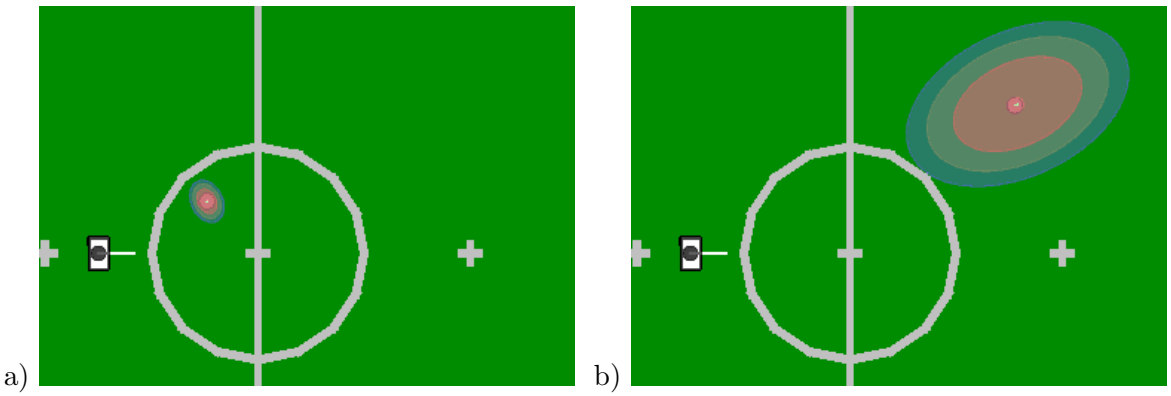


Figure 4.19: Ball model and measurement covariances for short (a) and medium (b) distances. The orange circles show the ball models computed from the probability distribution. The larger ellipses show the assumed covariances of the measurements.

fairly accurate, without the noise from the particle filter. In cases such as a mismatch of the goal posts – i. e. if the distance between the seen and estimated position is too large –, the provided pose is considered to be invalid. In such a situation, the behavior can react by looking for field lines and goal posts to improve the localization.

4.2.3 Ball Tracking

Having used a particle filter to estimate the velocity and position of the ball in recent years [31], the module BallLocator now uses Kalman filters to derive the actual ball motion, the *BallModel*, given the perception described in Sect. 4.1.5.5. 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 was naturally handled by the particle filter, deserves some attention when using a Kalman filter. Instead of only one, we use twelve multivariate Gaussian probability distributions to represent the belief concerning the ball. Each of these distributions is used independently for the prediction step and the correction step of the filter. Effectively, there are twelve Kalman filters running in every frame. Only one of those distributions is used to generate the actual ball model. That distribution is chosen depending on how well the current measurement, i. e. the position the ball is currently seen at, fits and how small the variance of that distribution is. That way we get a pretty accurate estimate of the ball motion while being able to quickly react on displacements of the ball, for example when the ball is moved by the referee after being kicked off the field.

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

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

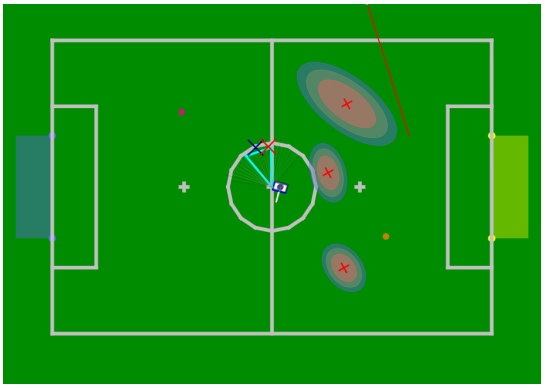
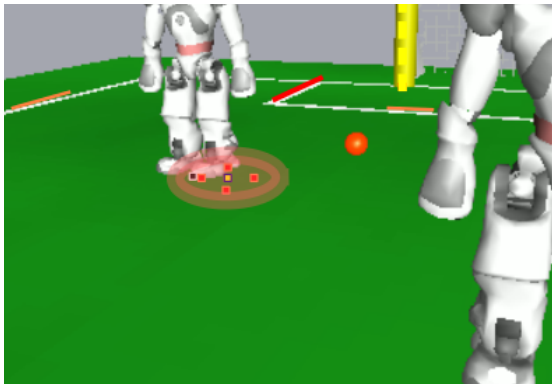
a)  b) 

Figure 4.20: a) The ellipses are the error ellipses of the observed robots and indicate the covariance of each robot. They are stretched because the observing robot changed its direction. b) The red σ-points are projected on the camera image. The black dot is the measured pixel and the red ellipse shows equiprobable points around the orange dot, which is the mean of the σ-points.

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. 4.19).

4.2.4 Robot Tracking

To track robots that are recognized via vision, we use an unscented Kalman filter for each detected robot since the transformation from the image coordinates to the relative position on the field is nonlinear. Actually it is not a real Kalman filter, since we did not implement a motion model yet. So we are currently assuming that all recognized robots do not move but we add a significant noise in each cycle so that we can react faster to the robot's movements. Thereby motion is considered as noise. This assumption has become very inaccurate and needs to be reworked in the future, since by now, most of the SPL teams have a quite fast gait.

The robots' environment is partially observable. The challenging part of the RobotLocator is to match recognized robots with robots in the model. The basic idea is to calculate the Mahalanobis distance of the detected robots to the robots in the model and always match those robots with the lowest distance. This approach would work perfectly for fully observable environments. The Mahalanobis distance as developed in [23] can be calculated with the dot product $\langle ., . \rangle$ as follows:

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

In fact, almost never every robot is tracked at the same time. So it is difficult to find out whether a detected robot correlates with a robot in the model or is a new robot. To solve this problem, we introduced the Euclidean distance as a criterion to determine whether we update an existing robot or add a new robot to our model. If the Euclidean distance of the recognized robot is too high, it will not update the previously tracked robot. Another difficulty is losing track of robots we did not see for a long time. They will not be tracked any longer if the approximated probability of an area around the mean is below a certain threshold.

The detailed robot tracking algorithm is shown in algorithm 4. The notation $tr.\mu$ refers to the

current mean of robot $tr \in \mathcal{T}$.

The motion model g only subtracts the current odometry offset (O_x, O_y, O_α) from the relative position of the tracked robots:

$$g(x_t) = \begin{pmatrix} \cos O_\alpha & \sin O_\alpha \\ -\sin O_\alpha & \cos O_\alpha \end{pmatrix} \cdot x_t - \begin{pmatrix} O_x \\ O_y \end{pmatrix}$$

The additional prediction noise covariance matrix R_t consists of the motion noise of the observed robot

$$\Sigma(\sigma_{m_x}, \sigma_{m_y}) = \begin{pmatrix} \sigma_{m_x}^2 & 0 \\ 0 & \sigma_{m_y}^2 \end{pmatrix}$$

and the odometry noise covariance of the observing robot Σ_O. We cannot apply an exact model of the odometry noise here as it cannot be represented by a covariance matrix. So we have to approximate the noise. We assume that the potential position of the observed robot spreads more on a circular path, when the observing robot rotates. The approximation first calculates the standard deviation of the noise in x and y directions for an observed robot in front of the observing robot with the euclidean distance d and then rotates the corresponding covariance matrix with the angle of the observed robot. The standard deviations of the noise covariance matrix are:

$$\sigma_{o_y} = d \tan \sigma_o O_\alpha \text{ and } \sigma_{o_x} = d - \sqrt{d^2 - \sigma_{o_y}}$$

The covariance matrix Σ_O results from the rotation around the observed robot's angle in the observing robot's coordinate system α:

$$\Sigma_O = \Sigma(\alpha, \sigma_{o_x}, \sigma_{o_y}) = \begin{pmatrix} \cos\alpha & \sin\alpha \\ -\sin\alpha & \cos\alpha \end{pmatrix} \begin{pmatrix} \sigma_{o_x}^2 & 0 \\ 0 & \sigma_{o_y}^2 \end{pmatrix} \begin{pmatrix} \cos\alpha & \sin\alpha \\ -\sin\alpha & \cos\alpha \end{pmatrix}^T$$

In the last equation, you also see a notation we used in the algorithm: $\Sigma(\alpha, \sigma_x, \sigma_y)$ is a covariance matrix of independent random variables x and y rotated by α.

The only parameter we have to define in these equations is σ_o, which should be between 0 and 1 and indicates the reliability of the odometry data. If σ_o is 0, the covariance matrix will be 0. You can see the result for a robot that turned around in figure 4.20 a).

The measurement model h projects the relative position on the field onto the camera image. You can see the σ-points and the corresponding error ellipse of the resulting covariance projected on the image in figure 4.20 b). There is no complicated measurement noise model. We manually tuned the parameters σ_{s_x} and σ_{s_y}.

4.2.5 Ultrasonic Obstacle Detection

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

To compute a robust model, all ultrasonic measurements are added to a grid (cf. Fig. 4.21a) which rasterizes the robot's local environment. The grid currently used has a resolution of 45×45

Algorithm 4 Robot Tracking Algorithm

Initialization:
$\mathcal{T} \leftarrow$ list of previously tracked robots (initially empty)
$\mathcal{P} \leftarrow$ list of currently perceived robots
$u_t \leftarrow$ translational and rotational odometry offset

Motion Update:
for all $tr \in \mathcal{T}$ **do**
  $\alpha \leftarrow$ angle to $tr.\mu_{t-1}$
  $R_t \leftarrow \Sigma(0, \sigma_{m_x}, \sigma_{m_y}) + \Sigma_O$
  $\mathcal{X}_{t-1} \leftarrow \sigma$-points of $tr.\mu_{t-1}$
  $\overline{\mathcal{X}}_t^* \leftarrow \{x_t | x_{t-1} \in \mathcal{X}_{t-1} \wedge x_t = g(u_t, x_{t-1})\}$
  $tr.\overline{\mu}_t \leftarrow \frac{1}{|\overline{\mathcal{X}}_t^*|} \sum_{x_t \in \overline{\mathcal{X}}_t^*} x_t$
  $tr.\overline{\Sigma}_t \leftarrow Cov(\overline{\mathcal{X}}^*) + R_t$
end for

Sensor Update:
for all $pr \in \mathcal{P}$ **do**
  $z_t \leftarrow$ pixel below robot pr on the camera image
  $f_t \leftarrow z_t$ projected on the field
  $tr \leftarrow \text{argmin}_{tr \in \mathcal{T}} (f_t - tr.\overline{\mu}_t)(tr.\overline{\Sigma}_t^{-1}(f_t - tr.\overline{\mu}_t))$
  if $euclideanDist(tr.\overline{\mu}_t, f_t) \leq$ maxAllowedDist **then**
    if tr is not updated yet **then**
      $Q_t \leftarrow \Sigma(0, \sigma_{s_x}, \sigma_{s_y})$
      $\overline{\mathcal{X}}_t \leftarrow \sigma$-points of $tr.\overline{\mu}_t$
      $\overline{\mathcal{Z}}_t \leftarrow \{z | x_t \in \overline{\mathcal{X}}_t \wedge z = h(x_t)\}$
      $\overline{z}_t = \frac{1}{\overline{\mathcal{Z}}_t} \sum_{z \in \overline{\mathcal{Z}}_t} z$
      $\Sigma_h \leftarrow \frac{1}{2} \sum_{z_t \in \overline{\mathcal{Z}}_t} (z_t - \overline{z_t})(z_t - \overline{z_t})^T$
      $\Sigma_{h\overline{\mathcal{X}}} \leftarrow Cov(h(\overline{\mathcal{X}}), \overline{\mathcal{X}})$
      $K_t \leftarrow \Sigma_{h\overline{\mathcal{X}}}^T (\Sigma_h + Q_t)^{-1}$
      $tr.\mu_t \leftarrow K_t(z_t - \overline{z_t})$
      $tr.\Sigma_t \leftarrow tr.\overline{\Sigma}_t - K_t \Sigma_{h\overline{\mathcal{X}}}$
    end if
  else
    $\alpha \leftarrow$ angle to f_t
    $pr.\Sigma_t \leftarrow \Sigma(\alpha, \sigma_{i_x}, \sigma_{i_y})$
    $pr.\mu_t \leftarrow f_t$
    Append pr to $\mathcal{T}$
  end if
end for
for all $tr \in \mathcal{T}$ **do**
  if $2\pi\sqrt{det(tr.\Sigma_t)} <$ minProbabilityAtMean **then**
    Remove tr from $\mathcal{T}$
  else if tr has not been updated but should be visible **then**
    $\alpha \leftarrow$ angle to the tracked robot
    $tr.\Sigma_t \leftarrow tr.\Sigma_t + \Sigma(\alpha, \sigma_{n_x}, \sigma_{n_y})$
  end if
end for

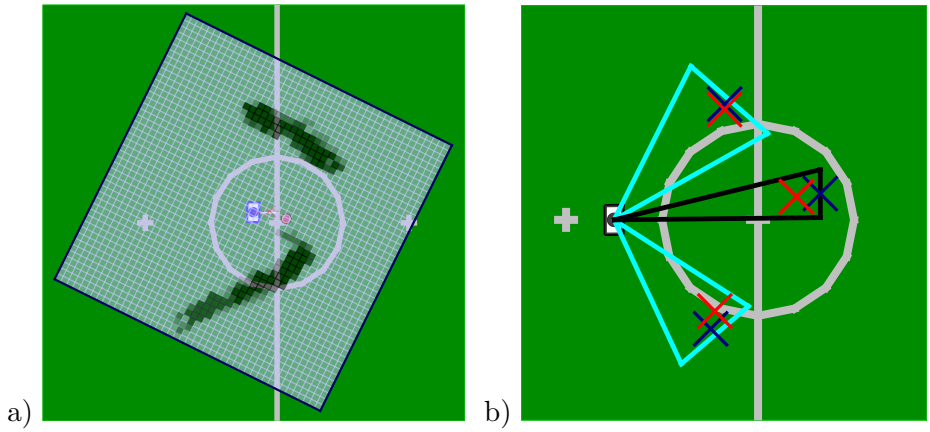


Figure 4.21: Modeling obstacle information: a) A robot's local occupancy grid. The cells' color depends on their counter n_o, the darker a cell, the higher the value of n_o. b) A different robot's *ObstacleModel*. Each triangle represents the angles to the borders of an obstacle. The cyan triangles result from obstacles that have been detected in the *USObstacleGrid*. The black triangle results from the *RobotsModel*. A red cross denotes the closest point of an obstacle. A blue cross denotes the center of an obstacle.

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

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

The module's output, the *USObstacleGrid*, contains the whole grid which is later processed by the ObstacleCombinator (cf. Sect. 4.2.7) that merges occupied cells to actual obstacles.

4.2.6 Arm Contact Recognition

The *ArmContactModel* provides basic information about the Nao's arm contact state. It is represented as a single Boolean value for the left arm and the right arm respectively. The ArmContactModelProvider uses the *FilteredJointData* to calculate the deviation between the desired joint request and the actual joint data. This deviation is accumulated over time when the robot is standing or walking and is decreased each frame by a constant value. The accumulated deviation is reset in case of a fall. A robot is considered to have arm contact if the accumulation exceeds a configurable threshold.

This simple approach provides solid information about the robot's arm contact state. Nevertheless, we plan to refine the model in our future work.

4.2.7 Obstacle Combination

In the previous sections, three different sources of obstacle information have been presented. All three have different advantages and drawbacks: The *USObstacleGrid* contains robust information about the presence of obstacles in the robot's vicinity. However, in addition to the low resolution, the Nao's ultrasonic sensors have only a limited range in which they operate reliably. Robots that lie on the ground cannot be perceived at all. The *RobotsModel* is less reliable as its computation relies on a proper perception of the robots' waistbands but it contributes important information about lying robots as well as about robots at medium distances. Finally, the *ArmContactModel* contains information about collisions with other robots at the left or the right, an area which is in many situations not covered by the other two models.

Other modules can access these models independent of each other, focusing on one special kind of information. But in some cases, such as obstacle avoidance or other behaviors, a combined model is desired. For this purpose, the ObstacleCombinator that provides the generic ObstacleModel has been developed. It converts all three models to a consistent model that contains information about the distances and angles to all current obstacles. An example is shown in Fig. 4.21b.

4.2.8 Largest Free Part of the Opponent Goal

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

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

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

The information provided in the *FreePartOfOpponentGoalModel* is currently used by the KickPoseProvider, which is described in Sect. 6.3.4.

4.2.9 Field Coverage

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

Since the introduction of the team-wide ball model (cf. Sect. 4.2.10), a robot only needs to actively search for the ball if all team members lost knowledge of the ball position. Consequently, if a robot is searching, it can be sure its team members do that, too. Knowing which parts of the field are visible to the team members, patrolling can happen dynamically in a much more organized way.

4.2.9.1 Local Field Coverage

To keep track of which parts of the field are visible to a robot, the field is divided into a very coarse grid of cells, each cell being a square that has a size of $\frac{1}{4}m^2$. To determine which of the cells are currently visible, the current image is projected onto the field. Then all cells whose center lies within the projected image are candidates for being marked as visible.

There are two conditions under which cells within the projected image are not marked as visible. There first one is pretty natural: There may be other robots obstructing the view to certain parts of the field. Depending on the point of view of the viewing robot, another robot may create an invisible area, a "shadow", on the field. No cell whose center lies within such a shadow is marked as visible. The second condition is completely artificial: All cells with centers more than $2m$ away from the viewing robot are not marked as visible. The information about other robots on the field gets very unreliable beyond that distance and missing a robot while potentially looking at cells which are far away would lead to marking large parts of the field as visible while they actually are not. An example local field coverage is depicted in Fig. 4.22.

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

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

4.2.9.2 Global Field Coverage

To make use of the field coverage grids of the other robots, each robot has to communicate its grid to its team mates. Given this year's field dimensions, 4 byte timestamps, and a cell's edge length of $\frac{1}{2}m$, there are $4\ bytes * \frac{4m \cdot 6m}{(\frac{1}{2}m)^2} = 384\ bytes$ which have to be sent in every team communication cycle in addition to all other communication our robots do during gameplay. Since the resulting bandwidth requirement would be beyond the bandwidth cap set by this year's rules, the timestamps are 'compressed' to one byte and the grid is not sent as a whole but in intervals. For each cell c that is part of the interval which is to be sent to the other robots, a one byte coverage value $coverage(c)$ is computed, such that $time - (255 - coverage(c)) \cdot tick$ roughly matches the timestamp stored for cell c, with *time* being the reference, i. e. the current, timestamp. Setting $tick = 300ms$ leads to coverage values which range up to $76.5seconds$ into the past. With $n = 3$ being the number of intervals the grid is divided into, the field coverage only adds $\frac{96}{3} + 4\ bytes = 36\ bytes$ to the team communication. The 4 extra bytes are for the

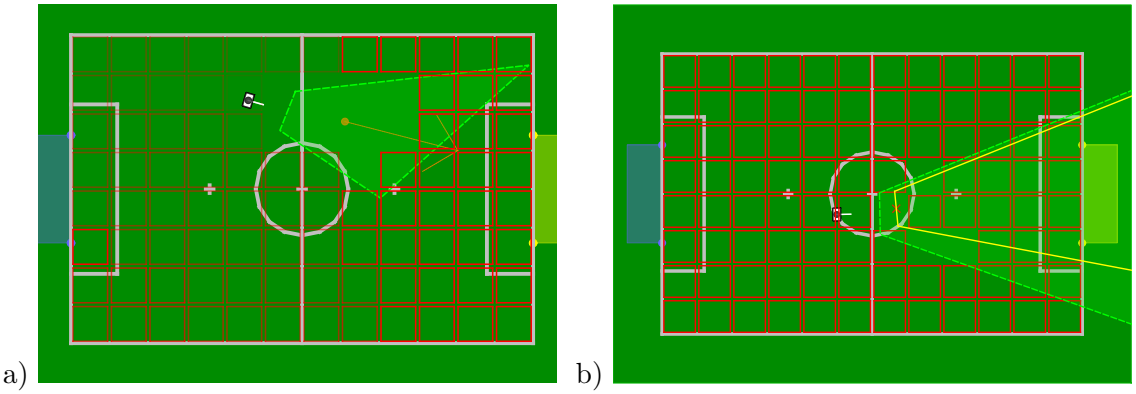


Figure 4.22: The local field coverage. a) The local grid after walking around on the field. The more intense the red borders of the cells are the less these cells are covered. The camera image projected onto the field is depicted by the dashed green line. b) An opponent robot prevents cells from being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

reference timestamp to regain the original timestamps of each cell.

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

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

Based on the values in the global field coverage grid, it has to be decided which parts of the field are covered by the robots and which parts are not, i. e. which parts are unknown to the whole team. The obvious way to do this is a threshold value, which marks the minimum coverage value a cell can have and still be considered to be covered, i. e.'known'. The only catch is that this threshold value has to be determined dynamically. With a fixed threshold we could easily end up with the entire field being considered uncovered or covered, although there were cells on the field which were covered so much worse than others, so that it would be desirable that the team 're-covers' those cells first and leave the rest of the field as it is. The situation is analogous to determining which parts of a gray scale image are black or white, so we applied the Otsu algorithm [25] to compute the threshold.

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

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

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

Instead of summing over all cells in the grid we build a histogram h of all coverage values, so

we can get rid of min and rewrite eq. 4.1 as

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

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

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

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

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

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

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

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

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

4.2.10 Combined World Model

Unlike some other domains, such as the Small Size League, the robots in the SPL do not have a common and consistent model of the world, but each of them has an individual world model, estimated on the base of its own limited perception. As such a model is a necessity for creating cooperative behavior, we implemented a combined world model that lets all robots of a team have an assumption of the current state of the world, even if parts of it were not seen by the robot itself. This assumption is consistent among the team of robots (aside from delays in the

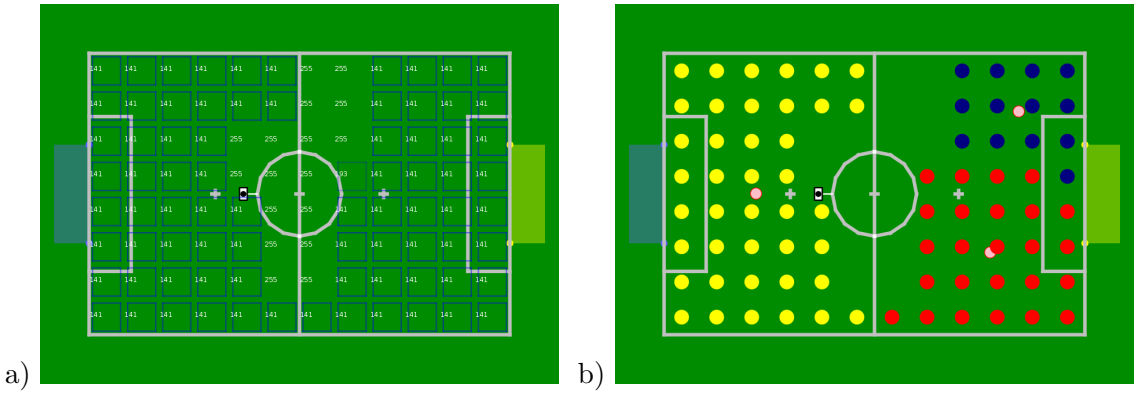


Figure 4.23: The global field coverage. a) The global grid obtained by merging the communicated local grids. b) The largest connected components of each cluster of uncovered cells. Each cell of such a component is marked by a yellow, blue or red circle to indicate the cluster assignment. The pink circles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

team communication) and consists of three different parts (*global ball model*, *positions of the teammates* and *positions of opponent players*) that are described below.

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

4.2.10.1 Global Ball Model

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

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

Based on these factors, a common ball model, containing an approximated position and velocity, is calculated.

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

4.2.10.2 Positions of Teammates

For the coordination of the team, for instance for the role selection or execution of tactics, it is important to know the current positions of all teammates. Computing those positions does not require special calculations such as filtering or clustering, because each robot sends its filtered position to the teammates via team communication. This means that each robot is able to get an accurate assumption of all positions of its teammates by listening to the team communication. Besides to the coordination of the team, the positions of the teammates are of particular importance for distinguishing whether ultrasound measurements belong to own or opponent players (cf. Sect. 4.2.10.3).

4.2.10.3 Positions of Opponent Players

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

Compared to only using the local model of opponent robots, which is 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" robots model, 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 are used that do not point to a teammate, i. e. all ultrasound measurements that are located near a teammate are excluded from the merging process.

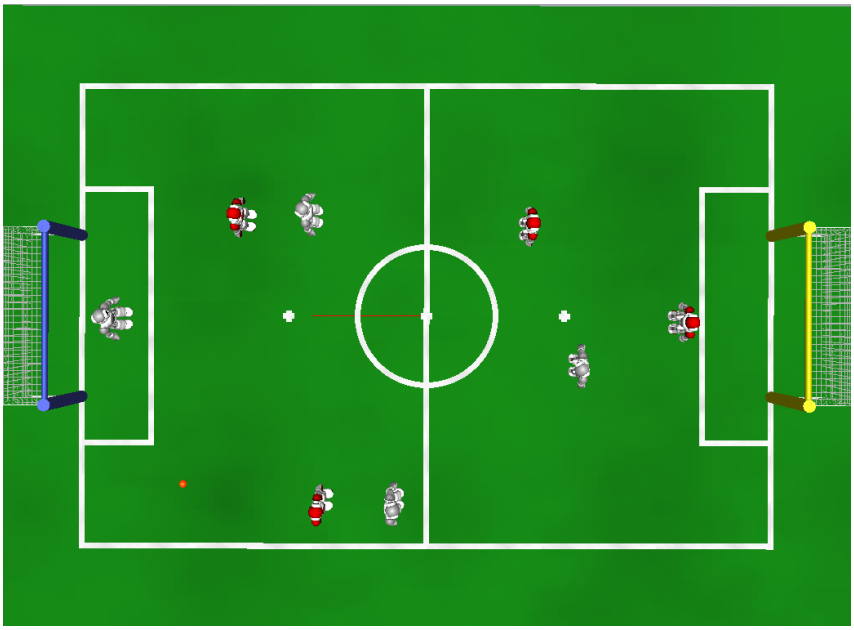


Figure 4.24: Overview of a playing situation where opponent robots are shown in red. The ball is located in the lower left quadrant.

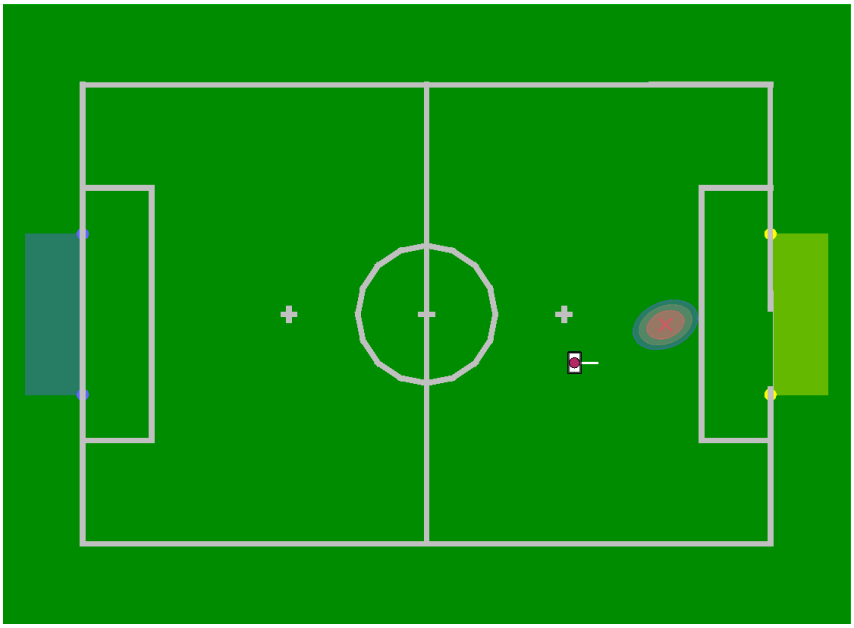


Figure 4.25: The local world model of the robot on the right side. The robot is only able to see one opponent robot. All other opponent robots and the ball could not be seen.

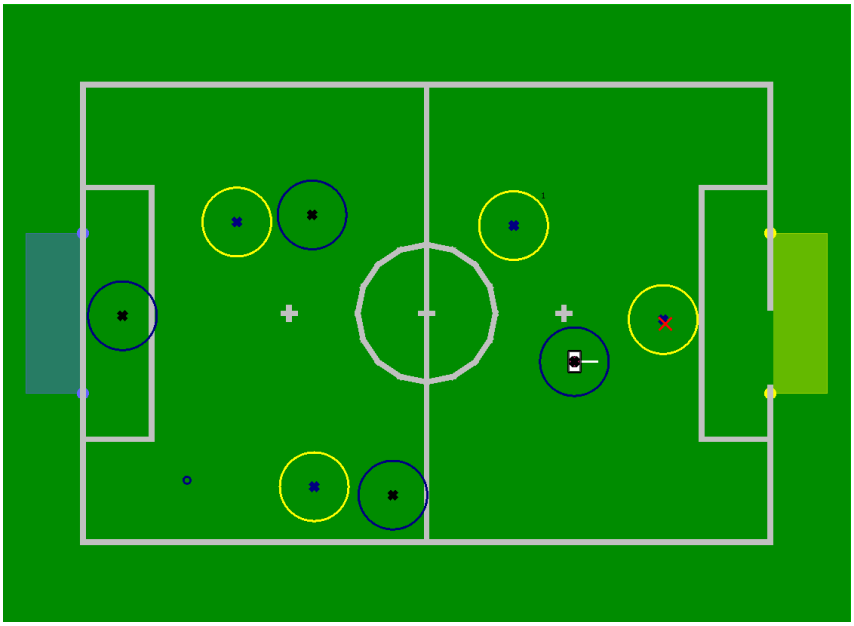


Figure 4.26: The combined world model for the robot on the right side. The blue circles show the own teammates, the yellow circles show the opponent players and the small blue circle shows the global ball.

After the clustering is finished, the positions of each cluster that represents a single opponent robot have to be reduced to a single position. For the reduction the covariances of the measurements are merged using the *measurement update* step of a Kalman filter. The result of the whole merging procedure is a set of positions, the *RobotsModel*, that represent the estimated positions of the opponent robots.

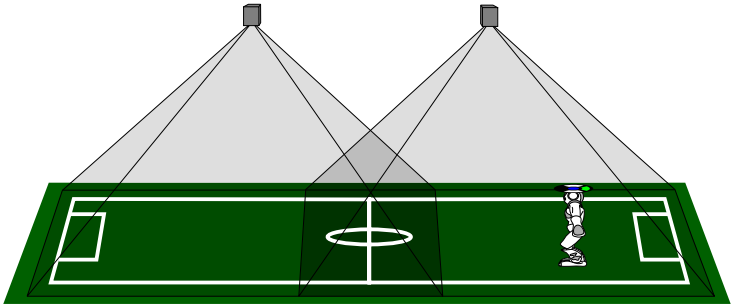
a)  b) 

Figure 4.27: Ground Truth: a) A Nao robot equipped with a colored pattern for global tracking. b) Two cameras with overlapping images are mounted above the field.

4.2.11 Ground Truth

As source for ground truth data, providing *RobotPose*, *BallModel* and *RobotsModel*, a global tracking system is used. A unique marker is fixed on the robot's head (cf. Fig. 4.27) and tracked by the standard vision system of the RoboCup Small Size League [37], processing images from a camera hanging above the field. The system provides the position as well as the rotation of the robot on the field.

In its default configuration, the standard vision system transmits UDP/IP packets to the multicast address 224.5.23.2. The robot and the vision system have to be configured to use the correct network device for this address. But the packets are not time synchronized with the robot's world model. Thus, we implemented a plugin for the SSL server that writes the ground truth data to a shared memory and a tool, the *SSLConnector*, that broadcasts this data via team communication. The advantage of this approach is that the B-Human team communication already provides synchronized timestamps for the received data.

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

The commands (cf. Sect. 8.8) *mr SSLVisionData TeamMateDataProvider* and *mr GroundTruthRobotPose GroundTruthProvider* will enable synchronized ground truth to provide the *GroundTruthRobotPose*. The command *mr GroundTruthRobotPose SSLVision* will enable the module SSLVision that reads the SSL vision server UDP packets. The estimated *RobotPose* can still be calculated by the RobotPoseValidator, both representations can be streamed to a connected SimRobot instance or stored in a log file. The module GroundTruthEvaluator provides some plots to compare the estimated and actual position. The module buffers the estimated models in order to compare them with the delayed ground truth data.

Chapter 5

Motion

The process *Motion* comprises the two essential tasks *Sensing* (cf. Sect. 5.1) and *Motion Control* (cf. Sect. 5.2). *Sensing* contains all modules (except camera modules) that are responsible for preprocessing sensor data and *Motion Control* all modules that compiles joint angles.

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

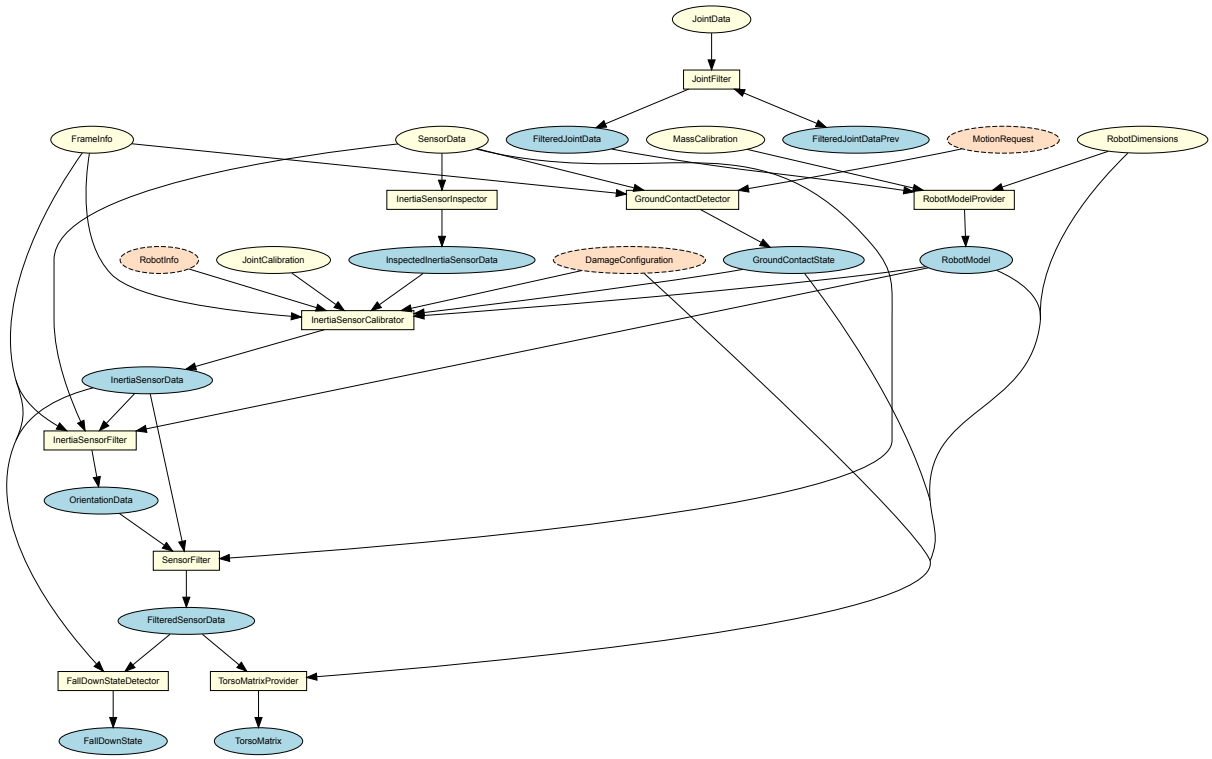


Figure 5.1: All modules and representations in the area *Sensing*. Blue ellipses mark representations provided by sensing modules, which are marked as white squares. White ellipses mark required representations from sensing modules. In contrast, red ellipses with a dashed outline mark required representations from other sources than sensing.

5.1 Sensing

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

The NaoProvider receives all sensor readings from the *NaoQi* module *libbhuman* (cf. Sect. 3.1), adds a configured calibration bias for each sensor and provides them as *SensorData* and *JointData*. In this matter it is important to ensure that these sensor readings do not contain unexpectedly large or small values, which may require special treatment in other modules.

Therefore, the *JointData* passes through the JointFilter module that provides the representation *FilteredJointData* (cf. Sect. 5.1.1). According to this the *SensorData* passes the modules InertiaSensorInspector, InertiaSensorCalibrator (cf. Sect. 5.1.4), InertiaSensorFilter (cf. Sect. 5.1.5) and also the SensorFilter, which provides the representation *FilteredSensorData*.

The module InertiaSensorCalibrator determines the bias of the readings from the IMU considering the *RobotModel* (cf. Sect. 5.1.3) and the *GroundContactState* (cf. Sect. 5.1.2). Thereby the *RobotModel* provides the position of the robot's limbs and the *GroundContactState* the information whether the robot's feet are touching the ground. Based on the *FilteredSensorData* and the *RobotModel* it is possible to calculate the *TorsoMatrix* (cf. Sect. 5.1.6), which describes a transformation from the ground to an origin point within the torso of the Nao. The representation *FilteredSensorData* is also considered to detect whether a robot has fallen down by the module FallDownStateDetector, which provides the *FallDownState* (cf. Sect. 5.1.7). Figure 5.1 shows all *Sensing* modules and the provided representations.

5.1.1 Joint Data Filtering

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

5.1.2 Ground Contact Recognition

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

Besides the InertiaSensorInspector, InertiaSensorFilter and the SensorFilter, the module GroundContactDetector that provides the *GroundContactState* is the only module that uses the unfiltered *SensorData*, since the *GroundContactState* is required by the InertiaSensorCalibrator. Hence it is not possible for the GroundContactDetector to use the *FilteredSensorData*.

In order to identify the *GroundContactState* the force sensing resistors in the Nao's feet and the electric current sensors of the leg motors are used. The *GroundContactState* is considered to be true if a weighted sum of these readings exceeds a certain threshold for a certain period of time. Since the sensors may be faulty and the Nao's force sensing resistors tend to be highly inaccurate the GroundContactDetector calculates confidence values that are taken into account as well. The confidence in the force sensing resistors is defined as the sum of their mean squared errors. This definition is motivated by the idea that trustworthy force sensing resistors will always provide more or less homogeneous measured data. However, the confidence of the electric current sensors is defined as the Nao's angle of tilt since the measured data becomes more inaccurate the more the Nao is tilted.

5.1.3 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 one joint. By considering the measured joint angles of the representation *FilteredJointData* the calculation of each limb is ensured by the consecutively computations of the kinematic chains. Similar to the inverse kinematic (cf. Sect.5.2.2) the implementation is customized for the Nao, i. e., the kinematic chains are not described by a general purpose convention such as Denavit-Hartenberg parameters to save computation time.

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

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

$\vec{r}_i$ is calculated for each limb by considering the representation *RobotModel* and the precalculated position of its CoM (relative to the limb origin). The limb CoM positions and masses provided in the file *masses.cfg* were taken from the Nao documentation by Aldebaran.

5.1.4 Inertia Sensor Data Calibration

The InertiaSensorInspector module is responsible for dropping corrupted IMU sensor readings from the *SensorData*. The result will be provided as the *InspectedInertiaSensorData*. Dropping of broken inertial sensor values is necessary, because some received sensor measurements cannot be explained by the legal noise of the sensor. This malfunction occurs sporadically and affects most of the sensor readings from the IMU board within a single frame (cf. Fig. 5.2). The broken sensor readings are detected through comparing the difference of each value and their predecessor to a predefined threshold. If a frame with broken sensor measurements is found, all sensor readings from the IMU board are ignored.

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

The gyroscope sensors are hard to calibrate, since their calibration offset depends of the sensor's temperature, which cannot be observed. The temperature changes slowly as long as the robot

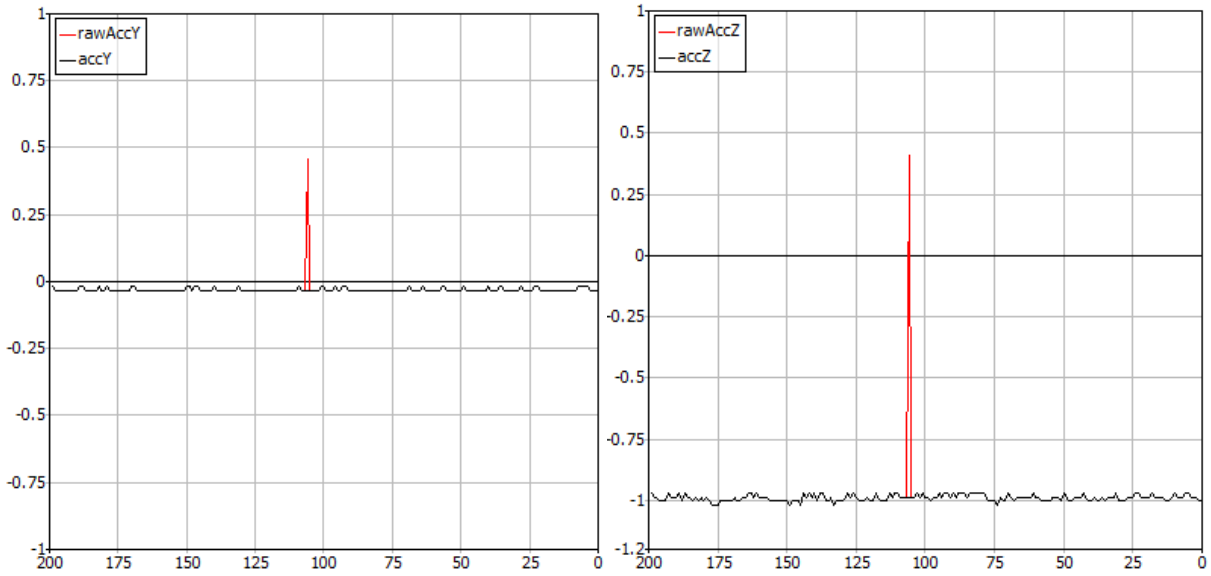


Figure 5.2: A typical broken inertia sensor data frame. The broken data was detected and replaced with its predecessor.

runs, so that it is necessary to redetermine the calibration offset constantly. Therefore, it is hypothesized that the robot has the same orientation at the beginning and ending of walking phases while all gyroscope values are collected during each phase. If the robot does not walk, the gyroscope values are collected for one second instead. The average of the collected values is filtered through a simple one-dimensional Kalman filter and the result is used as offset for the gyroscope sensor. The collection of gyroscope values is restricted to slow walking speeds and the ground contact state is used to avoid collecting gyroscope values in unstable situations.

A similar method is applied to the acceleration sensors. When the robot is standing it is assumed that both feet are evenly on the ground to calculate orientation of the robot's body and expected acceleration sensor readings. These expected acceleration sensor readings are considered to determine the calibration offset.

5.1.5 Inertia Sensor Data Filtering

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

A three-dimensional rotation matrix, which represents the orientation of the robot's torso, is used as the estimated state in the Kalman filtering process. In each cycle the rotation of the torso is predicted by adding an additional rotation to the estimated state. The additional rotation is computed using the readings from the gyroscope sensor. When these readings are missing since they have been dropped (cf. Sect. 5.1.4), the alteration of the feet's rotation relative to the torso is used instead.

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

of the robot rests evenly on the ground is used to update the estimate.

The resulting orientation (cf. Fig. 5.3) is provided as *OrientationData*.

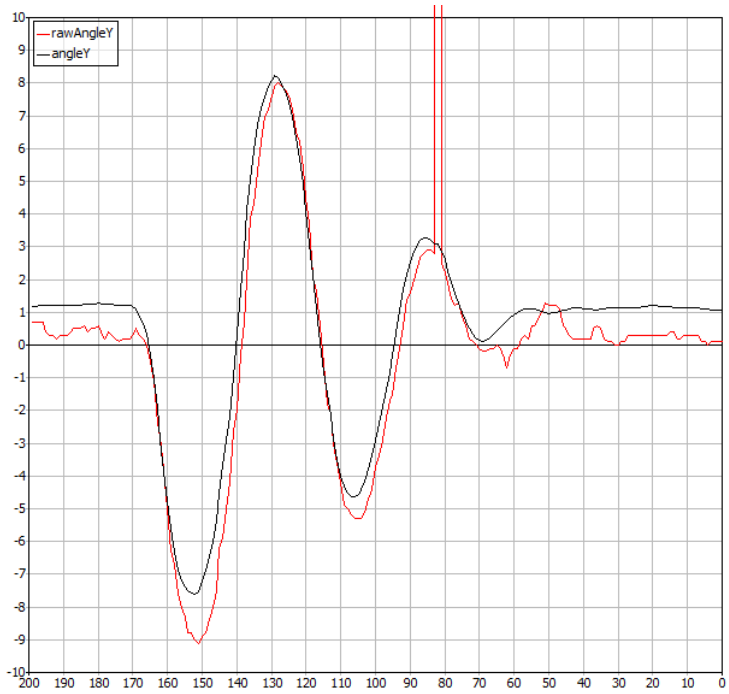


Figure 5.3: Comparison of the estimated pitch angle $angleY$ provided by the InertiaSensorFilter module and the pitch angle $rawAngleY$ that has been computed on IMU board.

5.1.6 Torso Matrix

The *TorsoMatrix* describes the three-dimensional transformation from the projection of the middle of both feet on the ground up to the center of hip within the robot torso. Additionally, the *TorsoMatrix* contains the alteration of the position of the center of hip including odometry. Hence, the *TorsoMatrix* is used by the *WalkingEngine* for estimating the odometry offset. The *CameraMatrix* within the *Cognition* process is 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. 5.1.5). The estimated rotation is represented as rotation matrix R.

$$f_l = -R \cdot t_l \tag{5.2}$$
$$f_r = -R \cdot t_r \tag{5.3}$$

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

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

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

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

The change of the position of the center of hip is determined by using the inverted torso matrix of the previous frame and concatenating the odometry offset. 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.

5.1.7 Detecting a Fall

Although we try our best at keeping our robots upright, it occasionally still happens that one of them falls over. In such a case it is helpful to switch off the joints and bring the head into a safe position to protect the robots hardware from unnecessary damage. The task of detecting such a situation is realized by the FallDownStateDetector which provides the *FallDownState*. After detecting a fall we only have a very small amount of time to get the necessary actions done. This is, because we have to be completely sure, that the fall is inevitable. It would have fatal consequences if a fall was mistakenly detected and instead of protecting the joints by switching them off, the robot would suffer an unnecessary collapse. For that reason, fall detection is split up into 2 phases, the first of which is to detect that the robot is staggering. If that is the case we bring the head into a save position, which can be done without risking the robots stability. In the second phase we detect whether the robot exceeds a second threshold in its tilt angle. If that is the case, we power down the joints to a low hardness. It has shown that this is a better approach than switching them off completely, because deactivated joints tend to gain too much momentum before hitting the floor. To determine the robot's tilt angle, the FallDownStateDetector utilizes the *FilteredSensorData*.

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

5.2 Motion Control

The B-Human motion control system generates all kinds of motions needed to play soccer with a robot. They are split into the different type of motions *walking*, *standing*, *kicking*, and *special actions*. The walking motion and a corresponding stand are dynamically generated by the WalkingEngine (cf. Sect. 5.2.1). Some kicking motions are generated by the Bézier-spline and inverse kinematics-based kick engine (BLAME, former BIKE) that is described in [24]. All further motions that are created by static joint angle patterns, are provided by the module SpecialActions (cf. Sect. 5.2.3). All three modules generate joint angles. The WalkingEngine provides the *WalkingEngineOutput* and the *WalkingEngineStandOutput*. The module BLAME provides the *BikeEngineOutput* and the module SpecialActions provides the *SpecialActionsOutput*. According to the *MotionRequest* the MotionSelector (cf. Sect. 5.2.5) calculates which motions to execute and how to interpolate between different motions while switching from one to another. This information is provided by the representation *MotionSelection*. If necessary, two modules calculate their joint request and the MotionCombinator (cf. Sect. 5.2.5) combines these according to the *MotionSeletion*. The MotionCombinator provides the final *JointRequest* and additional

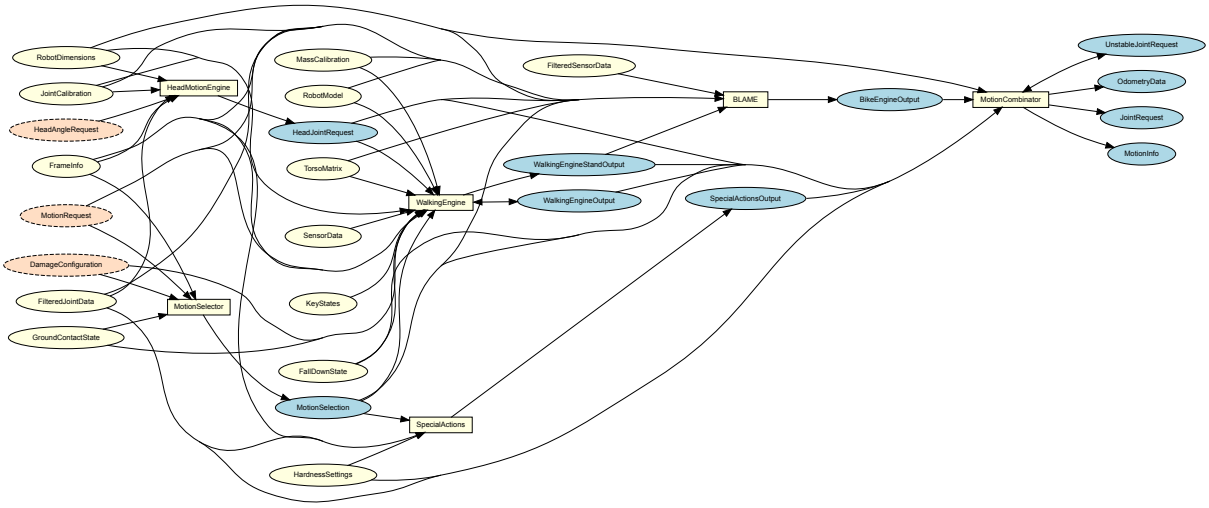


Figure 5.4: All modules and representations in the area *Motion Control*. Blue ellipses mark representations provided by motion control modules that are marked as white squares. White ellipses mark required representations from motion control modules. In contrast, red ellipses with a dashed outline mark required representations from other sources than motion control.

data such as *OdometryData* and *MotionInfo*. Figure 5.4 shows all modules in the area *Motion Control* and the representations they provide.

5.2.1 Walking

Walking motions and joint angles for standing are generated by the module WalkingEngine. Therefore, the WalkingEngine provides two sets of joint angles *WalkingEngineOutput* and *WalkingEngineStandOutput*. The distinction between these two sets exists for historic reasons. In fact, both sets are equal in order to trick the MotionCombinator (cf. Sect.5.2.5). The WalkingEngine handles transitions between walking and standing on its own.

The walking motions are generated using the computationally inexpensive model of an inverted pendulum to compute an intended trajectory for the robot's center of mass. Whenever the robot is walking or standing the WalkingEngine observes the center of mass motion that is actually happening using the *TorsoMatrix* and *RobotModel*. The observed center of mass motion is considered to adjust the intended center of mass trajectory to assure stability for further walking steps. This approach was already released and described in further detail in [6].

The generated walk and stand stance can be customized using the configuration file *Config/walking.cfg* (or individually for each robot using *Config/Robots/<robot>/walking.cfg*). The parameters are mainly grouped as *stand*, *walk*, *observer* and *balance* parameters. The *stand* parameters control the stand stance, which will also be used as initial pose for walking motions. The *walk* parameters control how to use the inverted pendulum in order to generate the walking motion. The *observer* parameters and *balance* parameters control how to observe the real center of mass motion and how to apply the observed motion in order to adjust the inverted pendulum, which is used to generate the walking motions.

5.2.1.1 In-Walk Kicks

Besides walking and standing the WalkingEngine has also minor kicking capabilities. The tasks of walking and kicking are often treated separately, both solved by different approaches. In

presence of opponent robots, such a composition might waste precious time as certain transition phases between walking and kicking are necessary to ensure stability. A common sequence is to walk, stand, kick, stand, and walk again. Since direct transitions between walking and kicking are likely to let the robot stumble or fall over, the WalkingEngine is able to carry out sidewards and forward kicks within the walk cycle.

Such an in-walk-kick is described individually for each kick type (forwards or sidewards) by a number of parameters, which are defined in the configuration files of the *Config/Kicks/* directory. On the one hand one configuration defines the sizes and speeds of the step before and during the kick (*setPreStepSize* and *setStepSize*), on the other hand a 6-D trajectory (three degrees in both translation and rotation; *setLeg* and *proceed*). This 6-D trajectory overlays the original trajectory of the swinging foot and thereby describes the actual kicking motion (cf. Fig. 5.5). In doing so the overlaying trajectory starts and stops at 0 in all dimensions and also in position and velocity. Thus, the kick retains the start and end positions as well as the speeds of a normal step. The instability resulting from the higher momentum of the kick is compensated by the walk during the steps following the kick.

a) b)

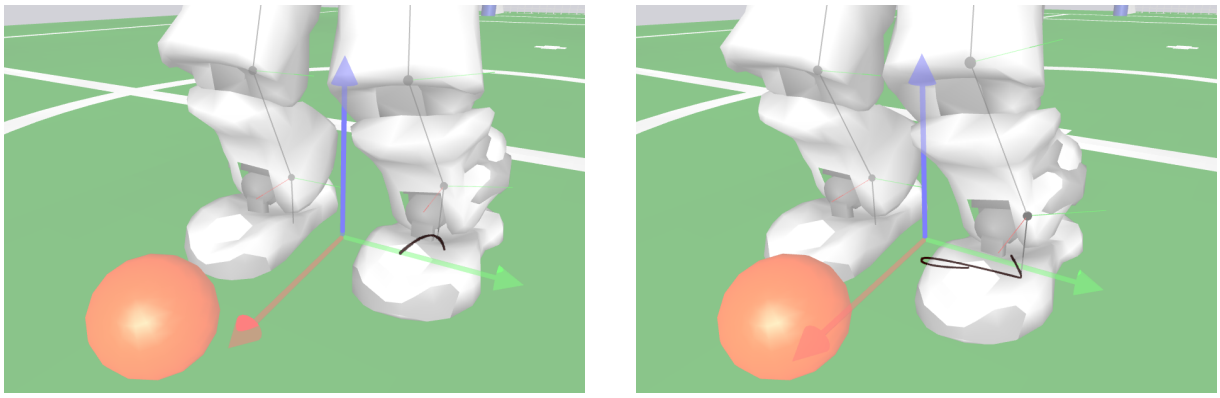


Figure 5.5: Visualization of swing foot trajectories (dark curves) while the robot approaches the ball. a) A swing foot trajectory for walking forwards and b) a modified swing foot trajectory for performing a sidewards in-walk kick to the right.

5.2.2 Inverse Kinematic

The inverse kinematics for the ankles of the Nao are a central component of the module WalkingEngine and are used for generating dynamic motions using BLAME. In general, they are a handy tool for generating motions but solving the inverse kinematics problem analytically for the Nao is not straightforward because of two special circumstances:

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

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. 5.6.

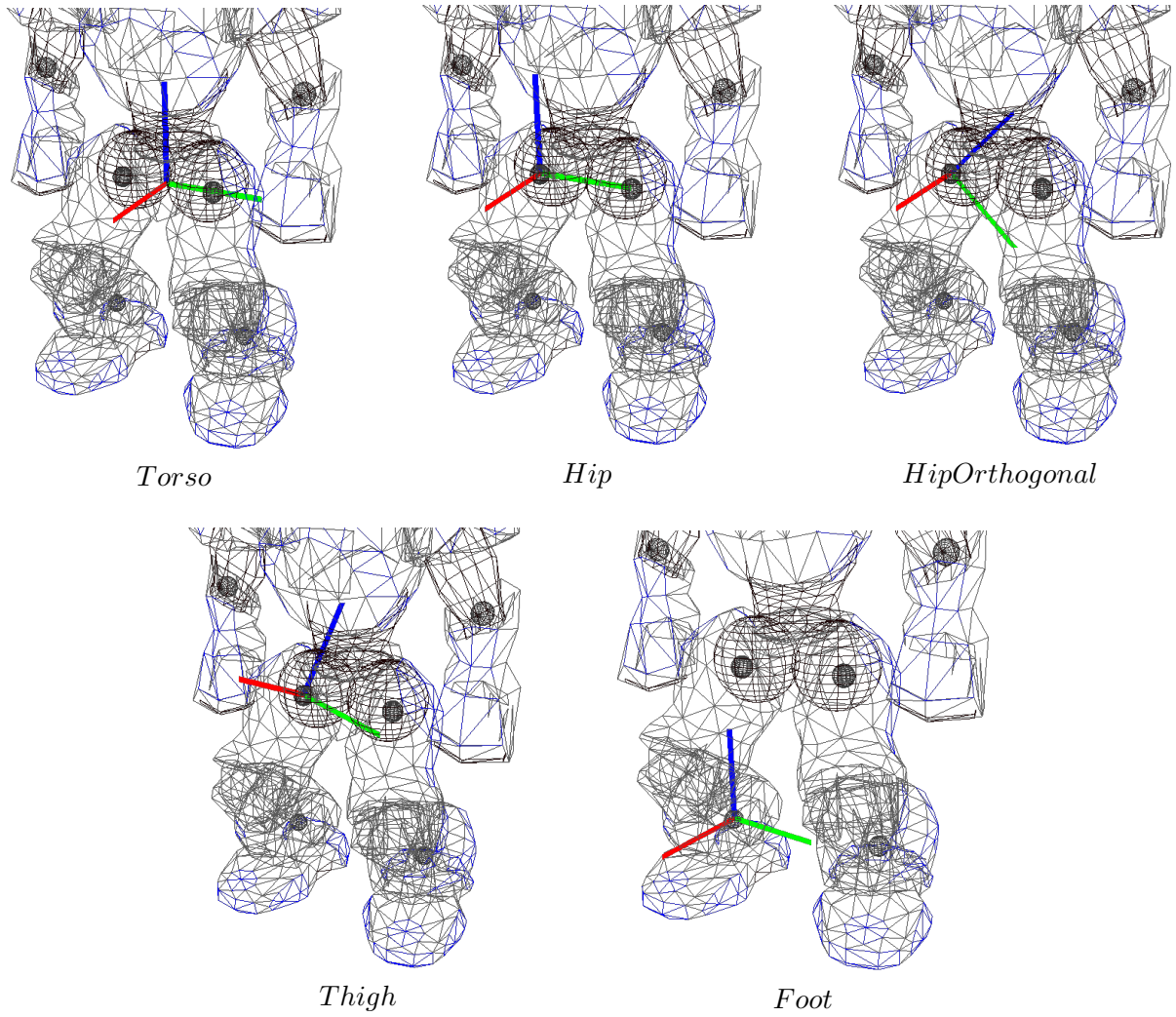


Figure 5.6: Visualization of coordinate frames used in the inverse kinematic. Red = x-axis, green = y-axis, blue = z-axis.

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]

$$Foot2Hip = Trans_y\left(\frac{l_{dist}}{2}\right) \cdot Foot2Torso \tag{5.6}$$

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

$$Foot2HipOrthogonal = Rot_x(\frac{\pi}{4}) \cdot Foot2Hip \tag{5.7}$$

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

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.

$$HipOrthogonal2Foot = Foot2HipOrthogonal^{-1} \tag{5.8}$$

The limbs of the leg and the knee form a triangle, in which an edge equals the length of the translation vector of $HipOrthogonal2Foot$ (l_{trans}). Because all three edges of this triangle are known (the other two edges, the lengths of the limbs, are fix properties of the Nao) the angles of the triangle can be computed using the law of cosines (5.9). Knowing that the angle enclosed by the limbs corresponds to the knee joint, that joint angle is computed by equation (5.10).

$$c^2 = a^2 + b^2 - 2 \cdot a \cdot b \cdot \cos\gamma \tag{5.9}$$

$$\gamma = \arccos \frac{{l_{upperLeg}}^2 + {l_{lowerLeg}}^2 - {l_{trans}}^2}{2 \cdot l_{upperLeg} \cdot l_{lowerLeg}} \tag{5.10}$$

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

$$\delta_{knee} = \pi - \gamma \tag{5.11}$$

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

$$\delta_{footPitch1} = \arccos \frac{{l_{lowerLeg}}^2 + {l_{trans}}^2 - {l_{upperLeg}}^2}{2 \cdot l_{lowerLeg} \cdot l_{trans}} \tag{5.12}$$

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

$$\delta_{footPitch2} = \text{atan2}(x, \sqrt{y^2 + z^2}) \tag{5.13}$$

$$\delta_{footRoll} = \text{atan2}(y, z) \tag{5.14}$$

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

$$\delta_{footPitch} = \delta_{footPitch1} + \delta_{footPitch2} \tag{5.15}$$

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

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

$$HipOrthogonal2Thigh = Thigh2Foot^{-1} \cdot HipOrthogonal2Foot \tag{5.16}$$

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

$$Thigh2Foot = Rot_x(\delta_{footRoll}) \cdot Rot_y(\delta_{footPitch}) \cdot Trans_z(l_{lowerLeg}) \cdot Rot_y(\delta_{knee}) \cdot Trans_z(l_{upperLeg}) \tag{5.17}$$

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

$$Rot_{Hip} = Rot_z(\delta_z) \cdot Rot_x(\delta_x) \cdot Rot_y(\delta_y) = \begin{pmatrix} c_y c_z - s_x s_y s_z & -c_x s_z & c_z s_y + c_y s_x s_z \\ c_z s_x s_y + c_y s_z & c_x c_z & -c_y c_z s_x + s_y s_z \\ -c_x s_y & s_x & c_x c_y \end{pmatrix} \tag{5.18}$$

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

$$\frac{-r_{01}}{r_{11}} = \frac{\cos\delta_x \cdot \sin\delta_z}{\cos\delta_x \cdot \cos\delta_z} = \frac{\sin\delta_z}{\cos\delta_z} = \tan\delta_z \tag{5.19}$$

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

$$\delta_z = \delta_{hipYaw} = \text{atan2}(-r_{01}, r_{11}) \tag{5.20}$$

$$\delta_y = \delta_{hipPitch} = \text{atan2}(-r_{20}, r_{22}) \tag{5.21}$$

At last the rotation by 45 degrees (cf. eq. 5.7) has to be compensated in joint space.

$$\delta_{hipRoll} = \delta_x - \frac{\pi}{4} \tag{5.22}$$

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

The computation described above can lead to different resulting values for the hip yaw joints. From these two joint values a single resulting value is determined, in which the interface allows to set 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

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

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.

5.2.3 Special Actions

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

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

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

An example of a special action:

```
motion_id = stand
```

[5] Ignored

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

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

5.2.4 Head Motions

Besides the motion of the body (arms and legs), the head motion is handled separately. This task is encapsulated within the separate module HeadMotionEngine for two reasons. The first reason is that the modules WalkingEngine and BLAME manipulate the center of mass. Therefore this modules need to consider the mass distribution of the head *before* its execution in order to compensate for head movements. The other reason is that the module smoothens the head movement by limiting the speed as well as the maximum angular acceleration of the head. In encapsulating this, the other motion modules do not have to concern about this.

The HeadMotionEngine takes the *HeadAngleRequest* generated by the CameraControlEngine (cf. Sect. 4.1.2) and produces the *HeadJointRequest*.

5.2.5 Motion Selection and Combination

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

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

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

In addition, some special action motions have a higher priority than others, e. g. stand-up motions, which is handled as a special case in the MotionSelector. These are executed immediately and it is not possible to leave the motion or the motion engine before it is finished.

Chapter 6

Behavior Control

The part of the B-Human system that performs the action selection is called *Behavior Control*. The behavior was modeled using the Extensible Agent Behavior Specification Language (XABSL) [22]. The main module – the BH2011BehaviorControl – provides the representations *MotionRequest*, *HeadMotionRequest*, *BehaviorLEDRequest* and *SoundRequest* (cf. Fig. 6.1). It is accompanied by the KickPoseProvider (cf. Sect. 6.3.4), the CameraControlEngine, which switches between both cameras, the LEDHandler, which transforms the abstract *BehaviorLEDRequest* to actual LED settings, the MatchStatisticsProvider (cf. Sect. 6.4), the GoaliePoseProvider, and the BH2011HeadControlEngine (cf. Sect. 6.3.3).

This chapter begins with a short overview of XABSL and how it is used in a simple way. Afterwards, it is shown how to set up a new behavior. Both issues 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 2011. Finally, we describe our match statistics logging as well as *b-script*, a new behavior specification language that was already used to describe our penalty behavior.

6.1 XABSL

XABSL is a language that has been designed to described an agent's behavior as a hierarchy of state machines. For using it, it is important to know its general structure. In XABSL, the following base elements are used: *options*, *states*, *decisions*, *input symbols*, and *output symbols*.

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 *initial state*. Inside a state, an action can be executed and a decision about a state transition (within the option) can be made. An action consists of the modification of output symbols (for example head motion requests or walk requests) and the call of other options. Input and output symbols are needed to create actions and decisions within a state. Input symbols are used for the decisions and output symbols are used for the actions. Actions may only consist of symbols and simple arithmetic operations. Other expressions cannot be used in XABSL. All symbols are implemented in the actual robot code and range from math symbols to specific robot symbols.

This structure is clarified with an example:

```
option example_option
{
  initial state first_state
```

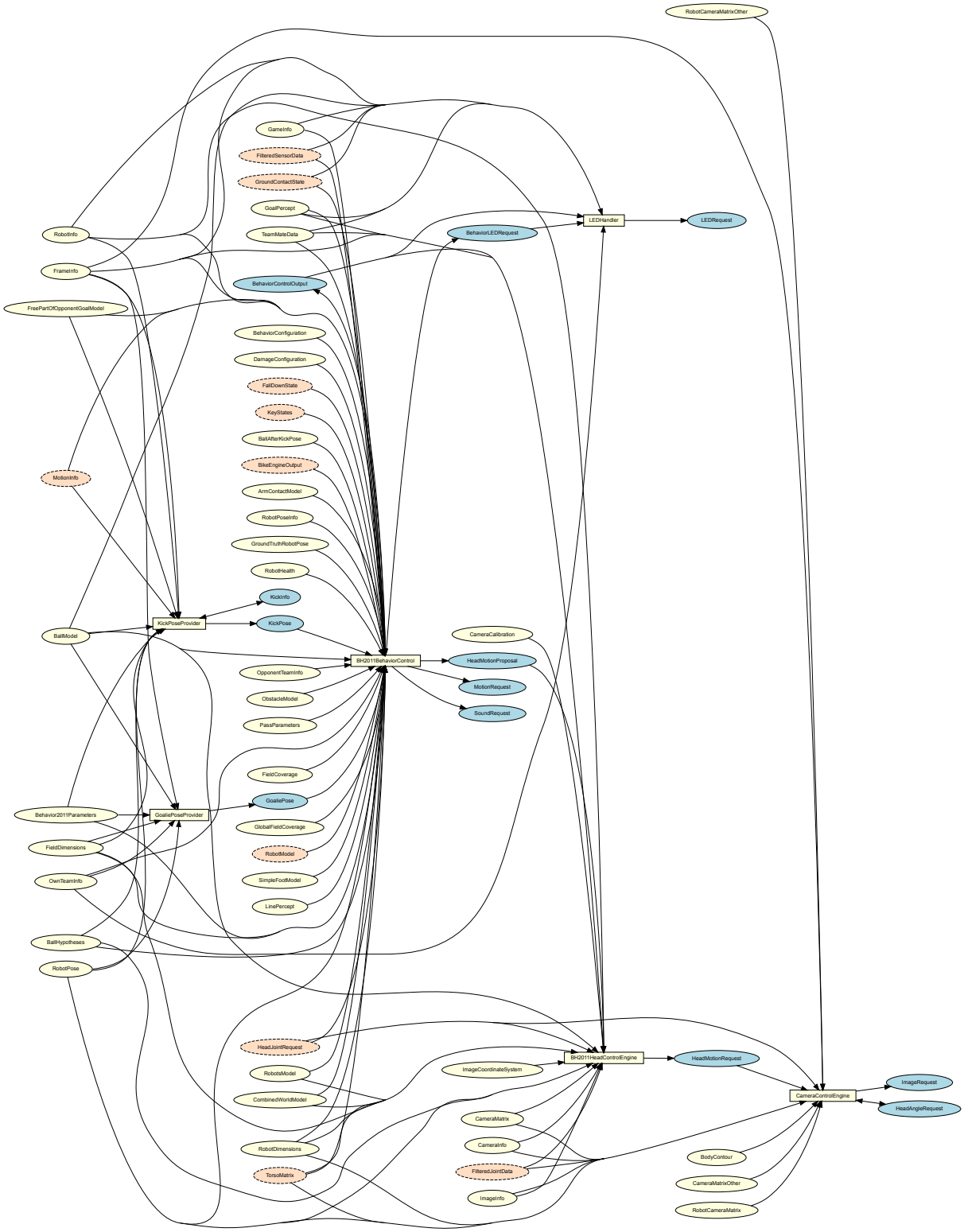


Figure 6.1: Behavior Control module graph. White squares represent behavior control modules. Blue ellipses mark representations provided by behavior control modules. White ellipses mark representations provided by cognition modules, whereas red ellipses with a dashed outline mark representations provided by motion modules.

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

  state second_state
  {
    action
    {
      secondOption();
    }
  }

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

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

```
option example_common_decision
{
  common decision
  {
    if(boolean_expression)
      goto first_state;
    else if(boolean_expression)
      goto second_state;
  }
  initial state first_state
  {
    decision
    {
      else if(boolean_expression)
        goto second_state;
```

```
      else
        stay;
    }
    action
    {
      output_symbol = input_symbol * 3
    }
  }

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

Options can have parameters that can be used similar to input symbols by using an @ in front of the parameter name:

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

option example_suboption
{

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

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

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

```
option example_superior_option
{
  initial state first_state
```

```
    {
      decision
      {
        if(action_done)
         goto second_state;
      }
      action
      {
        example_suboption();
      }
    }
    state second_state
    {
      action
      {
        output_symbol = input_symbol;
      }
    }
  }

  option example_suboption
  {

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

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

There are two special states to use the cooperation mechanism of XABSL [26, 27]. The *capacity state* is used to ensure that a certain state can only be executed by a specified number of robots at the same time. This is used, for instance, by the role change (cf. Sect. 6.3.8) where the state *striker* has a capacity of one. To let more robots act synchronously, the so-called *synchronized state* can be used.

6.2 Setting Up a New Behavior

To set up a new behavior, it is necessary to create a new directory in *Src/Modules/BehaviorControl*. This directory will contain the new behavior. To structure the behavior, it is advisable to create some subdirectories, such as directories for *Options* and *Symbols* (this is not mandatory). The option directory can be divided into subdirectories such as skills, head motions, or roles. It

is convenient to place all symbols in the *Symbols* directory. To create symbols, a header file, a source file, and a XABSL file are necessary. The files should be used for groups of symbols such as head symbols, ball symbols, and so on. This way it is easier to locate symbols.

After creating all symbols needed, it is necessary to create a file called *agents.xabsl* in the behavior directory, where all options needed are listed. This file is important, because it is used to start the behavior. Next to the *agents.xabsl*, the following files have to be available in the newly created behavior directory: *<name>BehaviorControl.cpp* and *<name>BehaviorControl.h*. To understand how these files work, look at the comments in the corresponding files available in the *BH2011BehaviorControl* directory. It is important to include all symbols needed within the *<name>BehaviorControl.cpp*. After these files are created correctly, the *<name>BehaviorControl.h* has to be added to the file *CognitionModules.cfg* that can be found in the directory *Config/Processes*.

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

6.3 Behavior Used at RoboCup 2011

In the following, the behavior used at RoboCup 2011 will be described. The corresponding option graph is shown in Fig. 6.2. It can be split into several parts which are explained hierarchically.

The root option of the whole behavior is the option *start_soccer* that is also used to suppress the immediate stand up after starting the software. When the chest button is pressed the first time after starting the software, the robot stands up (this mechanism is suppressed for a simulated robot). The option executes *head_control*, *body_control*, *display_control*, *official_button_interface*, and *penalty_control* in parallel.

The *official_button_interface* is used to switch between the different game states by using the chest button and foot bumpers of the robot. The *body_control* is used to invoke the corresponding options (e. g. the *initial*, *ready*, *set* or *playing* state). The *head_control* executes a specific head control mode. The option *penalty_control* is responsible for the correct behavior relating to the penalty mode. The option *display_control* is provides visible information about the running behavior.

All options, the strategies used, the dynamic role selection, and the several roles are described in detail in the following sections.

6.3.1 Button Interface

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

Besides the game state changes, the button interface includes decisions for changing the team color and the kick-off team. These can be changed by pressing the left or the right foot button, respectively. It is also possible to switch the game state to penalty shootout. Thereby, the

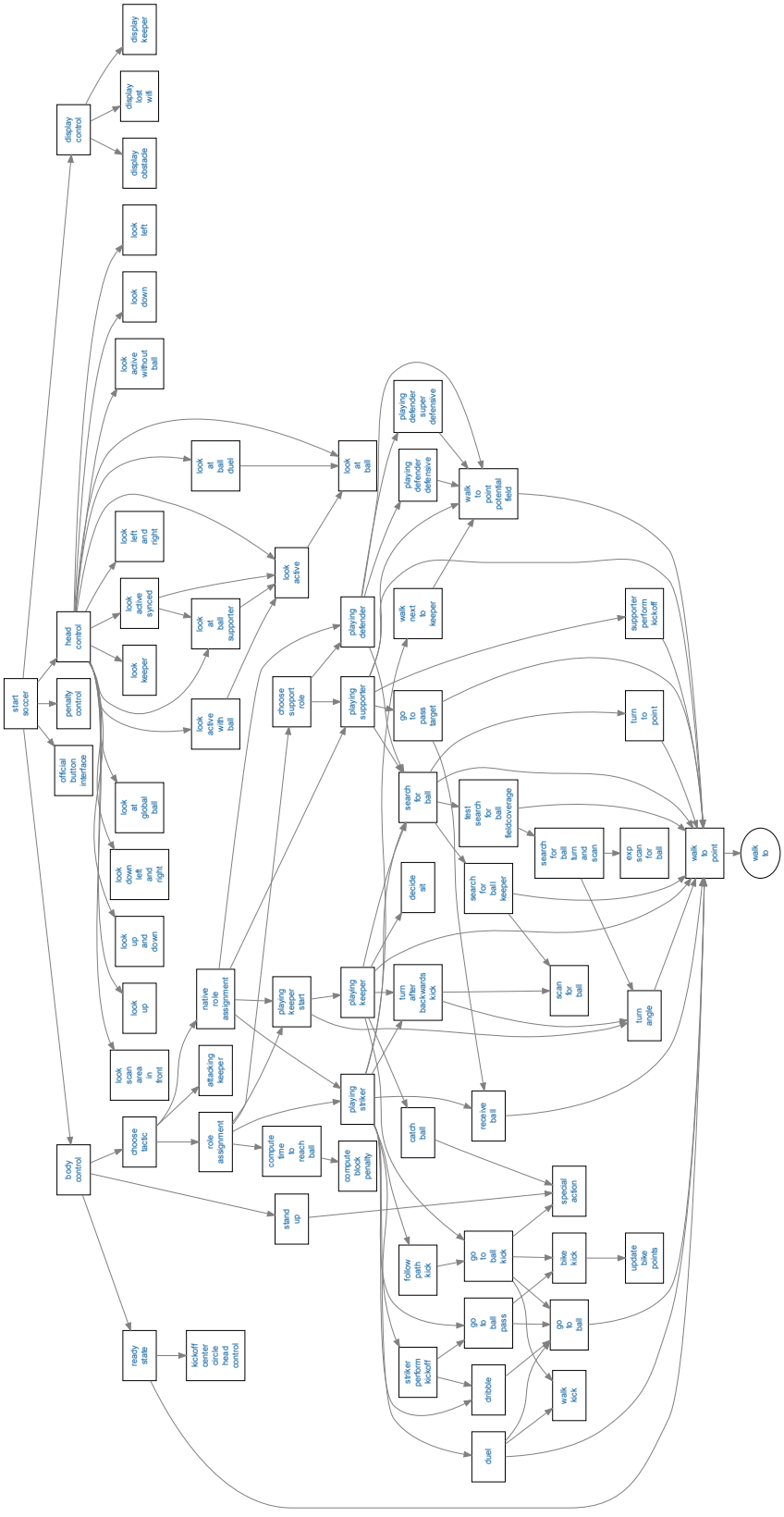


Figure 6.2: Option graph of the behavior used at RoboCup 2011

robots don't need any special software or to be rebooted to get ready. This is also done by pressing the right foot button which does not only cycle through own and opponent kick-off but also through penalty shootout. In addition to the button interface, the game state can also be controlled by the official GameController that overwrites all settings made via the button interface. The button interface is implemented according to the rules of 2011.

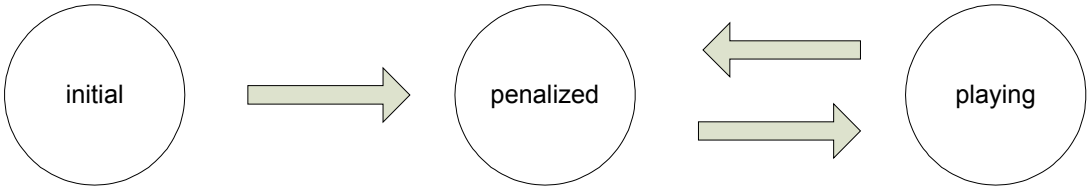


Figure 6.3: States and transitions of the button interface

6.3.2 Body Control

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

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

state_ready. The robot behaves as specified in (cf. Sect. 6.3.7).

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

state_playing. When the game state changes to playing, the robots begin to play soccer. Therefore the option *choose_tactic* is executed. This option invokes options that gather some information about the current game situation and it executes *role_assignment*, which is responsible for the dynamic role selection. After a robot has taken its role, it plays soccer according to the role definition. Various roles, tactics, and the role selection are described in detail in the following sections.

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

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

6.3.3 Head Control

The head control controls the angles of the two head joints, and thereby the direction in which the robot looks. In our behavior, the head control is an independent option that runs in parallel to the body control option. In general, it takes requests from body control to either execute a predefined motion pattern or to activate the module BH2011HeadControlEngine to dynamically control the head motions instead. The requests are passed from body control via the *head.control_mode* symbol. As long as the symbol does not change, the selected head control option is executed continuously. The problem of designing and choosing head control modes is that the information provided by our vision system is required from many software modules with different tasks. For instance the GoalPerceptor, BallPerceptor, LinePerceptor, and the RobotPerceptor provide input for the modeling modules (Sect. 4.2) that provide localization, the ball

position, and positions of other robots. However, providing images from all relevant areas on the field is often mutually exclusive, i. e., when the ball is located directly in front of the robot's feet, it cannot simultaneously look at the goal. In addition to only being able to gather some information at a time, speed constraints come into play, too. The solution to move the head around very fast to look at important areas more often proves impractical, since not only the images become blurred above a certain motion speed, but also because a high motion speed has a negative influence on the robot's walk due to the weight 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 (*panTilt*), a position on the field (*targetOnGround*), or a position relative to the robot (*target*). The head control modes used in our behavior are:

look_active. In this mode, the robot looks to the most interesting point which is calculated by the BH2011HeadControlEngine. Our solution for an effective head motion are points of interest. During the game, there are different important points on the field such as the goals, field lines, and the ball position. But the actual interest in a certain point and the unimportance of another one changes from situation to situation. So at a particular time, the robot needs to localize itself and it is unnecessary to look at the ball. In another case, the robot wants to walk to the ball which it does not see, when it is advisable to look at the estimated ball position, the robot got from its teammates. So we collect all points that seem to be important. Besides from the position of these points, we collect the time and presence of eye-contact. To calculate the actual point of interest, we use different factors, e. g. the location of the point, the possibility to focus it just by moving the head, the previous point of interest, and the so-called *ball factor*. The *ball factor* expresses the importance of the ball. The higher the factor, the more the robot looks at the ball. The information whether the upcoming point of interest is the ball or not is communicated with the other teammates. This information is used to improve the teams overall ball localization, i. e. the global ball model (cf. Sect. 4.2.10.1). Whenever the strikers point of interest is not the ball, the other field players observe it.

look_active_synced. This mode is only used by the supporter and the defender. Whenever the striker is using active vision and its next point of interest is not the ball, the defender and supporter players observe it instead. In the other case, the defending and supporting field players use normal active vision with an *ball factor* of 0.

look_at_ball. Looks at the estimated ball position. This mode is used after finding the ball in an earlier search to confirm the ball position. It is also used when the Nao is near the ball and prepares a kick or starts to dribble with the ball. We then collect information about the ball position over a longer period of time. When the robot has lost the ball, it starts to search for it using *look_active* with the highest possible *ball factor*. If the ball is rolling towards the robot, it tries to follow it, so that the ball is always in the upper part of the range of view.

look_at_global_ball. 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.

look_at_ball_supporter. This head control contains special handling for supporter and defender robots. If the ball is covered by other robots, the robot uses the global ball estimate instead of its own estimate, assuming that the ball becomes visible again. Often, there are situations where an accidental, not intended pass occurs, so that the ball is now nearer to

the supporter. This should result in a role change but to do this, it is necessary for the supporter to see the ball. When the ball was not seen, the supporter uses the global ball estimate, which can be imprecise in the vicinity of robot. To handle such situations, it additionally scans the area around the global ball estimate.

look_at_ball_duel. Since the normal *look_at_ball* tries to keep the ball in the upper part of its range of view, there is almost no chance to see the opponents feet, which is essential for smart dueling decisions. This head control simply tries to keep the ball in the lower part of the image.

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

look_down / look_up Moves the head down or rather up. Often used as a basic or finishing position in different states.

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

look_up_and_down. Moves the head up and down in a continuous motion until the Nao notices the ball. Combined with moving around the body, the robot has a good overview about its environment. This mode is used in different situations, e. g. when trying to improve the self localization of the keeper after performing a diving motion.

6.3.4 Kick Pose Provider

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

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

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

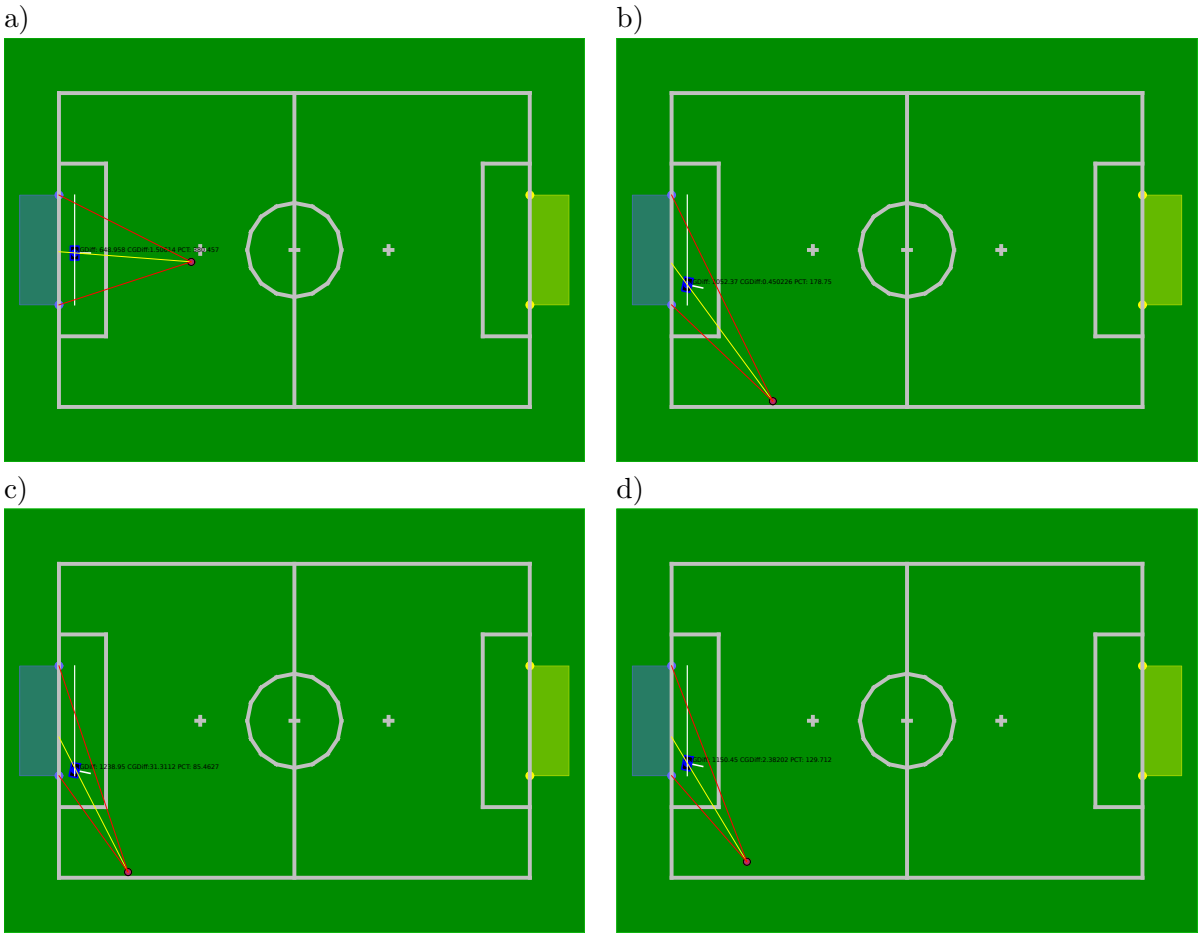


Figure 6.4: It takes more ball movement to provoke a positional change (blue robot pose) from image a) to b) than from image c) to d). The keeper is able to get back to the middle faster than the other way round.

The kick is usually directed towards the center of the opponent goal, however there are two exceptions. If the goal is not free, i. e. a robot is blocking the goal, the kick is directed towards the center of the largest free part of the opponent goal (cf. Sect. 4.2.8). The other exception is when the ball is lying too close to the opponent ground line and not directly in front of the goal. In that case, the striker kicks the ball in front of the opponent goal where either the supporter is waiting or where it has a better angle for kicking the ball into the goal in a second attempt.

6.3.5 Goalie Pose Provider

The rather simple module GoaliePoseProvider calculates a worthwhile defending pose for the keeper inside of its own penalty box. Furthermore, it also advises the keeper to stay where it is or to reach a new pose. The pose itself is calculated by the bisecting line of the angle between the ball and the goalposts. This line is intersected with an imaginary line with a fixed length (outreaching the goalposts for a few cm), parallel to the keeper's own ground line. Intersections are clipped if the calculated value exceeds the length of the line.

Since the ball is moving a lot during a game, resulting in continually different intersections, a new pose is only provided if the difference between the old and the current calculated pose is greater than a threshold, which is figured out dynamically. This value gets smaller the more the pose drifts away from the center to the left/right and gets bigger the other way around again.

This allows the keeper to get back to the middle of the goal faster (cf. Fig. 6.4). Note that changing the robot's position takes a long time, so it is necessary to perform as few positional changes as possible.

6.3.6 Path Planner

In recent years, the team used a reactive obstacle avoidance based on ultrasonic measurements. Such a reactive avoidance is not optimal because the robot can only react on the just measured obstacle, which might lead to the problem that an avoidance ends in a position directly near another obstacle. Another problem of the reactive avoidance is that it is executed by walking sidewards until the distance to the measured obstacle is high enough to walk forward again, which can take a very long time (cf. Fig. 6.7). Moreover, it is possible that the other robot, that was detected as obstacle, uses the same kind of obstacle avoidance which could lead to the situation that both robots avoid in the same direction until they leave the field.

For this reason, we developed a path planner that is based on the combined world model (cf. Sect. 4.2.10). The algorithm used for finding a path is the extended bidirectional Rapidly-Exploring Random Tree (RRT). This non-optimal algorithm is based on the random exploration of the search space and works on continuous values.

In general the Rapidly-Exploring Random Tree algorithm [19] works as follows:

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

This algorithm builds up a tree that quickly expands in few directions of the search space, because in each step of the algorithm the tree is enlarged by one edge in a random direction (cf. Fig. 6.5). For this general algorithm, different variants exist (e. g., RRT-Extend [19], RRT-Connect [12], and RRT-Bidirectional [20]) of which the extend and bidirectional[1] ones were used here (cf. Fig. 6.6). Using the extend variant restricts the expansion towards the random position to a given distance that is the same for each expansion, which has a direct influence on the expansion of the tree, whereas the used bidirectionally has no influence on the tree itself but it is used to decrease the run time. This is achieved by creating two separate trees, one beginning from the start point and one from the end point. Another modification is to replace the random position by the target position or a waypoint of the last found path with a given probability. Using this modifications helps to avoid oscillations of the path, for instance, if there is an obstacle on the direct way to the target.

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

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

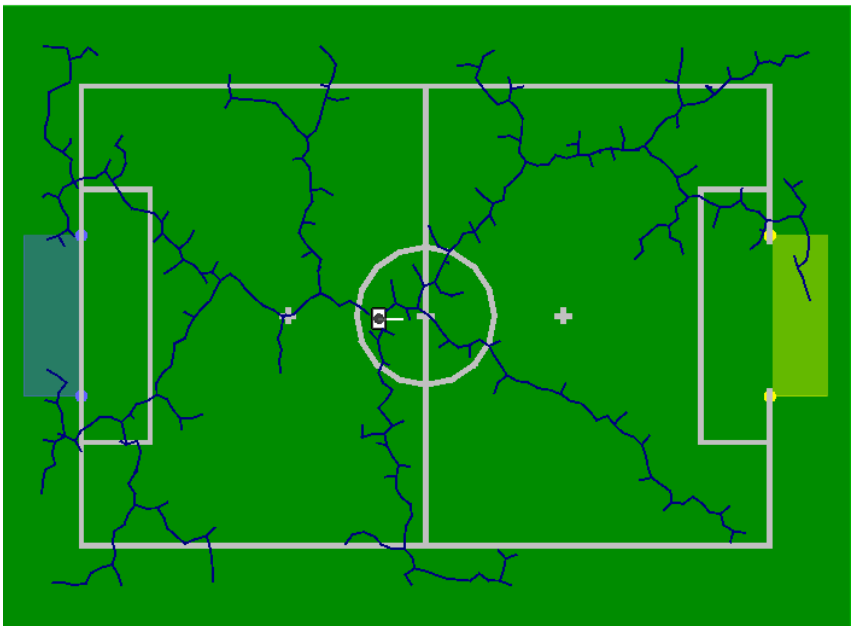


Figure 6.5: A tree build up with the RRT algorithm.

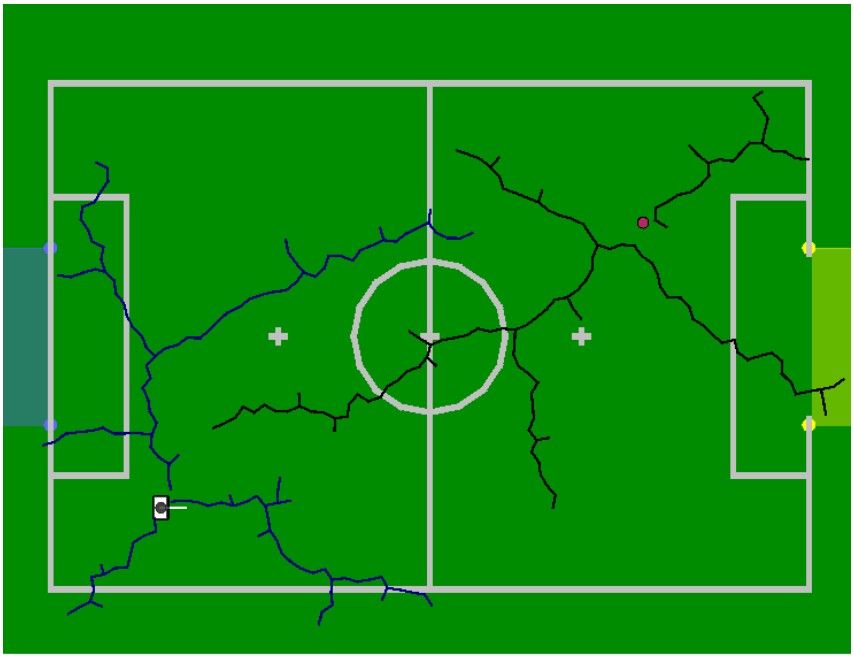


Figure 6.6: Two trees build up with a combination of the RRT-EXTEND and the RRT-BIDIRECTIONAL algorithm.

around the obstacles. Only if an obstacle is too near, the robot needs to walk sidewards and decrease the forward speed.(cf. Fig. 6.8)

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

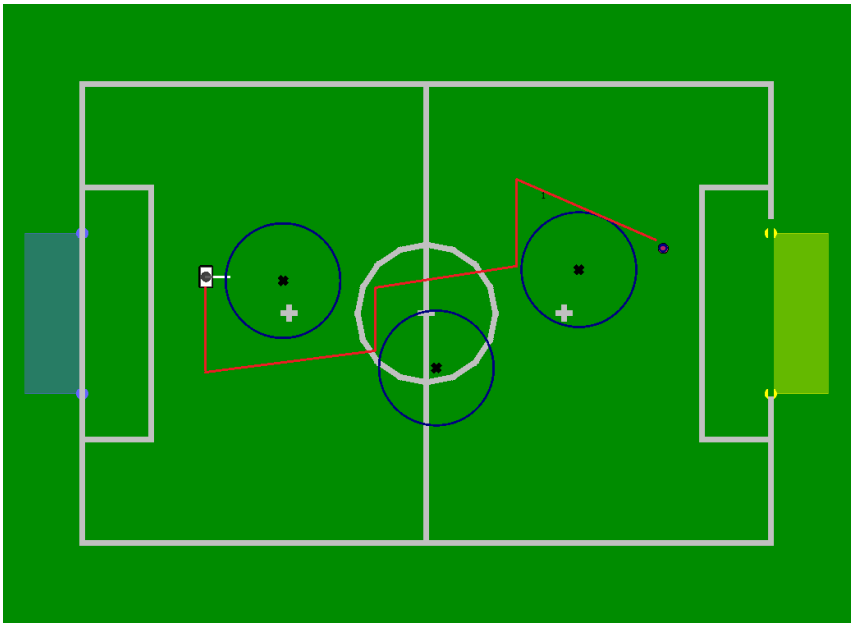


Figure 6.7: The path used to walk towards the ball using the reactive avoidance.

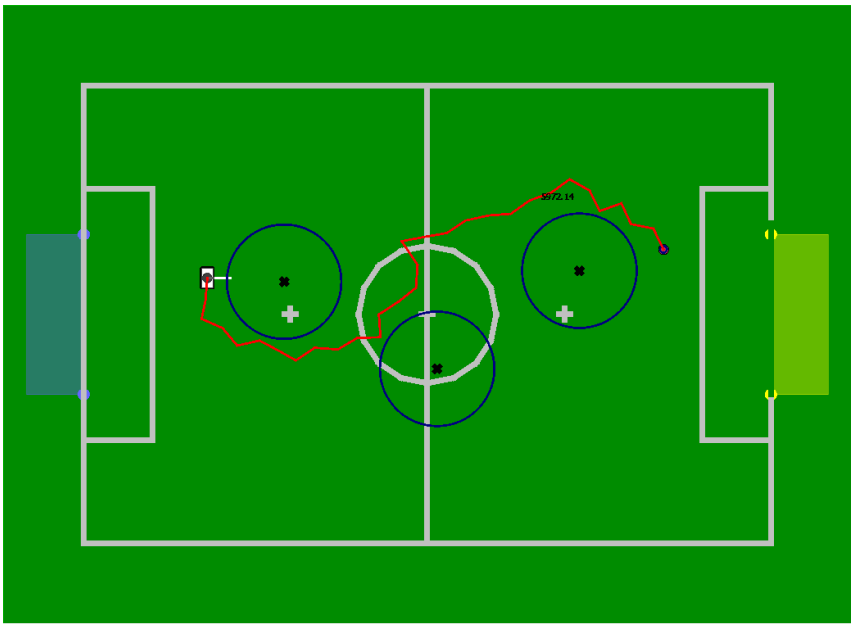


Figure 6.8: The path used to walk towards the ball using the RRT path planner.

6.3.7 Kickoff Configuration

In recent years, kickoff situations have been handled as specialized suboptions of the normal soccer playing behavior. As the number of robots per team has been increased in 2011, the specification of different, well-coordinated plays during this set-piece has become more complex, too. This is why we decided to outsource the kickoff handling to a set of new XABSL options, combined with a new configuration mechanism that allows to specify the team's behavior in a compact way. In this configuration, each kickoff variant K is a tuple $\langle w, P, A, A_p \rangle$ that consists of a weighting w, which determines its likelihood of being selected (cf. Sect. 6.3.7.1), a set of player poses P (cf. Sect. 6.3.7.2), and the actual action A – together with possible parameters A_p – that has to be carried out (cf. Sect. 6.3.7.3).

Obviously, two different sets of kickoff variants are specified, one for the case that the own team has to carry out the kickoff and one for the opponent's kickoff. The kickoff options control the behavior of all robots during the whole *READY* and *SET* game states as well as during the beginning of *PLAYING*, the transition to the normal soccer playing behavior depends on certain factors (cf. Sect. 6.3.7.3).

6.3.7.1 Kickoff Selection

The availability of different kickoff variants requires proper selection and negotiation mechanisms to ensure a consistent team behavior. For this purpose, we implemented a two-stage approach.

In a first step, when the *READY* signal has been received, each robot selects a random (with respect to the configured weightings w) kickoff variant K_r and communicates its selection to all teammates.

Subsequently, all robots wait until they have received the choices of all teammates but not longer than a certain timeout (a few seconds, configured depending on previously experienced communication delays at the venue). In parallel, each robots determines the team's *captain* for the upcoming kickoff, i. e. the robot that has the lowest identification number and is currently on the field. After the communication phase, which usually lasts significantly less than one second, all robots use the variant K_r that has been communicated by the captain.

This approach turned out to be robust and efficient. In addition, it automatically scales with the actual number of robots on the field. However, the case of single robots that are on the field but temporarily not able to communicate is not handled. Such a situation might, in the best and fortunately often occurring case, still result in a proper execution of the kickoff but could also require a manual placement by the referee and an according suboptimal execution.

The possibility of easily adapting each kickoff variant's weighting, in order to prefer some variants over others, has been used before several matches, depending on the walking speed and expected kickoff strategies of the opponent team.

6.3.7.2 Positioning

After the team has agreed on the variant for the upcoming kickoff. The according poses P have to be assigned to the robots. We do not rely on a fixed pose assignment (except for the goalkeeper, of course) based on robot numbers as such an approach has two significant drawbacks. Firstly, an important position might be left empty, if one robot is missing. Secondly, some robots might be forced to walk farther than necessary and thus risk to come too late.

As our self-localization provides a very precise pose estimate and all robots receive the poses of

their teammates via team communication, a robust dynamic pose assignment is possible. Our approach is realized by two sequential assignment rules:

1. The robot that is closest to the center position, i. e. the position in P that is closest to $(0, 0)$ according to the global coordinate system as described in Sect. 4.1.1, will walk to that position.

2. The two remaining robots are assigned to the remaining poses according to the y-component (cf. Sect. 4.1.1) of their current pose, i. e. the leftmost robot will walk to the leftmost kickoff pose.

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. During RoboCup 2011, B-Human always played with one defending robot (cf. Sect. 6.3.9.3). As this robot always stayed in the own half of the field, it was in most cases the one which was selected for the central position, having only a short and often obstacle-free way to walk.

The second rule is not only a simple assignment heuristic but also leads to a faster positioning. Using the robots' y-positions (instead of, for instance, the absolute distance to the kickoff positions) ensures that the paths of both robots will not cross. In addition, the number of crossings of the central part of the field, i. e. the area between both goals, is reduced. Thereby, collisions and obstacle avoidance maneuvers occur less often and it is easier to reach all kickoff positions in time.

6.3.7.3 Actions

When the game state changes from *SET* to *PLAYING*, the according kickoff action A is carried out. The specification does not contain sub-actions for individual robots but only one global action, such as *pass*. Each robot can derive its individual behavior within the action from its current position. The actually executed behaviors are implemented as XABSL options.

For RoboCup 2011, the following actions have been implemented for the opponent team's kickoff:

play All robots immediately switch to their playing behavior. No kickoff behaviors are executed.

wait_single One of the two robots that do not occupy the center position waits for several seconds (configured in A_p) whilst all others immediately switch to their playing behavior. In case of an opponent kickoff that transports the ball quickly into our own half, this waiting behavior ensures that at least one field player faces the ball and is able to react in time.

wait_double This action is a more defensive variant of *wait_single*, leaving both non-centered robots at their positions for some time.

In theory, all of these actions could also be used for the own team's kickoff. But as they do not include any ball handling behaviors, the following actions have been implemented:

dribble The robot that carries out the kickoff dribbles the ball towards a position configured in A_p. All other robots switch to their playing behavior. This action turned out to be very successful and was used in most games.

kick The robot that carries out the kickoff kicks the ball to a position configured in A_p. A reasonable precision is achieved by using the kicking engine presented in [24]. All other robots switch to their playing behavior.

pass Similar to the kick *action*, the robot that carries out the kickoff kicks the ball to a position configured in A_p. During the kick and the preceding alignment of the kicking robot, one of the other robots walks to a position close to the kicking target to receive the pass. The remaining robot switches to its playing behavior. If there is no robot available for walking to the pass receiver position, the kicking robot switches to a default *dribble* action.

All 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 ten seconds) has been reached.

6.3.8 Roles and Tactics

The option *choose_tactic* invokes the options *attacking_keeper*, *check_catch_possibility*, and *role_assignment* in parallel. The first two options are used to gather some information about the current game situation and the last option assigns the dynamic roles according to this information and the selected tactic. It is also possible to use only the native roles given by the robot numbers for testing purposes. In contrast to last year's behavior, the ready state is independent from roles and tactics.

The decision whether the keeper behaves as a normal field player or not takes place in *attacking_keeper*. At the moment, when the keeper (per definition the robot with the number 1) is the only player on the field, it becomes striker. When there are other players in game again, it returns to its native role after it loses the striker's role (decision according to dynamic role changes as described in Sect. 6.3.8.2). While the keeper is in the role of the striker, the support parts are assigned as if the keeper is absent. Therefore, there is always a defender placed in front of the own goal in the game.

The option *check_catch_possibility* is used to get the information if a field player is able to intercept a ball rolling towards him. If a player, except the keeper, is able to do this, it should become a striker immediately. This takes place independently of the computation of the best time to reach the ball. In the role of the striker, the option to intercept the ball is invoked. If the action succeeds and the ball is located in front of the player, then it is likely that the robot will stay striker after this action (decision according to dynamic role changes as described in Sect. 6.3.8.2). Otherwise, it will switch into any support role again, according to the selected tactic.

The option *role_assignment* selects the striker according to the current game situation and invokes the option *choose_support_role* which assigns the support roles depending on the current game situation and the chosen tactic. Moreover, the robot with the number 1 becomes keeper if it is on the field and it should not attack based on the decision made in *attacking_keeper*. The different tactics and the role selection will be described in detail in the following sections.

6.3.8.1 Tactics

There are three tactics – namely *offensive*, *normal*, and *defensive* – that are described in detail in this section (cf. Fig. 6.9). In contrast to last year's behavior, the tactic choice is configurable and depends no longer on the current goal ratio of the game state.

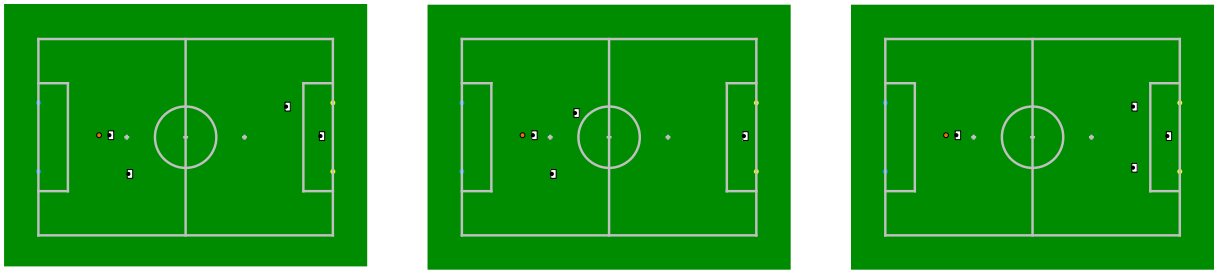


Figure 6.9: Typical situations for each of the tactics (attacking the blue goal on the left). From left to right: normal, offensive, and defensive tactics.

By default, B-Human played with the *normal tactic* at RoboCup 2011. Thus, each match was played with a keeper, a defender, a supporter, and a striker. At this tactic, the striker always walks towards the ball if possible (cf. Sect. 6.3.9.1), the supporter always walks next to the striker with a defined offset, the defender stands in front of the own penalty area placed sidewards from the keeper, and the keeper stays inside its penalty area.

The *offensive tactic* also contains a striker, a supporter and a keeper, but instead of a defender, the other available robot becomes a second supporter and walks next to the other side of the striker, so that there are one striker and two supporters. The supporter that has the smaller x position also has a backward offset to the striker that is a bit higher than the one of the supporter next to the striker. By this arrangement, a triangle is formed. The disadvantage of this tactic is that it might become very crowded close to the ball and the robots interfere each other.

The *defensive tactic* contains two defenders instead of the one or two supporters in the other tactics, a keeper, and a striker. The defenders stand in front of the own penalty area, one on the right side of the keeper and the other on the left side. The disadvantage of this tactic is that if the ball is in the opponent half the distance between the robots will be too high and the possibility of team play decreases.

6.3.8.2 Role Selection

The dynamic role selection consists mainly of the options *role_assignment* and *choose_support_role*. In general, the role selection is dependent on the number of field players, their current roles, the selected tactic and the current game situation. The role selection will be described for the normal tactic as it has been the default one. The main differences in the role selection for the other tactics will be mentioned where required. The dynamic role selection only takes place during the game state *playing*.

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

striker. To prevent an oscillation of the striker and any other support role, the synchronization mechanism of XABSL (cf. Sect. 6.1) is used. The roles of the striker and the keeper (according to the decision made in *attacking_keeper*) are assigned in the option *role_assignment*. All other support roles (defender and supporter) are assigned in *choose_support_role* according to the selected tactic and the state of the keeper (if it is on the field and if it is attacking or not). The selection between the different support roles depends on the x-coordinate of the robots on the field. The robot with the less x position takes the more defensive role. In the defensive tactic, the decision whether the defender plays left or right depends on the y-position of the robot.

Of course, a robot that is penalized must be excluded from the role selection. For the remaining robots, the role selection mainly depends on the number of the robots that are currently on the field, which can be any number between zero and four. This variation can occur because of broken or penalized robots. If no robot is on the field, no role selection needs to be done.

For the case that only one robot is on the field, this robot always gets the striker's role, independent from the native role and the game situation, because the striker is the only role a robot can take to score goals whereas the other roles are only sensible in conjunction with the striker.

If two robots are on the field, a more defensive tactic is chosen in general. One of the players takes a defensive role (defender or keeper) whereas the other player becomes striker. The decision about the defensive role depends on whether the robot with player number 1 (per definition the keeper) is on the field and the decision made in *attacking_keeper*. If this robot is available and currently not attacking, this robot becomes keeper, and the other robot becomes striker. Otherwise the roles defender and striker are assigned.

For the case that three robots are on the field the roles keeper or defender, supporter, and striker are assigned. The decision about the defensive role depends on whether the robot with player number 1 (per definition the keeper) is on the field and the decision made in *attacking_keeper*. If this robot is available and currently not attacking, this robot becomes keeper.

The last possible case is that all four robots are on the field, which get the roles striker, supporter, defender, and keeper in a normal tactic. Naturally, the robot with player number 1 is the keeper, whereas the defender's, supporter's, and striker's role are assigned to the other robots. The following differences between the various tactics can be concluded: In the offensive tactic there are two supporters and no defender assigned and in the defensive tactic there are two supporters and no defender assigned. The role selection takes place, as mentioned above, for every case. For the special case that the keeper is attacking and four robots are on the field, one robot takes the role of a defender, two robots take the role of a supporter, and the robot with the number 1 stays striker until its role switches to any support role.

6.3.9 Different Roles

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

6.3.9.1 Striker

The main task of the striker is to go to the ball and to kick it into the opponent goal. To achieve this simple-sounding target, several situations need to be taken into account:

go_to_ball_kick. Whenever no special situation is active (such as kick-off, duel, search for ball, or behavior near the goal), 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 *go_to_ball_kick*.

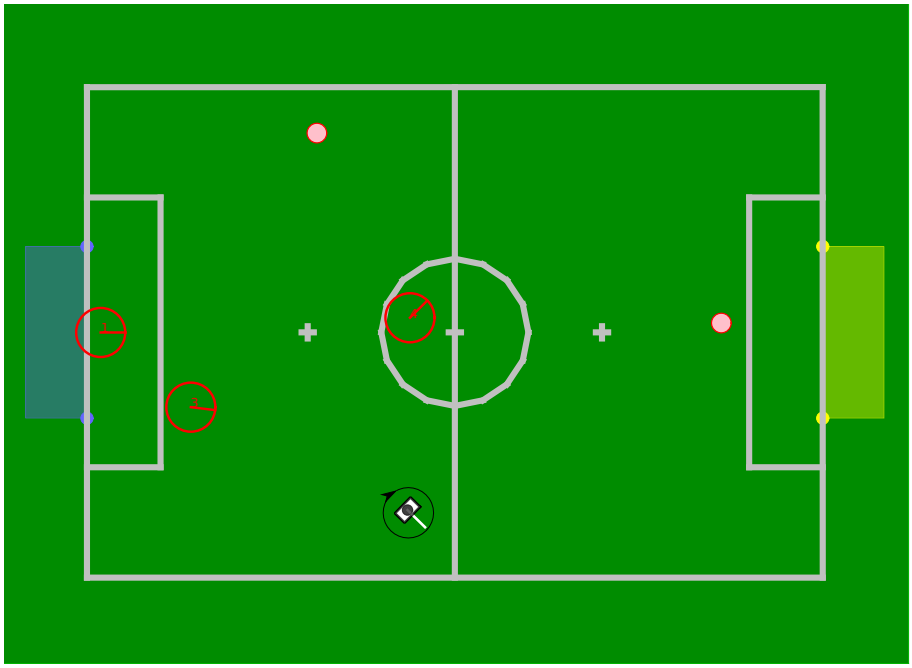


Figure 6.10: Visualization of the search-for-ball procedure. A robot first spins around and then walks to its patrol target (pink circles). The patrol targets change depending on the number of searching robots and on their positions (red circles).

To reach a good position near the ball, the *KickPose* (cf. Sect. 6.3.4) is used and the striker walks to the provided pose. The coordinates are directly passed to the walking engine (cf. Sect. 5.2.1) as a target. To reach the ball in an acceptable time, the striker first turns and then walks straight towards the ball when the distance to the ball is quite high. In case that the striker's position is between ball and goal or on a level with the ball and the pose is behind the ball, it has to walk around the ball until the pose is reached. Whether the robot turns left or right around the ball depends on the angle to the ball and to the goal.

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

Which kick will be executed is selected by the KickPoseProvider (cf. Sect. 6.3.4). The following kicks or kick directions were used at RoboCup 2011: forward (with variable strength), backward, sideways, and diagonally. After reaching the *KickPose*, the striker executes the chosen kick immediately if no other state becomes active before.

To avoid obstacles on the way to the target, there are two strategies available. For great distances, the path planner (cf. Sect. 6.3.6) is used and for small distances to the target the striker walks sidewards around the obstacle based on the ultrasonic measurement (cf. Sect. 4.2.5). The latter was the only obstacle avoidance method in the last year's behavior.

search_for_ball. While turning, the robot aligns its head to the positions provided by the FieldCoverageProvider (cf. Sect. 4.2.9.1). If, after a whole turn, no ball was observed, the robot patrols to a position provided by the GlobalFieldCoverageProvider (cf. Sect. 4.2.9.2). Figure 6.10 visualizes this procedure.

duel. Whenever there is a close obstacle between the ball and the goal, the robot hurls itself

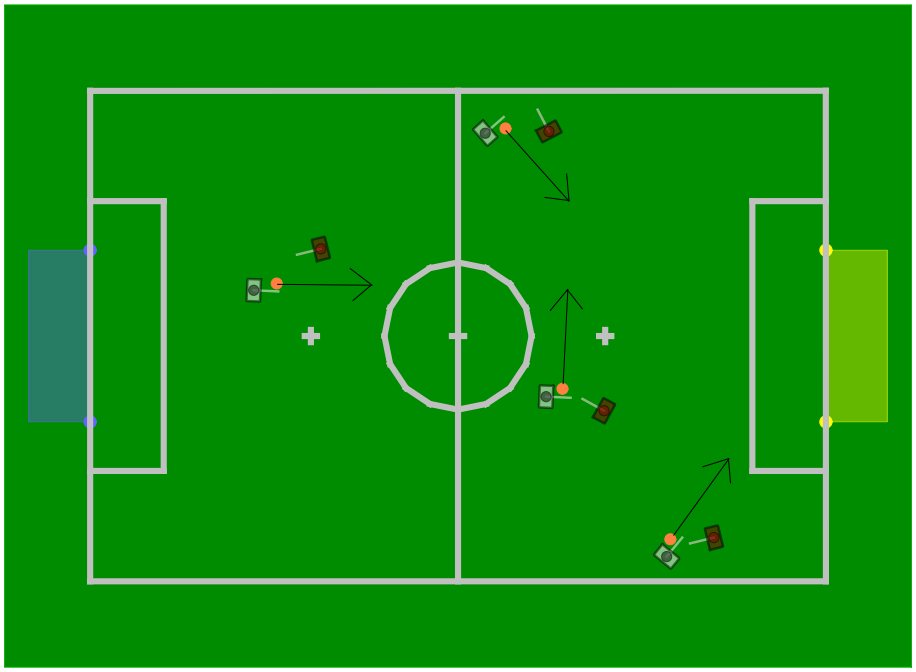


Figure 6.11: A set of possible tackling situations. The black lines mark the desired kick direction.

into a tackle. Depending on the position of the robot and the obstacle, there are several actions to perform (cf. Fig. 6.11):

- Robot is near the left or right sideline.
  - If the robot is orientated 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 orientated away from the opponent goal, the robot will perform a sidewards kick to the middle of the field.
- 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 depending on the angle to the obstacle. The kick is always directed away from the obstacle, so that the robot can follow the ball with less risk to run into the opponent and – of course – we do not want to offer gifts.
  - If the obstacle leaves room for a direct kick towards the opponent's goal, the robot will perform a fast forward kick. The kick leg is chosen by the smallest distance from the leg to the ball.

If the execution time of a duel action takes more than three seconds, the robot will move backwards for a short time to reduce the probability of getting punished for pushing. The fast kicks are implemented within the walking engine (cf. Sect. 5.2.1). Tackles are deactivated in the vicinity of the opponent's penalty box, because goalposts might be deemed as obstacles, so the striker might never score.

pass_before_goal. If the ball is lying near the opponent ground line, so that it is unlikely to score a goal, the striker has to decide whether to pass or to dribble.

- The striker will perform a pass to the front of the middle of the penalty box if there is a supporter ready to receive the ball.

- The striker starts to dribble towards the front of the goal if there are no supportive players on the field or if the receiving supporter is too far away.

The passing itself is performed with BIKE, so there is no additional module needed. The normal kick was edited in a way that the kicking foot is not dynamically justified to the ball position. Additionally, the execution time of the kicking phase and the strike out range of the foot is dynamically applied, according to the distance of the pass target. To have some kind of mapping between distance and execution time, it is necessary to perform some measurements. We measured the distance the ball moved after getting kicked with different execution times: 300 ms, 350 ms, 400 ms, 500 ms, 600 ms, and 700 ms. After this, we used linear regression to approximate a function that fits to these value pairs. The formula used at the competitions of 2011 was

$$f(x) = 1.6498e^{-5}x^2 - 0.194 * x + 863.424 \tag{6.1}$$

where x is the distance to the pass target. These calculations are performed within some *XABSL* BH2011BehaviorControl symbols.

As mentioned before, even if there is a supporter on the field, a pass will only occur if the supporter is less than 3 m away from the ball and if it is standing in between a certain range of angles, as shown in Figure 6.12. This will reduce the danger that the robot passes too far towards the own goal.

On entering the *pass_before_goal* state, the striker will inform the nearest supporter. While the striker is moving towards the ball, the supporter moves in front of the opponent's penalty box. If the striker has aligned behind the ball towards the pass target, it informs the supporter again, so that the supporter will stop moving and starts looking at the ball. The pass is only initialized if the receiving robot is in the range of the legal receiving positions (cf. Fig. 6.12). After the pass was performed, the ball should have moved close enough to the supporter, so that a role change will occur. If the passing scheme is aborted somehow, the receiving supporter will be informed.

dribble_before_goal. As already described in *pass_before_goal*, this state is a fallback if it is not possible to perform a pass towards a teammate. Instead the striker aligns behind the ball and moves forward slowly. If the robot drifts too far away from its target (shortly before the opponents penalty box) it starts again to align. If it is possible again to score directly the state *go_to_ball_kick* is executed.

6.3.9.2 Supporter

The main task of the supporter is, as the name indicates, to support the striker. In contrast to last year's behavior, there is only one specified role for the supporter, which is used in each tactic as mentioned in Sect. 6.3.8.1.

Most of the time, when the game state is *PLAYING*, the supporter tries to get a good position to support the striker. This is done by positioning the robot next to the striker with a defined offset. It is important that the supporter is able to see the ball from time to time. Therefore, it is aligned to an angle between the ball and the opponent goal or just to the ball if the supporter is in front of the ball. When the offensive tactic is used and all robots are on the field there are two supporters in the game. In this situation, the striker and the two supporters build a triangle, as described in Sect. 6.3.8.1.

Besides positioning, several special operations exist. The *search for ball* behavior mentioned above (cf. Sect. 6.3.9.1) is used for the supporter as well as the obstacle avoidance, and a special

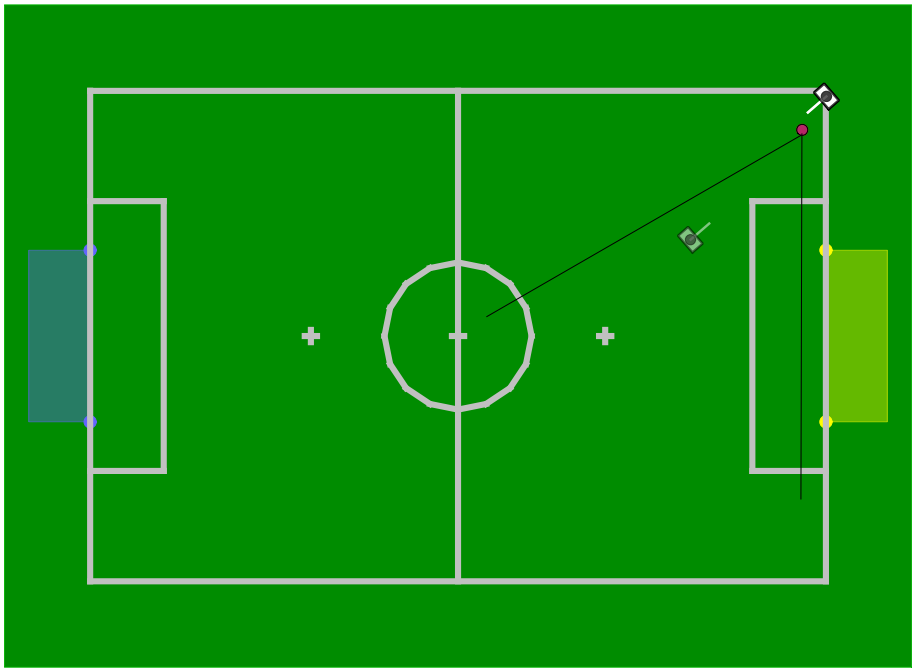


Figure 6.12: Passing targets are only valid if they are between the both black lines.

positioning, when the keeper decides to walk towards the ball. In the latter case, the supporter positions on the middle line facing towards the center circle where the field side depends on the position of the ball seen by the teammates (cf. Fig. 6.13). Another special situation is the case that the ball is lying next to the opponent goal and the striker wants to execute a pass. In this case, the supporter leaves its regular position and walks to a position in front of the goal to receive the pass. When the game state is *READY*, no support role exists and each robot behaves as described in Sect. 6.3.7.

In any other case, the supporter positions next to the striker, where the side is kept until the target position comes too close to the field border. This way, most of the field is covered and an oscillation of positions is prevented. The supporter is positioned approximately 1.1 m to the side and 0.3 m to the back relative to the striker.

The positioning of the supporter is realized by a potential field [11] as depicted in Fig. 6.14. The decision to use a potential field instead of just letting the robot walk to the desired position was taken because the supporter is supposed not to disturb the striker. The potential field is a combination of several basic potential fields that are described below. The output of the potential field is the desired walking direction of the robot, where the supporter should always be turned towards the ball, except when the target position is far away.

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

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

Rejection of the other supporter. The two supporters should cover as much field as possible

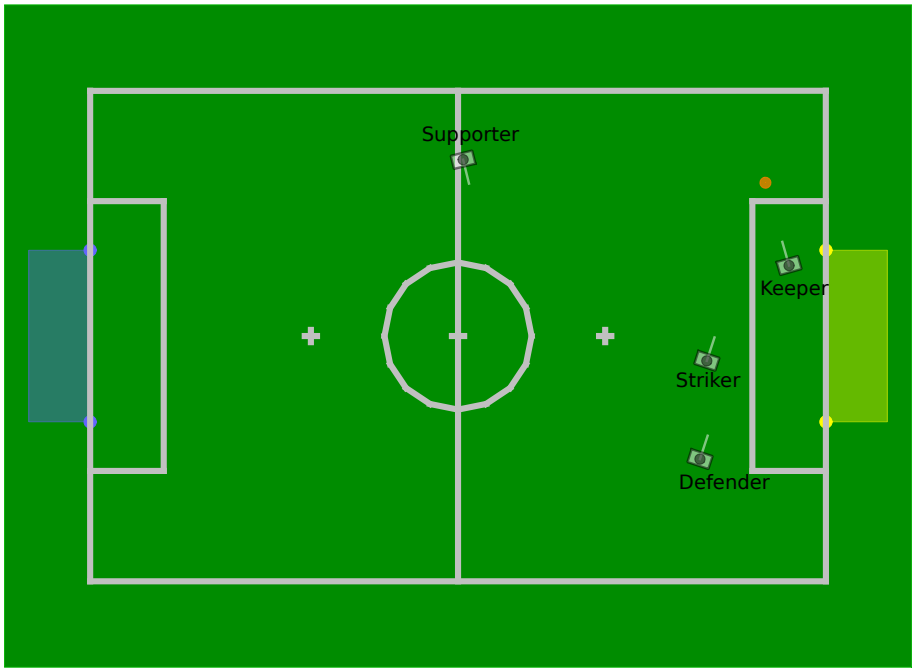


Figure 6.13: Positioning of the field players when the keeper decides to kick the ball away. The striker stays near the ball in front of the penalty area. The defender walks to one of the penalty area corners. Last but not least, the supporter waits for the ball at the middle line. Note that these positions display the desired target positions using the *normal* tactic.

which lays behind the striker. This means that if both supporters are near to each other, there is a huge part of the field free for other players. This should be avoided.

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

Rejection of the ball-goal-line. The supporter is rejected from the ball-goal-line. This ensures that the striker does not accidentally hit the supporter when trying to score a goal.

Attraction towards the back of the striker. The combination of the previous two potential fields can lead to a local minimum in front of the striker, which is a bad position for the supporter. To prevent this situation an additional potential field is introduced that moves the supporter behind the striker.

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

6.3.9.3 Defender

The defender's task is, in general, to defend the own goal without entering the own penalty area. The defender appears if the keeper is out of the field or attacking, or if the defensive or normal tactic is chosen. Similar to the behavior of the keeper, the defender tries to position between the

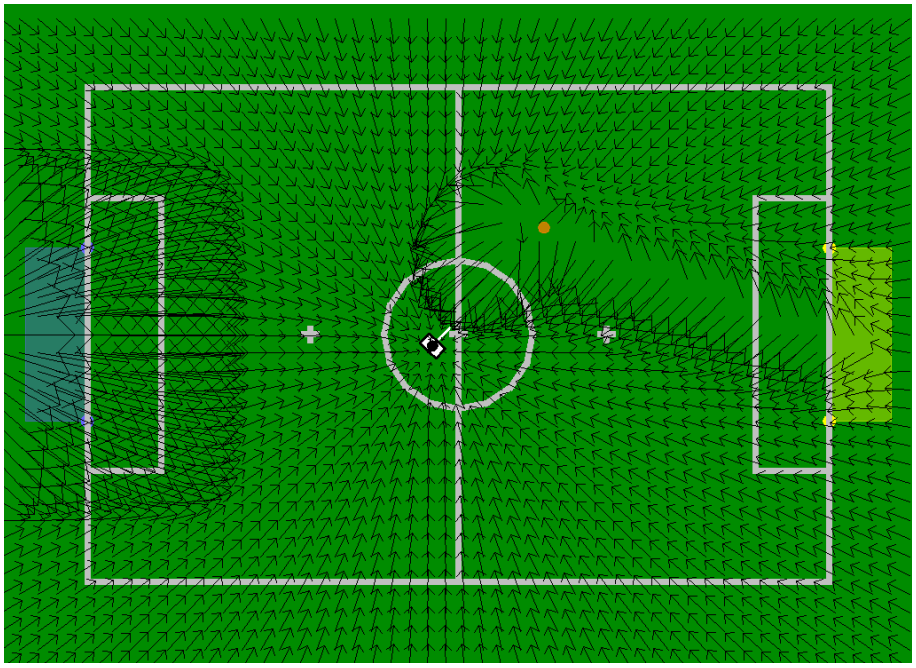


Figure 6.14: Visualization of the potential field for positioning the supporter. The components that are combined can be clearly seen: the rejection of the own penalty area, the attraction of the target position, the circular rejection of the striker with a bias to the own goal, and the rejection of the line between ball and opponent goal.

ball and the own goal. It has a fixed x-position on the field and only moves to the left and right, i. e. it only moves on an imaginary line located in front of the own penalty area (cf. Fig. 6.15). The positioning of the robot is realized with the potential field that is also used for positioning the supporter (cf. Sect. 6.3.9.2). Also the obstacle avoidance is completely the same as for the striker.

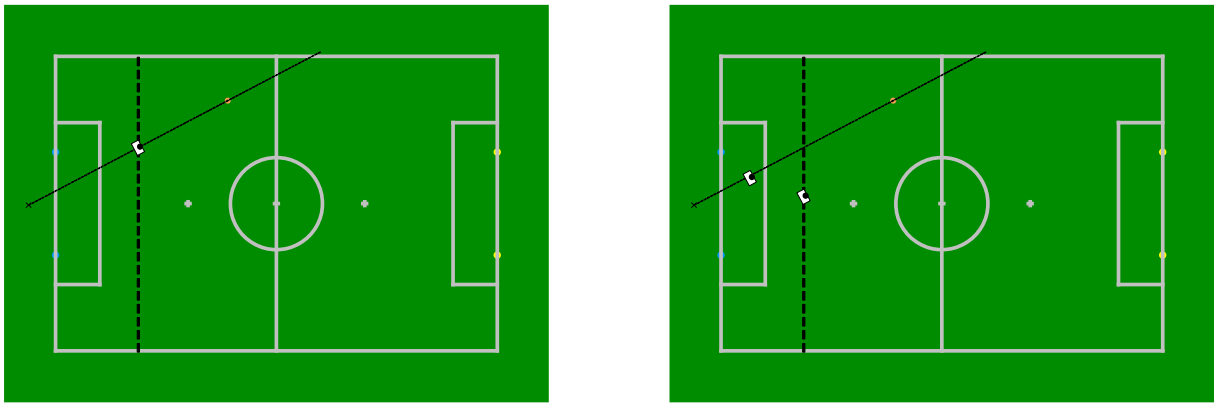


Figure 6.15: The positioning of the defender. The first picture shows the centered position if no keeper is on the field, and the second depicts the defender placed sidewards to the keeper.

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

in a good position to kick the ball.

If the defensive tactic is chosen, all four robots are on the field, and the keeper stays in its goal, there are two defenders on the field. One stays on the left side of the keeper and the other one stays on the right side.

6.3.9.4 Keeper

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

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

In any other situation, the robot will not leave its position within the own penalty area, and it executes different actions according to different situations. Normally, the keeper walks to a position computed by the GoaliePoseProvider (cf. Sect. 6.3.5) module. After reaching the desired pose, the goalie either stands or crouches depending on the distance of the ball to its own goal. If the ball is in the opponents half of the field, the keeper stands. In the other case, if the ball is in the own half, it crouches. Additionally, to avoid oscillation between these two decisions, there is an hysteresis applied. The idea behind this behavior is that on the one hand, the keeper should be prepared to jump if the ball is close to the own goal. On the other hand, the robot should react as fast as possible on fast breaks.

If the ball is rolling towards the goal, the goalie has to select from a given set of actions: staying crouched, spreading the legs, or diving. The decision is made by considering the velocity and the distance of the ball. The estimates are used to calculate both the remaining time until the ball will intersect the lateral axis of the goalie and the position of intersection. If the ball rolls straight towards the goalkeeper, or it is expected to clearly miss the goal, the keeper remains crouched. In case of a close intersection, the goalie changes to a wide defensive posture to increase its range (for approximately four seconds, conforming to the rules of 2011). 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 execution of a cover motion is delayed by a third of a second to compensate false positive ball estimates.

Sometimes, after getting up after a diving motion, the self localization is incorrect, more precisely the robot's rotation is wrong by 180°. To avoid situations where the keeper has an erroneous rotation and looks into the own goal, we provide a boolean *XABSL* symbol that gives information whether the goal, the ball and the center circle percepts stayed away for thirty seconds. If so, the goalie starts to rotate by 180° and looks up and down in hope to get some perceptions and to consequently improve the self localization. This procedure is handled in the superior option of the normal goalie behavior.

The introduction of a common ball model (cf. Sect. 4.2.10.1) opened new possibilities for the goalie's behavior. In 2010, there were several problems when the ball disappeared from the goalie's field of view (for example when the ball was covered by field players) because it immediately started to use *active vision*. This sometimes lead to situations in which the ball was moving towards the own goal and the keeper was looking at anything but the ball. With a global ball estimate, we are able to avoid such situations. If the ball was not seen for seven seconds, the goalie has to decide what to do next. Either the keeper's teammates also have lost the ball, which is implied by a not valid global ball model or the ball disappeared just for the keeper but the global ball model is still valid. In the first case, the goalie walks straight to the middle of the goal and starts searching for the ball on arrival. The search procedure itself differs from the field players' procedure. Depending on the last seen position of the ball, the keeper rotates by 45° to the left or right and scans the whole area in front of it. After one scan, the keeper rotates to the other side and starts another scan. This procedure is repeated until either the ball was seen or the global ball model becomes valid again. In the second case, the goalie looks at the global ball model instead of its own estimate and should stay where it is, which increases the probability to find the ball faster.

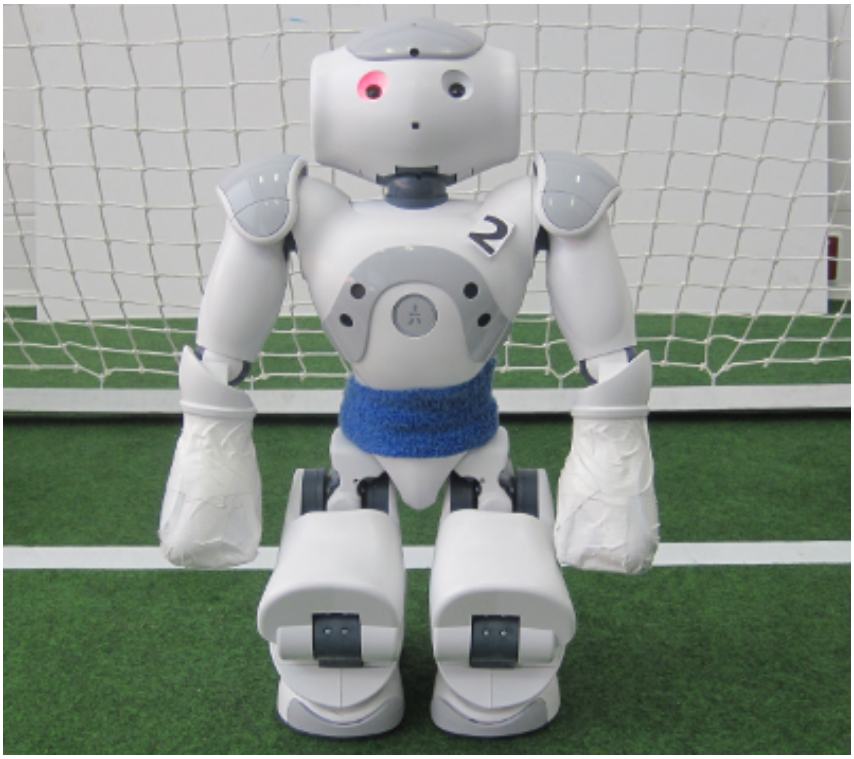


Figure 6.16: Crouching position of the keeper.

6.3.10 Penalty Control

Penalty control is responsible to stop the robot's movement while being penalized. It is the last option to be called within *start_soccer*. Therefore, it can overwrite the movement requests made by the body and head control options. It is implemented as a separate option (and not separately in body control and head control) to have a single place easily accessible to implement the penalty behavior. Basically, when the robot is not penalized, this option does nothing. When the robot is penalized, it sets the motion request to *stand* and lets the robot look down. In the first two seconds after the robot is unpenalized, it keeps control over the robot's movement and

lets it walk straight forward to get back onto the field. More importantly, the robot looks at a predefined angle of 45° to its left. This is done to help localizing the robot, as the particles in the SelfLocator are evenly distributed at the two possible reentry points on the field, and if the goal color on one side can be determined, the robot knows exactly on which side of the field it was put back into the game. After that, the robot's head is moved from left to right to quickly check whether the ball can be seen in front of it.

6.3.11 Display Control

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

6.3.11.1 Right Eye

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

6.3.11.2 Left Eye

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

6.3.11.3 Torso (Chest Button)

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

6.3.11.4 Feet

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

6.3.11.5 Ears

- The right ear shows the battery level of the robot. For each 10% of battery loss, an LED is switched off. Additionally, the LEDs that are still on start blinking when the distance to a detected obstacle is less than 50 cm.
- The left ear shows the number of players connected by the wireless. For each connected player, one third of the led circle is switched on. Additionally, the left ear gives information whether there is a GameController operating or not. If there is a GameController running, the behavior of the left ear should display the number of connected players as described before. If there are connection problems or if there is no GameController running at all, every second led displaying the number of connected players is switched off. The remaining activated LEDs are shifted every 500 ms, so you get the impression that the LEDs are rotating.

6.4 Match Statistics

The tuning and debugging of behaviors is a rather difficult task if there are just handwritten notes on observed problems, which are often subjective and incomplete. Due to the same reasons, comparing two behaviors is also a tricky business. Therefore, gathering concrete data on a match is a valuable enhancement. The MatchStatisticProvider was designed to provide helpful information to have a basis for behavior comparison rather than to provide detailed debugging information (for this purpose, the logging mechanism was implemented (cf. Sect. 8.7)).

In each half of a game, the following data is collected by each robot:

- fixed size data
  - kickoff timestamp
  - final whistle timestamp
  - a heat map, which is a grid over the field, representing the robot's positional distribution
  - time spent having a certain role (supporter, striker, keeper and defender)
  - longest time during which no ball was seen
  - time went by without receiving GameController packages
  - longest time spent on searching the ball
  - mean time to shoot a goal, per robot and team
  - time passed without ground contact
  - the total time spent on duelling
  - time being penalized
  - kick count
  - overall covered distance
- variable size data
  - each pose where a kick was performed, including:
    - timestamp of this kick
    - the executed kick type

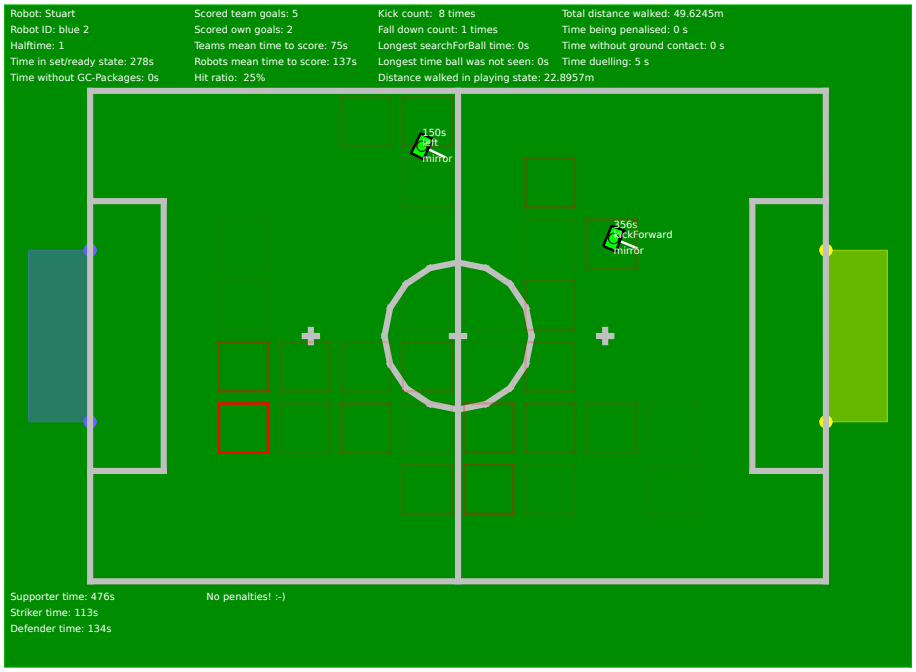


Figure 6.17: The red grid represents a robot's positional distribution. From the green robot poses, goals were scored.

    - result of this kick: ball went out, hit the goal, or no effect
- each position where the robot fell over, including:
    - a flag that gives information whether this fall was caused by a kick or not
    - the kick that causes a fall
    - the executed stand up motion identifier
- each pose where the robot started to search for the ball, including:
    - the time spent on searching the ball
    - a flag that gives information whether this search was successful
    - a flag that gives information whether this robot was patrolling
    - the position of the recovered ball
    - the final patrol pose when this search stopped
- each pose where the robot was penalized, including:
    - the time stamp of this penalty
    - the penalty applied
- each path, this robot has taken being in playing state, including:
    - the active role on path parts

Data is collected if the robot switches from initial to playing state. On entering the finished state, the data gathering is interrupted. All collected information can be stored by either sending a *DEBUG_REQUEST* via the *SimRobot* console or via the *bush* (cf. Sect. 2.7). Figure 6.17 and Figure 6.18 display some typical visualizations of a *MatchStatistic* (*RoboCup2011*, *B-Human* vs. *Dutch Nao Team*, 1st half).

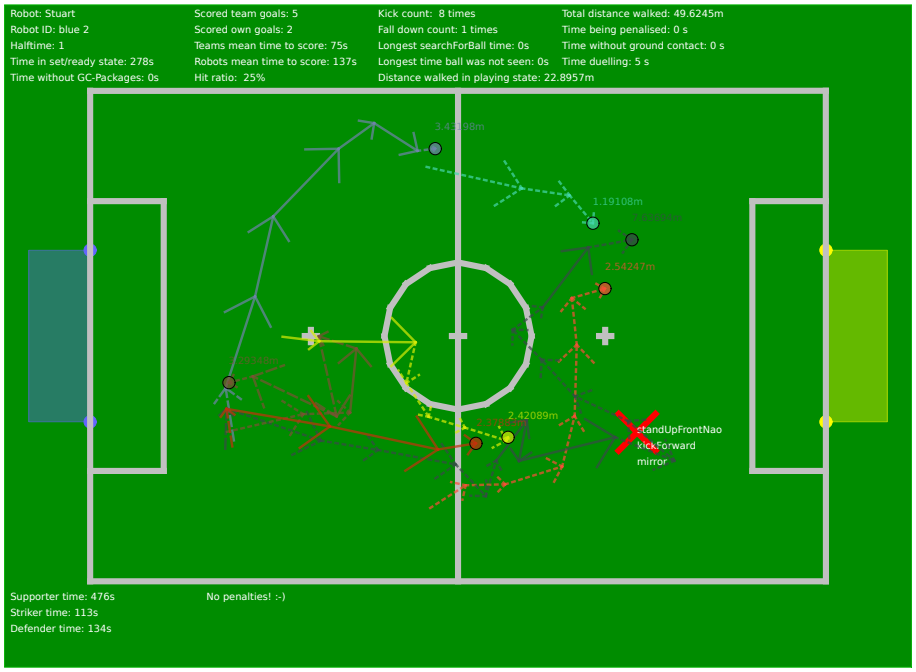


Figure 6.18: Arrows of the same color represent a path, the robot went in playing state. The red cross marks a position where the robot fell over because of an unstable kick.

6.5 b-script

b-script is an experimental behavior specification language. It utilizes a specialized generator implementation called *tasks* to describe robot behaviors hierarchically in a procedural and scripting-like fashion. The general approach is similar to XABSL, but instead of using finite state machines to describe the components in the hierarchy, each component is described by a *task* (or *function*). As in many scripting languages, each file declares a module and each module declares *tasks* and *functions*. Dependencies between modules can be declared using the *required* statement at the beginning of each module. In contrast to most other scripting languages, *b-script* uses a strong type system to ensure run time safety.

Further information is available at `www.informatik.uni-bremen.de/~jeffry/b-script`.

6.5.1 Tasks

Tasks are a specialized generator implementation which provides a function-like handling of generators. The differences of *tasks* compared to usual generators (as known for example from Python) are the following:

- each *task* is associated with a single context which is stored in the *b-script* runtime environment and will be retrieved automatically when the *task* gets called. As a result, *tasks* can be called just like functions
- each *task* (context) has a state, it can be set explicitly using `yield <stateValue>`, besides the state will be set automatically to `running` if the *task* yielded without explicitly setting the state or to `done` if the *task* terminated. The state of a *task* can be retrieved with `taskState(<taskIdentifier>)` and be used for further control decisions
- the context of a *task* is reset automatically at a call if the *task* was in `failure` or `done` state

or was not called at all during the last execution cycle, otherwise the *task* will continue its actual context

- the parameters of a *task* are evaluated at every call

The grammar to declare a *task* is as follows:

```
task <identifier>([<variableDeclaration> [{, <variableDeclaration>}]])
  <statementBlock>
```

Thus a simple *task* might look like this:

```
task helloWorldTask()
  io::prints("Hello")
  yield

  io::prints("World")
  #implicit yield TaskState::done
```

Since the context of a *task* preserves the local variables and the program counter over consecutive calls, these can be used to describe functions like state-based routines. Thus, by using the control flow, we can flexibly design custom state-based automata. Fig. 6.19 depicts how to describe behavior components utilizing *tasks*.

The basic principle to use these *tasks* to describe hierarchical robot behaviors is, that the *tasks* will be called like functions and will execute one step, based on the actual state of the *task*, and suspend again afterwards. Thus when a *task* executed a step it will 'return' to its caller which will continue its execution until it is done executing one step and suspends again. When a *task* terminates, i.e. reaches the end of its statement block, the *task* will yield automatically with the task state `done`.

In contrast to *functions*, *tasks* do not support return values.

6.5.2 Functions

Besides *tasks*, *b-script* supports regular functions. The grammar to declare a *function* is as follows:

```
func <identifier>([<variableDeclaration> [{, <variableDeclaration>}]]) : <
    returnType>
  <statementBlock>
```

Thus a simple *function* might look like this:

```
func addOrSub(int x, int y) : int
  if(x + y % 2 == 0)
    return x + y
  else
    return x - y
```

Functions might be used to outsource certain calculations (e.g. `getPositionBehindBall()`), but might also be used to describe stateless components in the hierarchy (e.g. the `walk(x, y, theta)` sub-behavior).

6.5.3 Statements and Expressions

To implement the bodies of *tasks* as well as *functions*, *b-script* supports basic procedural programming statements and expressions. The supported binary expression are $+, -, *, /, \%, and, or, <, >, <=, >=, ==, !=$ and the supported unary expression are ! and $-$.

a)

```
task gotoKickPoseAndKick()
  func readyToKick() : bool
    return #determine kick pose reached

  for(; !readyToKick(); yield)
    Motion::walkToPose(input.kickPose.pose, 100.0, true)
    if(input.kickPose.pose.translation.abs() < 250)
      HeadControl::lookAtBallAndGoal()
    else
      HeadControl::lookActive(1.0)

  for(; true; yield)
    HeadControl::lookAtBall()
    Motion::executeKickPose()
    if(taskState(Motion::executeKickPose) == TaskState::done)
      break
```

b)

```
task lookAtBallAndGoal()

  int startTime = input.time

  for(; true; yield)
    float progress = ((input.time - startTime) % 1000) / 1000.0

    if(progress < 0.75)
      lookAtBall()
    else
      lookAtOppGoal()
```

c)

```
task executeWalkingEngineKick(int kickType)
  int executedKick = kickType
  #set output signal until motion agent reacts
  for(; input.motionInfo.walk.kickType != executedKick or \
        input.motionInfo.motion != BH::Motion::walk; yield)
    output.motion.walk.kickType = kickType
    executedKick = kickType

  #reset output signal and wait until kick is finished
  for(; input.motionInfo.walk.kickType == executedKick and \
        input.motionInfo.motion == BH::Motion::walk; yield)
    output.motion.walk.kickType = BH::WalkKickType::none
```

Figure 6.19: a) the *gotoKickPoseAndKick task* used for penalty behavior b) a low level *task* from the HeadControl module c) another low level *task* executing a certain walking engine kick from the Motion module

Besides those expression, the following statements are supported:

Statement	Example
variable declaration	`int myLocalIntVariable = 5`
assignment	`myLocalIntVariable = addOrSub(4, 2) + 1`
becomes assignment	`myLocalIntVariable += 73`
call statement	`addOrSub(2, 4)` or `helloWorldTask()`
return	`return myLocalIntVariable / 73`
yield	`yield TaskState::failure`
if-else-statement	see Sect. 6.5.3.1
for-statement	see Sect. 6.5.3.2
break and continue	

6.5.3.1 if-Statement

The if-statement has the same syntax as in C/C++ like languages:

```
if(<condition>)
  <statementBlock>
{else if(<condition>)
  <statementBlock>}
[else
  <statementBlock>]
```

Her is an example:

```
bool greatBool = Math::random() > 0.5
bool otherBool = Math::random() > 0.5

if(greatBool and otherBool)
  io::printsln("greatBool and otherBool are true")
else if(greatBool)
  io::printsln("greatBool is true")
else if(otherBool)
  io::printsln("otherBool is true")
else
  io::prints("neither greatBool")
  io::printsln("nor otherBool is true")
```

6.5.3.2 for-Statement

The for-statement is the known three-expression-for-statement from C/C++ like languages, containing a domain specific extensions called 'livelock-protection'. The grammar definitions for a for-statement is as follows:

```
for(<variableDeclaration>; <condition>; <statement>) [: <livelock-specifier(
    int)>]
  <statementBlock>
```

Thus a for-statement (inside a *task*) might look like this:

```
for(; input.ball.position.abs() > 200; ) : 2
  gotoBall()
  if(taskState(gotoBall) == TaskState::failure)
    continue
  yield
```

To control the behavior of a for-loop, the regular `break` and `continue` statements are available.

Since nested for-loops are often used to conduct the control flow of a *task* to design 'states', it introduces the risk of creating livelocks. Such a livelock would result in a dead agent. Therefore *b-script* introduces the so-called 'livelock-protection' for *tasks*. To realize it, each for-loop is allowed to execute only a specified maximum number of iterations per execution cycle. The number of allowed iterations before the 'livelock-protection' interrupts a for-loop is by default 1 but might be overwritten (as in the example with 2).

6.5.4 Environment – Input/Output

To access the behavior input and output values, *b-script* has two predefined global variables. The input values are accessible globally through the identifier *input*, containing all sensor and world model input variables. The output values are accessible globally through the identifier *output*, containing all available actuator values.

To avoid any communication through the *input* and *output* variables, the *input* is a read only variable and the *output* is a write only variable. Thus the only communication between *tasks* and *functions* are the parameters and task state return values respectively.

6.5.5 Compiling to C++

The *b-script* library contains the ability to generate C++ code from *b-script* modules. The generated code can be loaded by the *b-script* engine and executed very efficiently. The B-Human build system compiles the *b-script* behavior to C++ code and compiles and links the generated code to a shared library which will be loaded dynamically at runtime by the BScriptBehaviorEngine. The resulting library is *libb-scriptBehavior.so* (or *libb-scriptBehaviorSim.so* when working with SimRobot).

6.5.6 Debugging

b-script supports a few debugging facilities. The B-Human BScriptBehaviorEngine provides the following debug commands:

- *vfd worldState module:BScriptBehaviorEngine:callTree*
  draws the call tree in the field view
- *dr module:BScriptBehaviorEngine:reloadBehavior*
  reloads the behavior (either from *Config/Behavior/* or from the binary module library *Config/libb-scriptBehavior.so*) and reinitializes the *b-script*Engine
- *get module:BScriptBehaviorEngine:useModuleLibrary*
  set module:BScriptBehaviorEngine:useModuleLibrary {*true, false*}
  if set to true [default] it causes the engine to use the binary module library (*Config/libb-scriptBehavior[Sim].so*), otherwise the behavior in *Config/Behavior* will be executed using the *b-script* interpreter

6.6 Penalty Behavior Used at RoboCup 2011

The RoboCup 2011 penalty behavior was implemented using *b-script*. The BScriptBehaviorEngine implements a B-Human module which uses the *b-script* library to execute *b-script* behaviors.

The *b-script* behavior is located in *Config/Behavior/* and the main behavior module (which will be called from the BScriptBehaviorEngine) is *Config/Behavior/Behavior.bs*. To activate the penalty behavior (and the BScriptBehaviorEngine) the location needs to be configured to 'Penalty' using either the *settings.cfg* (cf. Sect. A) or *copyfiles* (cf. Sect. 2.4). Since only the *Striker* and the *Keeper* participate in a penalty shootout, the player number should be either 1 (Keeper) or 3 (Striker).

The basic structure of the behavior code is as follows:

- *HeadControls*: contains basic skills to control the head motion of the robot
- *Tools*:
  - *Motion.bs*: contains wrapper functions and tasks which provide high level interfaces to the output structure
  - *Common.bs*: contains some commonly used utility functions like field2Relative(Math::Vector2 p)
- *Skills*: contains high level skills such as KickBallTo or SearchForBall
- *Roles*: contains modules which implement the different roles (i.e. keeper, defender,...)
- *ButtonInterface.bs*: implements the official SPL button interface (cf. Sect. 6.3.1)
- *PenaltyControl.bs* and *GetUpControl.bs*: are two modules which are used at a very high level in the hierarchy to handle penalties and fallen robots
- *Playing.bs* and *Ready.bs*: implement the Ready and Playing state
- *Behavior.bs*: is the main behavior module. It contains a main function which will be called frame-wise from the B-Human BScriptBehaviorEngine from within the robot software.

Chapter 7

Open Challenge

The Standard Platform League environment strongly relies on a fixed color assignment to simplify object detection and localization. In recent years, some field elements, such as certain colored goal parts and all beacons, have been removed, providing a more "natural" appearance of the soccer pitch. In our opinion, the next reasonable step is to remove one goal color. The contribution of B-Human's Open Challenge demonstration was to show that playing soccer in such an environment is already possible given the league's current state of the art.

7.1 Demonstration

B-Human's Open Challenge demonstration was a short match that had a realistic course of play. That means that the team consisted of four robots that were able to correctly position in the READY state, scored (no own) goals while PLAYING and recovered from kidnapping after having been PENALIZED or after having fallen over. During the match, spectators decided which robots to penalize. The two non-unique goals have been the challenge field's original goals, both covered by yellow sheets. A simulated scene of this setup is shown in Fig. 7.1.

We refrained from playing four against four for a couple of organizational reasons. First and foremost, it would have been simply too hard to organize and configure eight fully functional robots. Furthermore, playing against the own software looks ugly in general and would have detracted from the main content of the demonstration.

7.2 Implementation

Removing one goal color facilitates color calibration but poses a state estimation problem as no unique elements that allow a direct unambiguous computation of the robot's pose are left on the field. For self-localization, B-Human uses a combination of a particle filter (cf. Sect. 4.2.1) and a Kalman filter (cf. Sect. 4.2.2). The former computes a global position estimate; the latter performs local tracking for refining the global estimate. Both components needed to be modified for the Open Challenge demonstration.

7.2.1 Position Tracking

Having a known initial position and no robot kidnappings occur, a robot is able to track its position during a match even on a field without unique cues. For this sub-task of self-localization,

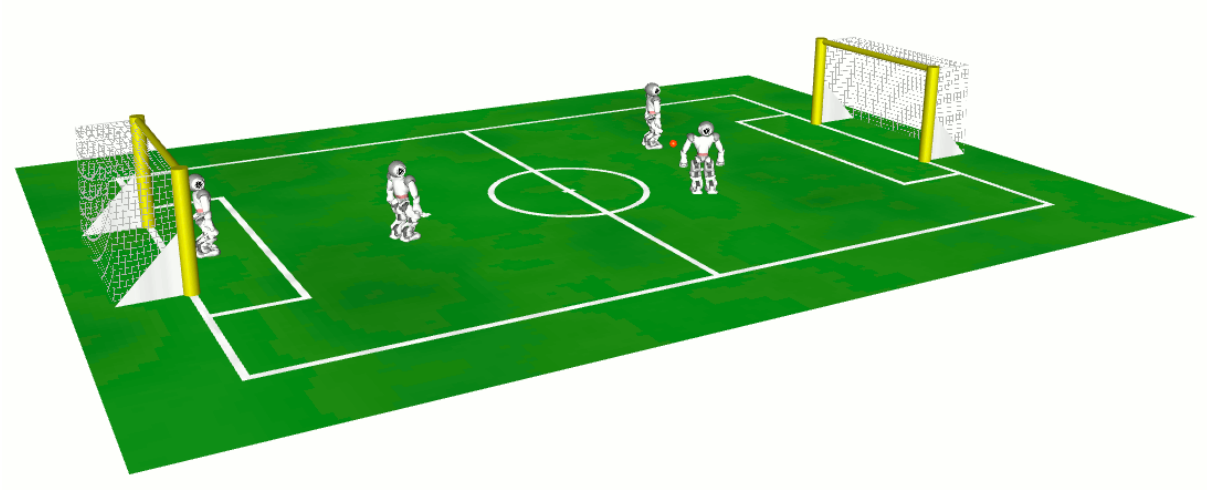


Figure 7.1: Setup of the Open Challenge demonstration. A team of four robots plays soccer on a field without unique goals.

the measurement models of the Kalman filter as well as of the particle filter have been adapted by adding two more yellow goal posts and removing the blue posts accordingly. In addition, the particle filter's sensor resetting mechanism – which should not be disabled even for tracking as otherwise, a few poor measurements might let the robot lose its track – needed to be adapted in similar manner. Every robot's initial pose is approximately known before the start of the game. If a robot falls over during the game, the resulting pose displacement is compensated by odometry (cf. Sect. 5.1.7).

However, keeping the particle filter's sensor resetting mechanism might generate samples that are at a totally wrong position due to the ambiguity of the goal posts that are used for computing the new samples. In the worst case, which occasionally occurs, these samples outstandingly match the perceptions made in the following execution cycles. Consequently, a majority of all samples will be replaced by samples at the resetting position. Fortunately, a Standard Platform League field with uniform goal colors is twofold point symmetric with reference to the field's center. This means that even if the sensor resetting causes a wrong position, this position is not arbitrary but symmetric to the previous position. By assuming that no teleportations to symmetric positions occur in reality, we can deal with this kind of symmetry in a straightforward manner by computing a mirrored robot pose whenever the sample set represents the symmetric counterpart.

7.2.2 Recovering from Kidnapping

However, the previously described position tracking is not capable to deal with a reoccurring event in the Standard Platform League scenario. A robot is penalized and has to reenter the field at one of the ends of the center line, depending on the current course of play. For self-localization, this situation is ambiguous and cannot be resolved by a state estimation process that relies on the fixed field elements only.

To resolve this ambiguity, we implemented a measurement model of the ball. As described in Sect. 4.2.10.1, all robots maintain a team-wide consistent ball model. A variant of this team model has been created for this challenge by excluding own ball observations and only fusing the observations of the teammates. This new model can now be set in relation to perceptions of the ball made from an assumed robot pose.

Currently, this model is used only in reentering situations after penalties (cf. Sect. 7.3). The robot keeps standing at its position until it has observed the ball for a previously configured number of times. The robot starts playing and the self-localization switches to the tracking mode again, if the observations made in this phase allow to reliably distinguish between the two possible positions. The robot restarts the observation phase, if the observations do not resolve the ambiguity. This happens when the ball is lying in or close to the center of the field.

7.3 Discussion of Open Problems

The presented approach performed well at the Open Challenge demonstration at RoboCup 2011 as well as in many previously conducted tests. However, we assume that in a real tournament, it will only perform well in maybe 95% of all situations. To tackle the remaining 5% (which will be important for actually winning!), several problems still exist. As the setup of our demonstration is currently considered to be the official SPL setup in 2012, we want to briefly describe the major problems that still need to be solved:

Recover from arbitrary kidnapping situations. We currently handle a default SPL kidnapping situation, i. e. reentering after penalty. If a game runs smooth, nothing else should happen to the robot. However, in some matches, other kinds of kidnappings might occur, such as long distance manual replacements after the *READY* state or complex fall situations in which multiple robots are tangled. In our current approach the latter can be solved if the tangled robot is not in the center circle area. To overcome these problems, the robot must be enabled to detect these particular situations for executing the recovery situations. A permanent usage of the ball model for self-localization would also be possible but bears some significant risks:

Communication of wrong models. There still remains a (in our case very) small risk that the team ball model is significantly incorrect and that the recovering robot is initialized with a wrong position. After this problem occurs, the incorrectly localized robot will not only kick towards the own goal but also confuse its teammates by broadcasting wrong positions. Such kind of problems were the reason for not permanently localizing based on the ball position. On the one hand, such an approach would allow a permanent possibility for recovering and thus bears the chance of more localization stability. On the other hand, errors might accumulate and escalate to a total delocalization of the whole team which cannot be resolved anymore.

Using the teammate positions. In addition to the team ball model, it would also be possible to match the perceived teammates against their communicated positions. This might help in ambiguous situations in which the ball is close to the field's center. However, also teammates can be at this position. Furthermore, other robots are hard to perceive over large distance and, above all, cannot be distinguished from each other, i. e. a goalkeeper in its own goal appears not different from a striker close to the opponent's goal.

Single Robots. If there is only one robot on the field, the approach will not work. The minimum number of robots is two but the more robots are involved, the more robust it will work.

We assume that most of these problems can be solved by using more sophisticated approaches for state estimation and data association and thereby, a higher reliability can be achieved. However, the problem of the single robot on the field remains. It occurs rarely in actual matches

but regularly when testing. Therefore, to have a self-localization that is able to cope with *all* situations, changes to the vision system, such as memorizing unique elements outside the field by using certain descriptors, seem to be necessary.

Chapter 8

SimRobot

8.1 Introduction

The B-Human software package uses the physical robotics simulator SimRobot [18, 16] as front end for software development. The simulator is not only used for working with simulated robots, but it also functions as graphical user interface for replaying log files and connecting to actual robots via LAN or WLAN.

8.2 Architecture

There are four libraries created when SimRobot is built, which are *RoboCup*, *SimRobotCore2*, *SimRobotHelp*, and *SimRobotEditor* (cf. Fig. 8.1)[1].

SimRobotCore2 is an enhancement of the previous simulation core. It is the most important part of the SimRobot application, because it models the robots and the environment, simulates sensor readings, and executes commands given by the controller or the user. The core is platform independent and it is connected to a user interface and a controller via a well defined interface.

With the new core, the scene description language RoSiML [5] was extended and some new elements were included to improve the visualization and physics of the robots.

The library *RoboCup* 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 *RoboCup* library. In the scene files, this library is referenced by the `name` attribute within the `Scene` element.

The other two libraries contain add-ons, which can be loaded on demand. *SimRobotHelp* contains the help menu, which is shown by clicking *Help* in the menu and then *View Help* can be chosen. Unfortunately, the help system is currently both highly incomplete and outdated. *SimRobotEditor* contains a simple editor with XML syntax highlighting. This editor is intended for modifying the scene description files.

8.3 Scene View

The scene view (cf. Fig. 8.2 right) appears if the *scene* is opened from the scene graph, e. g., by double-clicking on the entry *RoboCup*. The view can be rotated around two axes, and it

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

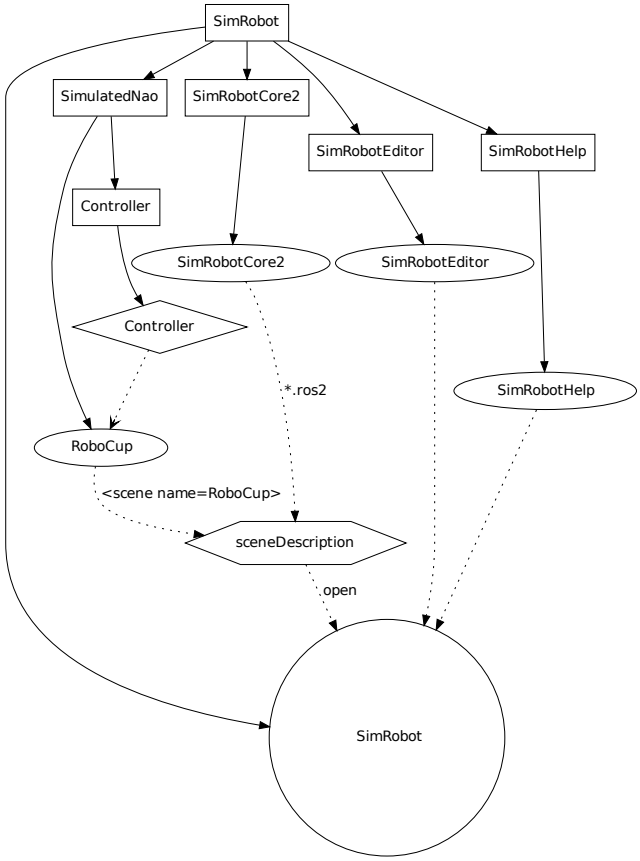


Figure 8.1: This figure shows the most important libraries of *SimRobot* (excluding foreign libraries). The rectangles represent the projects, which create the appropriate library. Dynamic libraries are represented by ellipses and the static one by a diamond. Note: the static library will be linked together with the *SimulatedNao* code. The result is the library *RoboCup*.

supports several mouse operations:

- Left-clicking an object allows dragging it to another position. Robots and the ball can be moved in that way.
- Left-clicking while pressing the *Shift* key allows rotating objects around their centers.
- Select an *active* robot by double-clicking it. Robots are active if they are defined in the compound *robots* in the scene description file (cf. Sect. 8.5).

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

8.4 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 ten kinds of views related to information

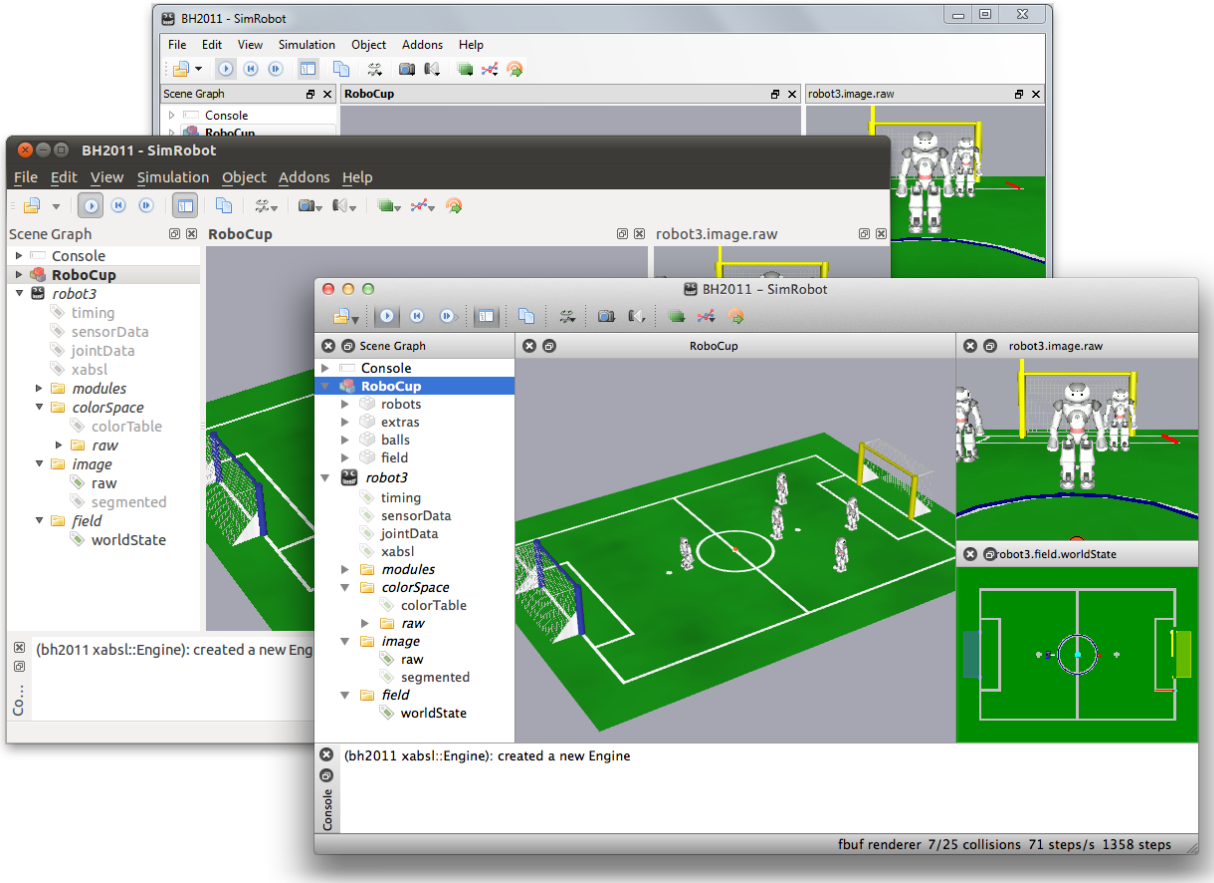


Figure 8.2: SimRobot running on Windows, Linux, and Mac OS X. The left pane shows the scene graph, the center pane shows a scene view, and on the right there are an image view and a field view. The console window is shown at the bottom.

received from robots: *image views*, *color space views*, *field views*, the *Xabsl view*, the *sensor data view*, the *joint data view*, *plot views*, the *timing view*, *module views*, and the *kick view*. Field, image, and plot views display debug drawings or plots received from the robot, whereas the other views visualize certain color channels, the current color table, specific information about the current state of the robot's behavior, its sensor readings, the timing of the modules it executes, or the module configuration itself. The *kick view* actually is a kick editor. All information views can be selected from the scene graph (cf. Fig. 8.2 left).

8.4.1 Image Views

An image view (cf. left of Fig. 8.3) displays information in the system of coordinates of the camera image. It is defined by giving it a name and a background image using the console command *vi* and by adding debug drawings to the view using the command *vid* (cf. Sect. 8.8.3).

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

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

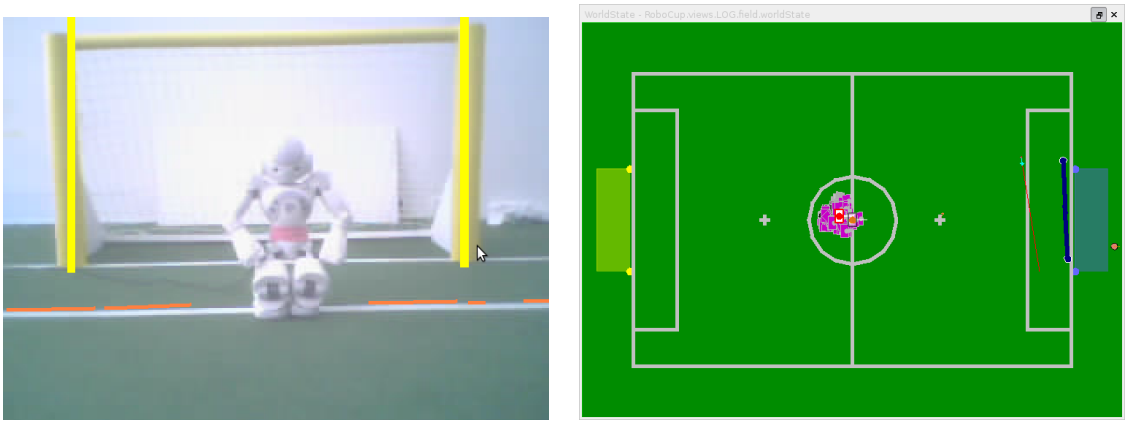


Figure 8.3: Image view and field view with several debug drawings

Image views are linked with a toolbar for color table creation. This provides fast access to some color table functions, especially to those affecting the training set for the nearest neighbor based color table. Table 8.1 provides detailed descriptions of the toolbar functions. If color table editing is activated (which it is not by default, cf. commands *ct on* and *ct off* in Sect. 8.8.3), image views will react to the following mouse commands[2]:

Left mouse button. The color of the pixel or region selected is added to the currently selected color class. Depending on the current configuration, the neighboring pixels may also be taken into account and a larger cube may be changed in the color table (cf. commands *ct imageRadius* and *ct colorSpaceRadius* in Sect. 8.8.3). However if a region is selected, the *imageRadius* is ignored. If the smart mode is active (which it is by default, cf. command *ct smart* in Sect. 8.8.3), only a subset of the colors in the region may be assigned to the selected color class, or if the selected colors are too different, none of them might be. If the training mode for nearest neighbor sets is activated, the selected pixel will also be added to the training set. In this case the colors classified in the color table serves as a preview. The actual color table produced by the tree can be generated through the reclassification command *ct reclassify* or the corresponding toolbar button (see table 8.1).

Left mouse button + Shift. If only a single pixel is selected, the color class of that pixel is chosen as the current color class. It is a shortcut for *ct <color>* (cf. Sect. 8.8.3). If a region is selected, all colors of pixels in that region that are not already assigned to a color class are assigned to the selected color class. Thus all colors in a certain region can be assigned to a color class without destroying any previous assignments.

Left mouse button + Ctrl. Undoes the previous action. Currently, up to ten steps can be undone. All commands that modify the color table can be undone, including, e. g., *ct clear* and *ct load* (cf. Sect. 8.8.3).

Left mouse button + Shift + Ctrl. The color of the pixel selected is deleted from its color class. Depending on the current configuration, the neighboring pixels may also be taken into account and a larger cube is changed in the color table (cf. commands *ct imageRadius* and *ct colorSpaceRadius* in Sect. 8.8.3). However if a region is selected, the *imageRadius* is ignored.

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

[2]Note that on Mac OS X, the key *Cmd* is used instead of *Ctrl*.

Initiates the reclassification of the color table. This generates a nearest neighbor tree from the training set and computes the final color table.

Removes the last trained color from the training vector. To let you see your training data the current color table is replaced by one, showing only the training data. To regain the final color table run the reclassification again.

Deletes the whole color table and the training set.

no color		goal yellow	
field line white		goal blue	
field green		waistband blue	
ball orange		waistband red	

Table 8.1: Clicking on one of the colors will not only select the color, it enables the training mode, too. So the color table creation can be done completely by using the mouse. "No color" can be used to explicit declare colors as uncolored to clear regions.

8.4.2 Color Space Views

Color space views visualize image information in 3-D (cf. Fig. 8.4). They can be rotated by clicking into them with the left mouse button and dragging the mouse afterwards. There are three kinds of color space views:

Color Table. This view displays the current color table in YCbCr space. Each entry that is assigned to a certain color class is displayed in a prototypical color. The view is useful while editing color tables (cf. Fig. 8.4 down right).

Image Color Space. This view displays the distribution of all pixels of an image in a certain color space (*HSI*, *RGB*, *TSL*, or *YCbCr*). It can be displayed by selecting the entry *all* for a certain color space in the scene graph (cf. Fig. 8.4 top right).

Image Color Channel. This view displays an image while using a certain color channel as height information (cf. Fig. 8.4 left).

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

```
v3 image raw
```

8.4.3 Field Views

A field view (cf. right of Fig. 8.3) 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:

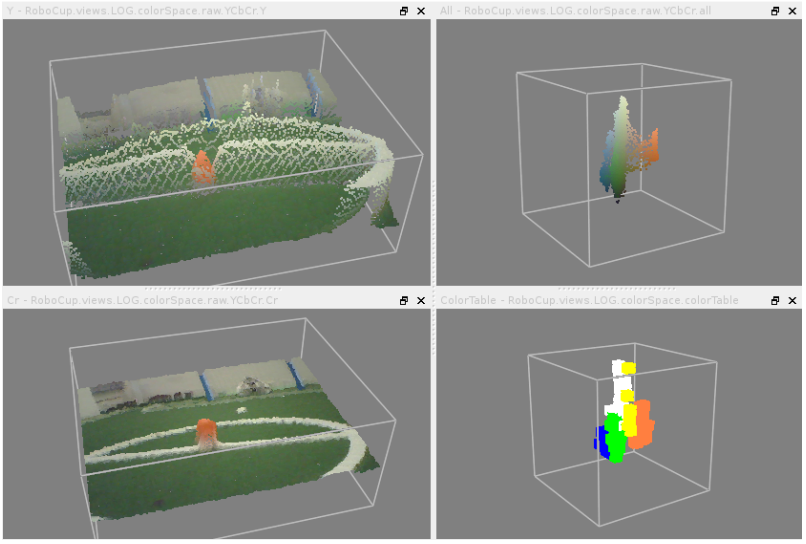


Figure 8.4: Color channel views, image color space view, and color table view

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

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

# views relative to robot
# from now, relative to estimated robot pose
vfd worldState origin:RobotPose
vfd worldState representation:BallModel
vfd worldState representation:BallPercept:Field
vfd worldState representation:LinePercept:Field
vfd worldState representation:GoalPercept:Field
vfd worldState representation:ObstacleModel
vfd worldState module:ObstacleModelProvider:us
vfd worldState representation:MotionRequest
vfd worldState representation:FreePartOfOpponentGoalModel
vfd worldState representation:RobotsModel:robots

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

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

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

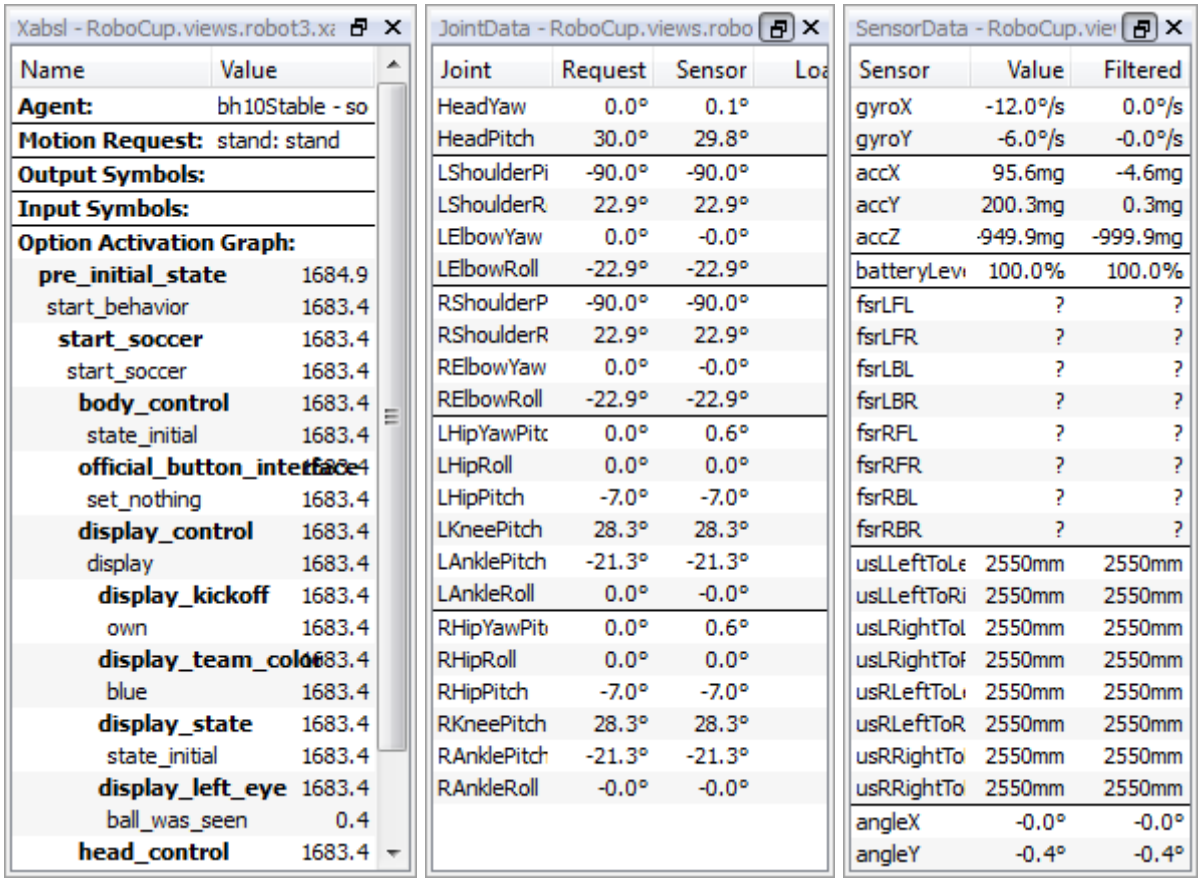


Figure 8.5: Xabsl view, joint data view and sensor data view

8.4.4 Xabsl View

The Xabsl view is part of the set of views of each robot. It displays information about the robot behavior currently executed (cf. left view in Fig. 8.5). In addition, two debug requests have to be sent (cf. Sect. 8.8.3):

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

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

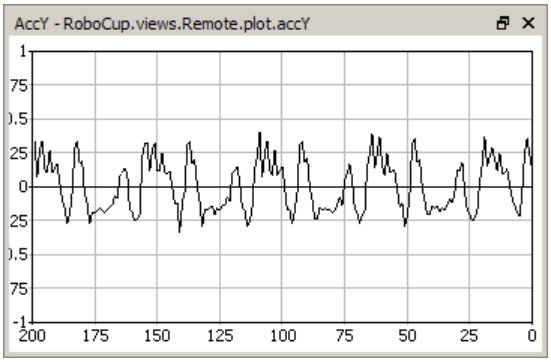
The view can also display the current values of input symbols and output symbols. The symbols to display are selected using the console commands *xis* and *xos* (cf. Sect. 8.8.3).

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

8.4.5 Sensor Data View

The sensor data view displays all the sensor data taken by the robot, e. g. accelerations, gyro measurements, pressure readings, and sonar readings (cf. right view in Fig. 8.5). To display this information, the following debug requests must be sent:

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



Timing - RoboCup.views.Remote.timing

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

Figure 8.6: The plot view and the timing view

8.4.6 Joint Data View

Similar to sensor data view the joint data view displays all the joint data taken by the robot, e. g. requested and measured joint angles, temperature and load (cf. middle view in Fig. 8.5). To display this information, the following debug requests must be sent:

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

8.4.7 Plot Views

Plot views allow plotting data sent from the robot control program through the macro `PLOT` (cf. Fig. 8.6 left). They keep a history of the values sent of a defined size. Several plots can be displayed in the same plot view in different colors. A plot view is defined by giving it a name using the console command *vp* and by adding plots to the view using the command *vpd* (cf. Sect. 8.8.3).

For instance, the view on the left side of Figure 8.6 was defined as:

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

8.4.8 Timing View

The timing view displays statistics about the speed of certain modules (cf. Fig. 8.6 right). It shows the minimum, maximum, and average runtime of the execution of a module in milliseconds. In addition, the average frequency with which the module was executed (in Hz), as well as the minimum and maximum time difference between two consecutive executions of the module (in milliseconds) is displayed. All statistics sum up the last 100 invocations of the module. The timing view only displays information on modules the debug request for sending profiling information of which was sent, i. e., to display information about the speed of the module that generates the representation *Foo*, the console command *dr stopwatch:Foo* must have been executed. Please note that time measurements are limited to full milliseconds, so the maximum and minimum execution durations will always be given in this precision. However, the average can be more precise.

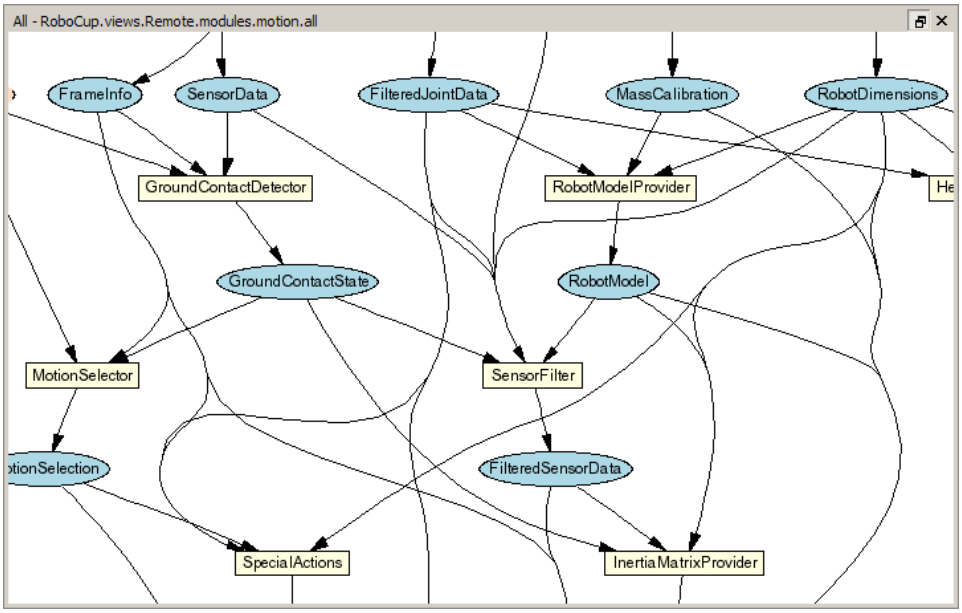


Figure 8.7: The module view shows a part of the modules in the process *Motion*

8.4.9 Module Views

Since all the information about the current module configuration can be requested from the robot control program, it is possible to generate a visual representation automatically. The graphs such as the one that is shown in Figure 8.7 are generated by the program *dot* from the *Graphviz* package [4] in the background. Modules are displayed as yellow rectangles and representations are shown as blue ellipses. Representations that are received from another process are displayed in orange and have a dashed border. If they are missing completely due to a wrong module configuration, both label and border are displayed in red. The modules of each process can either be displayed as a whole or separated into the categories that were specified as the second parameter of the macro `MAKE_MODULE` when they were defined. There is a module view for the process *Cognition* and its categories *Infrastructure*, *Perception*, *Modeling*, and *BehaviorControl*, and one for the process *Motion* and its categories *Infrastructure*, *Sensing*, and *MotionControl*.

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

8.4.10 Kick View

The basic idea of the kick view shown in Fig. 8.8 is to visualize and edit basic configurations of motions for the dynamic motion engine described in [24]. In doing so the central element of this view is a 3-D robot model. Regardless if the controller is connected to a simulated or a real robot, this model represents the current robot posture.

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

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

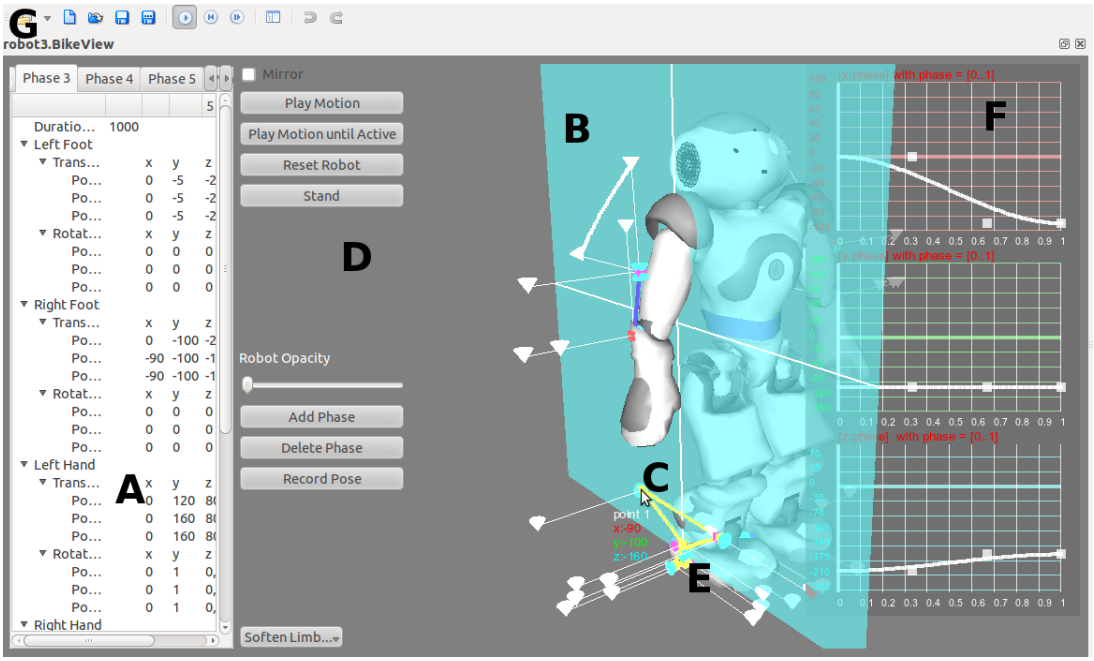


Figure 8.8: The kick view. *A* marks the editor view, *B* denotes the drag and drop plane, *C* points at a clipped curve, *D* tags the buttons that control the 3D-View, e.g. play the motion or reset the robot, *E* labels one of the control points, *F* points at the subviews and *G* points at the tool bar, where a motion can be saved or loaded.

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

3-D View. In this view each combined curve of each limb is directly attached to the robot model and therefore painted into the 3-dimensional space. Since it is useful to observe the motion curves from different angles, the view angle can be rotated by clicking with the *left mouse button* into the free space and dragging it in one direction. In order to inspect more or less details of a motion, the view can also be zoomed in or out by using the *mouse wheel* or the *page up/down* buttons.

A motion configuration is not only visualized by this view, it is also editable. Thereby the user can click on one of the visualized control points (cf. Fig. 8.8 at *E*) and drag it to the desired position. In order to visualize the current dragging plane a light blue area (cf. Fig. 8.8 at *B*) is displayed during the dragging process. This area displays the mapping between the screen and the model coordinates and can be adjusted by using the *right mouse button* and choosing the desired axis configuration.

Another feature of this view is the ability to display unreachable parts of motion curves. Since a motion curve defines the movement of a single limb, an unreachable part is a set of points that cannot be traversed by the limb due to mechanic limitations. The unreachable parts will be clipped automatically and marked with orange color (cf. Fig. 8.8 at *C*).

1-D/2-D View. In some cases a movement only happens in the 2-dimensional or 1-dimensional space (for example: Raise a leg is a movement along the z-axis only). For that reason more detailed sub views are required. Those views can be displayed as an overlay to the 3-D view by using the context menu, which opens by clicking with the right mouse button and choose *Display 1D Views* or *Display 2D Views*. This only works within the left area, where the control buttons are (cf. Fig. 8.8 left from *D*). By clciking with the right mouse button within the 3-D view the context menu for choosing the drag plane appears. The

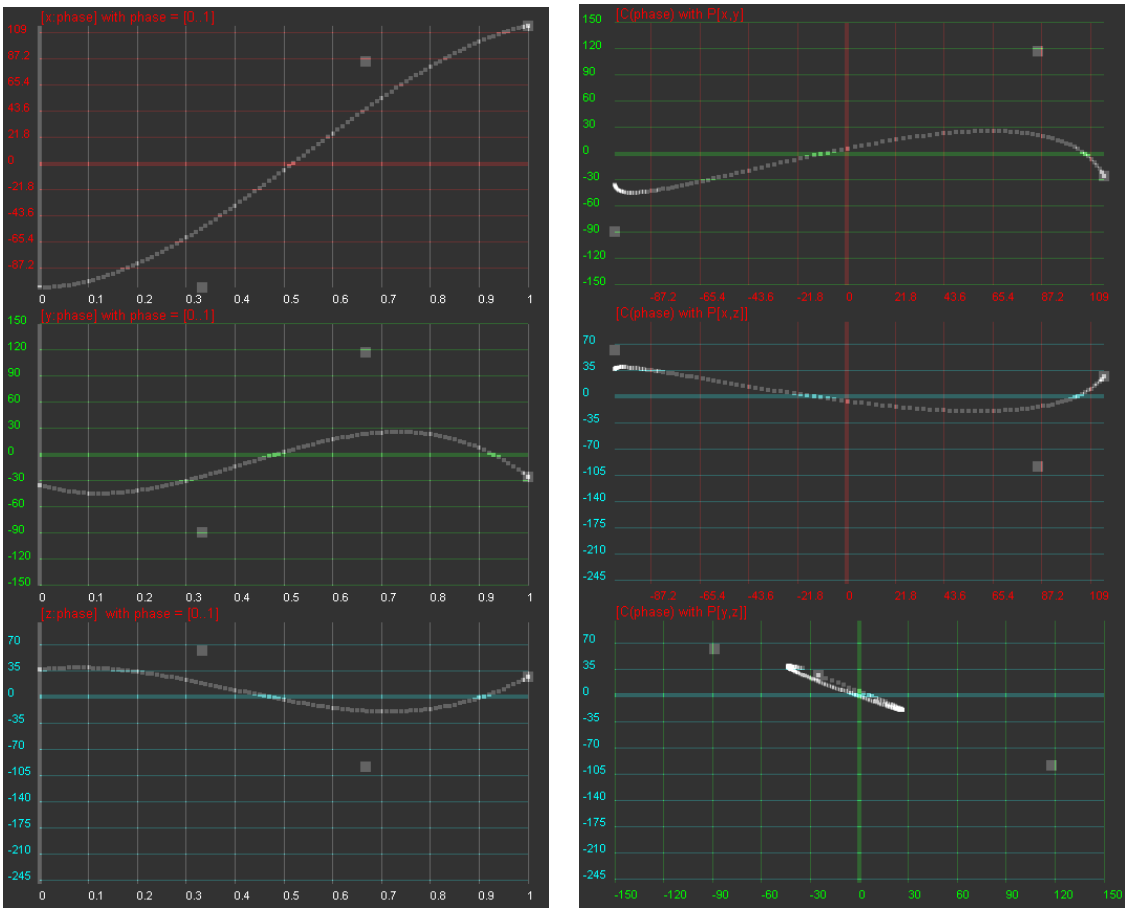


Figure 8.9: Left: 1-D sub views. Right: 2-D sub views

second opportunity to display a sub view is by clicking at the *BikeEdit* entry in the menu. So the same menu, which appears as context menu, will be displayed.

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

The 2-D view (cf. Fig. 8.9) is divided into three sub views. Each of these sub views represents only two dimensions of the activated curve. The curve displayed in the sub views is defined by equation 8.1 with $P_i = \begin{pmatrix} c_{x_i} \\ c_{y_i} \end{pmatrix}$, $P_i = \begin{pmatrix} c_{x_i} \\ c_{z_i} \end{pmatrix}$ or $P_i = \begin{pmatrix} c_{y_i} \\ c_{z_i} \end{pmatrix}$.
The 1-D sub views (cf. Fig. 8.9) are basically structured as the 2-D sub views. The difference is that each single sub view displays the relation between one dimension of the activated curve and the time t. That means that in equation 8.1 P_i is defined as: $P_i = c_{x_i}$, $P_i = c_{y_i}$, or $P_i = c_{z_i}$.

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

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

By means of this view the user is able to change certain values directly without using drag

and drop. Values directly changed will trigger a repainting of the 3-D view, and therefore, changes will be visualized immediately. This view also allows phases to be reordered by drag and drop, to add new phases, or to delete phases.

To save or load a motion the kick view has to be the active view, so some buttons to do this appearing at the tool bar (cf. Fig. 8.8 at *G*).

8.5 Scene Description Files

The language of scene description files is an extended version of RoSiML [5]. To use this new version and the new SimRobotCore2 the scene files has to end with *.ros2*, such as *BH2011.ros2*. In the following the most important elements, which are necessary to add robots, dummies and balls, are shortly described (based upon *BH2011.ros2*). For a more detailed documentation see appendix B.

<Include href="...">. First at all the descriptions of the Nao, the ball and the field are included. Those 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 *Body*-tag the *ref*-attribute specifies which nao-model should be used and *name* gets the robot a name in the scene graph of the simulation. (NOTE: the nao-model was included by the *.rsi2* file. The standard color of the Nao's waist band is set to blue. To set it to red use <Set name="NaoWaistBandColor" value="red"> within the *Body*-tag.)

<Compound name="extras">. Below the compound *robots*, there is the compound *extras*. It contains *passive* robots, i. e. robots that just stand around, but that are not controlled by a program. Passive robots can be activated by moving their definition to the compound *robots* in order to activate them, but the referenced model has to be changed into "Nao" instead of "NaoDummy".

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

<Translation>, <Rotation>. With this tags within a *Body*-tag the position of robots and balls can be modified. Both tags have the attributes *x,y,z* with the default unit meter respectively radian. To set up a rotation in degrees the unit degree has to be appended (e.g. <Rotation z="180degree">).

A lot of scene description files can be found in *Config/Scenes*. Please note that there are two types of scene description files: the ones required to simulate one or more robots, and the ones that are sufficient to connect to a real robot or to replay a log file.

- Simulating robots scenes
  - BH2011
  - BikeScene

  - Game2011
  - MultiplePlayers2011
  - OpenChallenge2011
- Scenes for connect to a real robot
  - BikeSceneRemoteWithPuppet
  - RemoteRobot
  - ReplayRobot
  - ReplayVideo
  - ScriptRemoteRobot
  - TeamComm3D
  - TeamComm

The following section describes the structure of the ball definition-file *Ball2010SPL.rsi2*, so that the appearance and size of the ball can be easily changed or additional balls can be specified.

<**Surface**> This element defines the appearance of an surface. The attribute *name* gives the surface a description the other attributes (*diffuseColor, specularColor*) specify the appearance of the surface.

<**Material name="ballPlastic"**> This elements defines the friction of the material the ball is made of.

<**Body**> With the attribute name this body gets the name *ball*. Below this tag the properties of the ball are defined.

<**SphereAppearance**> The ball gets a round appearance with the radius 0.0325 meter (the default unit for the attribute *radius* is meter if there is no other unit specified). The surface of the ball is referenced by the attribute *ref*. It was defined above, but the surface specification could be defined within this tag as well.

<**SphereGeometry**> This tag specifies the physical structure of the ball. Logically the radius of the geometry structure should be the same as the radius of the appearance definition. Within the *SphereGeometry* tag the material has to be described. In this case the material *ballPalstic* was defined above and is referenced by the *ref* attribute.

<**SphereMass**> The mass of the ball is defined with the *value* attribute. The default unit is kg.

Last but not least the following text describes the crucial structure of the field definition-file *Field2010SPL.rsi2*, so you can change the size of the field or the goals or set other colors and positions.

<**Compound name="field"**> Below this tag the field is defined. At first there are five *BoxGeometry* elements. The first one describes the whole field (carpet), which is 6 meter wide, 8 meter deep and 0.2 meter high. The friction of the material *fieldCarpet* is defined in the ball description file. The next four *BoxGeometry* elements define invisible edges around the field.

<**ComplexAppearance**> After the geometry of the field the appearance follows. The lines of the fields are defined as vertices for the half of the field, so there is a *ComplexAppearance* named *fieldLines1* and a second one named *fieldLines2*, which is rotated around 180 degrees. The next two instructions draw the cross and colors the surface of the field green.

<**Compound ref="fieldGoal"**> This instruction references the field goal, which is defined later in this file. The positioning is done by a *Transition* tag. The *Set* tag sets the color of the goal. The colors are defined after the goal definition within a *Surface* tag.

After the description of the field, the goal is specified. The goal consists of two posts, which are defined through *CylinderGeometry* and *CylinderAppearance*. Also the crossbar is a cylinder, but therefore only a *CylinderAppearance* exists. Since there is no geometry element specified for the crossbar there are no collisions between the crossbar and other objects. Then there are descriptions for the triangles of the goal and the net. After <*ComplexApperance ref="fieldNet"*> there are three *BoxGeometry* elements, which describes the internal space of the goal, so that the ball cannot roll through the goalnet or the triangles.

8.6 TeamComm3D

The TeamComm3D visualizes the data of the team communication in a SimRobot scene (*TeamComm3D.ros2*). Therefore four robots of each team color are defined in the TeamComm3D scene, which are standing along the field at their default position before they enter the field during the ready state.

The TeamComm should help to show what the robots information about the world and their position in the field is. So, for example, it is possible to see the difference between the real position a robot has on the field and the position the self locator calculates for a robot.

8.7 Logging

The framework offers multiple ways for recording and replaying logs. These are compatible because of a common log format (see figure 8.10). At the beginning of a log there are two variables in the log header. Normally these are at first the used size of the log followed by the number of messages in the log. Since sometimes a log is written incrementally, neither the used size nor the number of messages is known. In such a case a special header is inserted, setting both values to `-1` (`0xFFFFFFFF`). 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 `ProcessBegin` and `ProcessEnd` message. Every log message consists of its id, its size and its payload.

8.7.1 Recoding Logs

There are two common ways to record a log. It is possible to record a log connecting SimRobot to a Nao and make use of the *log* command. The robot sends representations, that were requested to debug, to the simulator, which stores them in a `LogPlayer` derived from `MessageQueue`. The stored messages can be managed with the given commands before they are saved in a file. This works well as long as it is possible to connect to the robot.

So another approach, presented in [36], is used to save log data without the need of a connection to the robot. This one stores the requested debug messages in a special `MessageQueue` called

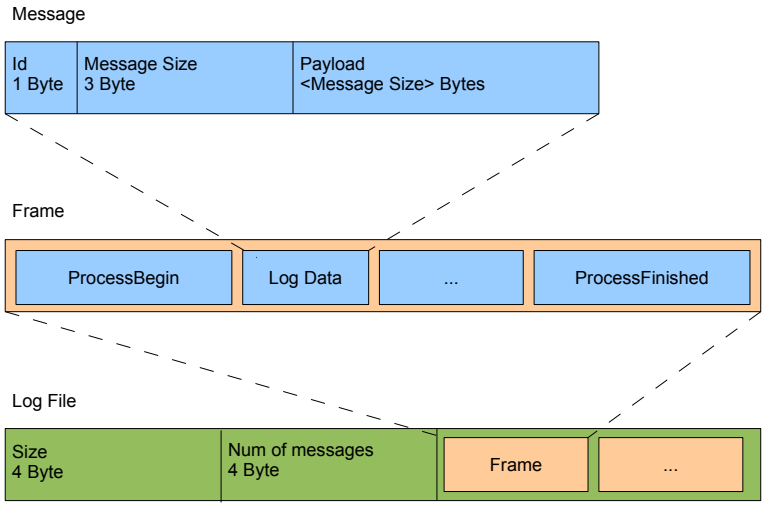


Figure 8.10: This graphic is borrowed from [36] and displays the log file format.

`LogQueue`. To prevent the loss of data, because of a full queue, the `LogQueue` saves its data to a file, when a given threshold is exceeded. Furthermore it filters the incoming messages so that only requested data is logged. To handle the different states of the logging, a state machine is implemented in `LogCtrl`. This manages the `LogQueue`, it is responsible for sending debug requests, initializing the queue and determine when to log. This approach can be configured using the *logging.cfg* file in the respective *Locations* directory:

queueSize Since the memory is finite the `LogQueue` can only reserve a certain amount of memory. This value is the maximum size of the queue in Byte. It is important to the performance of the logging. If the queue is to small, it has to write its data to a file quite often, what reduces the performance of the robot. If it is too big, it takes too much time to write the entire data to the file, which results in longer periods in which the robot is busy with the writing of the data.

logFileName This is the name of the produced log file relative to the *Config* directory.

logMode The logMode controls when log data is recorded. It can take on one of four values.

DISABLED If set to this value no log will be recorded.

ENABLED Data is recorded all the time.

ON_GAMESTATES When using this option, the robot will log data only during ready or playing game states. Other states are used to save data to the log file.

ON_GAMECONTROLLER This option is similar to the one above, beside that it only logs if a GameController is present.

logMessages This is a list of representations that shall be logged. It is advised to record at least one message that contains timestamps so the log can be played in real time (see. 8.7.2). Prefixing a '1' to a message will log the message only once. Except for Xabsl data, messages can only be logged, if they are provided WITH_OUTPUT.

logMotion Enabling this option will log the motion process data. If logMessages contains motion messages this option has to be enabled, otherwise the messages will not be logged. On the other hand, if this option is enabled, but no motion messages are to be logged, empty frames will be saved. So it is advised to activate this option only if motion data should be logged.

logCognition This option controls the logging of cognition data, like logMotion controls the logging of motion data.

8.7.2 Log Playback

Replaying a log file is done by the class `LogPlayer`. This is an extended `MessageQueue`, that loads the messages from the log file provides the data to the system frame by frame. By controlling the order in which the messages are inserted in the `targetQueue` the `LogPlayer` can control the speed of the payback. So a log can be just played as fast as the system admits, in real time, or backward. To play a log in real time timestamps are needed. Since not every representation contains timestamps it is important to pay attention to log at least one representation with timestamps, for example *FrameInfo* or *Image*.

A log file can be loaded using the *sl* command. If more than just one single log file is to be played, an offset should be used to synchronize all logs and videos. A video playback has been added to improve the analysis of game situations (see [36]). The command *sv* adds a `VideoView` showing a video. Since the simulator is based on QT the `VideoView` uses QTs multimedia framework phonon. Depending on the operating system different backends are supported, for example DirectShow on Windows, QuickTime on Mac OS and GStreamer or VLC on Linux. To navigate though replays the *log* command offers several options, that can be bound to keys (cf. Sect. 8.9.2). The playback of multiple log or video files is synchronized by `TimeCtrl`. `TimeCtrl` uses `TimeCtrlView` as user interface, that makes navigation through log files easier. A `TimeCtrlView` can be created using *stc*.

8.8 Console Commands

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

8.8.1 Initialization Commands

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

scwp <name> <a.b.c.d>. Starts a wireless connection to a real robot and attaches a 3-D robot model with it. The first parameter *name* defines the identifier that will be used for the robot and its model. The second parameter specifies the IP address of the real robot. Similar to the command *sc* this command will add a new robot to the list of available

robots, but will also add a robot model to the list of available *puppets* using *name*. The connection established between the model and the real robot will take care that all data measured by the robot – such as the joint angles – are applied to the model. The connection will be terminated, when the simulation is reset or the simulator is exited.

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

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

stc <name> Creates a TimeControl panel with the given name.

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

sv <name> <file> [<offset>] Opens a VideoView. The first parameter defines the name of the View, the second parameter specifies the name and path of the video file which will be played in the view. The third parameter is optional an controls the offset the video is started with. This enables the adjustment of the offset between a log and it's video.

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

8.8.2 Global Commands

bb on | off Switches the automatic placement for balls that are leaving the field on or off. When switched on a ball that is leaving the field is placed on the field again, according to the rules.

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

cls. Clears the console window.

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

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

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

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

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

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

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

8.8.3 Robot Commands

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

bike. Adds a kick view to the scene graph.

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

ci off | on [<fps>]. Switches the calculation of images on or off. With the optional parameter *fps*, a customized image frame rate can be set. The default value is 30. The simulation of the robot's camera image costs a lot of time, especially if several robots are simulated. In some development situations, it is better to switch off all low level processing of the

robots and to work with ground truth world states, i. e., world states that are directly delivered by the simulator. In such cases there is no need to waste processing power by calculating camera images. Therefore, it can be switched off. However, by default this option is switched on. Note that this command only has an effect on simulated robots.

ct off | **on** | **undo** | <**color**> [**replace** | **add** | <**hmin**> <**hmax**> <**smin**> <**smax**> <**imin**> <**imax**>] | **load** <**file**> | **save** <**file**> | **send** [<**ms**> | **off**] | **sendAndWrite** | **clear** [<**color**>] | **shrink** [<**color**>] | **grow** [<**color**>] | **imageRadius** <**number**> | **training** [**on** | **off**] | **reclassify** | **hash** | **kd** (**undo** | **clear** [<**color**>] | **save** <**file**> | **load** <**file**> | **replace** <**color**> <**color**> | **range** <**distance**> | **neighbors** <**count**>) | **colorSpaceRadius** <**number**> | **smart** [**off**] | **mask** <**color**>**.** This command controls editing the current color table. The parameters have the following meaning:

on | **off.** Activates or deactivates mouse handling in image views. If activated, clicking into an image view modifies the color table (cf. Sect. 8.4.1). Otherwise, mouse clicks are ignored.

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

<**color**> [**replace** | **add** | <**hmin**> <**hmax**> <**smin**> <**smax**> <**imin**> <**imax**>] Selects the given color as current color class. If nothing or replace is given existing classifications will be overwritten. If add is selected already existing classifications will not be overwritten, the classified color will belong to multiple color classes instead. If <hmin> <hmax> <smin> <smax> <imin> <imax> is given, the range selected in HSI color model coordinates will be added as the given color class.

(load | **save)** <**file**>**.** Loads or saves the color table. The default directory is the current location. The default extension is *.c64*. If there is a kt-tree training set it will be saved as well, using the same path with the extension *.kdt*.

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

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

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

training on | **off** Turn the training mode for k-d tree color classification on/off. When enabled clicking in the ImageView will produce training data.

reclassify Reclassifies the colors based on the learned k-d tree training data.

hash Calculates a sha1 hash of the color table. This allows to compare two color tables for similarity very easy.

kd undo | **clear** [<**color**>] | **save** <**file**> | **load** <**file**> | **replace** <**color**> <**color**> | **range** <**distance**> | **neighbors** <**count**>

undo Removes the last trained data from the k-d tree training set. Every step can be reverted. Even after loading a training set every step can be undone.

clear [<**color**>] Clears colors from a k-d tree training set. If no color is specified all data will be removed, otherwise only the given color is wiped out.

(load | save) <**file**> Loads/saves the k-d tree training set to a given path.

replace <**color**> <**color**> This command replaces one color in a k-d tree training set with another.

range <**distance**> This command specifies the range used to find neighbors in the reclassify process.

neighbors <**count**> This command specifies the amount of neighbors used to determine the color class during reclassification.

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

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

mask <**color**> Masks the given color class. When adding or replacing new pixel, only pixel that are already assigned to the selected color class are affected. When <color> is set to none, masking is disabled. The default value is none.

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

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

jc hide | show | motion (1 | 2) <**command**> **| (press | release)** <**button**> <**command**>**.** Sets a joystick command. If the first parameter is *press* or *release*, the number following is interpreted as the number of a joystick button. Legal numbers are between 1 and 32. Any text after this first parameter is part of the second parameter. The <*command*> parameter can contain any legal script command that will be executed in every frame while the corresponding button is pressed. The prefixes *press* or *release* restrict the execution to the corresponding event. The commands associated with the 26 first buttons can also be executed by pressing *Ctrl+Shift+A*... *Ctrl+Shift+Z* on the keyboard. If the first parameter is *motion*, an analog joystick command is defined. There are two slots for such commands, number 1 and 2, e. g., to independently control the robot's walking direction and its head. The remaining text defines a command that is executed whenever the readings of the analog joystick change. Within this command, $1... $6 can

be used as placeholders for up to six joystick axes. The scaling of the values of these axes is defined by the command *js* (see below). If the first parameter is *show*, any command executed will also be printed in the console window. *hide* will switch this feature off again, and *hide* is also the default.

jm <axis> <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...6). *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 the center itself, e. g., to compensate for the bad calibration of a joystick.

log ? | start | stop | pause | forward [image] | backward [image] | repeat | goto <number> | clear | (keep | remove) <message> {<message>} | (load | save | saveImages [raw]) <file> | cycle | once | full | jpeg. The command supports both recording and replaying log files. The latter is only possible if the current set of robot processes was created using the initialization command *sl* (cf. Sect. 8.8.1). The different parameters have the following meaning:

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

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

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

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

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

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

full | jpeg. These two commands decide whether uncompressed images received from the robot will also be written to the log file as full images, or JPEG-compressed. When the robot is connected by cable, sending uncompressed images is usually a lot faster than compressing them on the robot. By executing *log jpeg* they can still be saved

in JPEG format, saving a log memory space during recording as well as disk space later. Note that running image processing routines on JPEG images does not always give realistic results, because JPEG is not a lossless compression method, and it is optimized for human viewers, not for machine vision.

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

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

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

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

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

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

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

poll is necessary to recognize the new identifiers after the outer debug response has been activated.

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

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

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

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

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

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

saveImage [<**file**>] Saves the raw image of an robot. The image will be saved as bitmap file. If no path is specified *Config/raw_image.bmp* will be used as default option.

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

v3 ? [<**pattern**>] | <**image**> [**jpeg**] [<**name**>]**.** Adds a set of 3-D color space views for a certain image (cf. Sect. 8.4.2). The image can either be the camera image (simply specify *image*) or a debug image. It will be JPEG compressed if the option *jpeg* is specified. The last parameter is the name that will be given to the set of views. If the name is not given,

it will be the same as the name of the image. A question mark followed by an optional filter pattern will list all available images.

vf <name>. Adds a field view (cf. Sect. 8.4.3). A field view is the means for displaying debug drawings in field coordinates. The parameter defines the *name* of the view.

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

vi ? [<**pattern**>] | <**image**> [**jpeg**] [**segmented**] [<**name**>] [**gain** <**value**>] Adds an image view (cf. Sect. 8.4.1). 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 next parameter is the name that will be given to the set of views. If the name is not given, it will be the same as the name of the image plus the word *Segmented* if it should be segmented. With the last parameter the image gain can be adjusted, if no gain is specified the default value will be 1.0. A question mark followed by an optional filter pattern will list all available images.

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

vp <name> <numOfValues> <minValue> <maxValue> [<**yUnit**> <**xUnit**> <**xScale**>]**.** Adds a plot view (cf. Sect. 8.4.7). A plot view is the means for plotting data that was defined by the macro `PLOT` in the robot control program. The first parameter defines the *name* of the view. The second parameter is the number of entries in the plot, i. e. the size of the x axis. The plot view stores the last *numOfValues* data points sent for each plot and displays them. *minValue* and *maxValue* define the range of the y axis. The optional parameters serve the capability to improve the appearance of the plots by adding labels to both axes and by scaling the time-axis. The label drawing can be activated by using the context menu of the plot view.

vpd ? [<**pattern**>] | <**name**> **(?** [<**pattern**>] | <**drawing**> **(?** [<**pattern**>] | <**color**> | **off)).** Plots data in a certain color in a plot view. The first parameter is the name of a plot view that has been created using the command *vp* (see above). The second parameter

is the name of plot data that is defined in the robot control program. The third parameter defines the color for the plot. The plot is deactivated when the third parameter is *off*. The plots will be drawn in the sequence they were added, from back to front. Adding a plot a second time will move it to the front. A question mark directly after the command will list all plot views that are available. A question after a valid plot view will list all available plot data. Both question marks have an optional filter pattern that reduces the number of answers.

wek Sends walking engine kicks to the robot.

xbb ? [<**pattern**>] | **unchanged** | <**behavior**> {<**num**>}**.** Selects a Xabsl basic behavior. The command suppresses the basic behavior currently selected by the Xabsl engine and replaces it with the behavior specified by this command. Type *xbb ?* to list all available Xabsl basic behaviors. The resulting list can be shortened by specifying a search pattern after the question mark. Basic behaviors can be parametrized by a list of decimal numbers. Use *xbb unchanged* to switch back to the basic behavior currently selected by the Xabsl engine. The command *xbb* only works if the Xabsl symbols have been requested from the robot (cf. Sect. 8.4.4). Note that basic behaviors are not used anymore in the B-Human code.

xis ? [<**pattern**>] | <**inputSymbol**> **(on** | **off).** Switches the visualization of a Xabsl input symbol in the Xabsl view on or off. Type *xis ?* to list all available Xabsl input symbols. The resulting list can be shortened by specifying a search pattern after the question mark. The command *xis* only works if the Xabsl symbols have been requested from the robot (cf. Sect. 8.4.4).

xo ? [<**pattern**>] | **unchanged** | <**option**> {<**num**>}**.** Selects a Xabsl option. The command suppresses the option currently selected by the Xabsl engine and replaces it with the option specified by this command. Options can be parametrized by a list of decimal numbers. Type *xo ?* to list all available Xabsl options. The resulting list can be shortened by specifying a search pattern after the question mark. Use *xo unchanged* to switch back to the option currently selected by the Xabsl engine. The command *xo* only works if the Xabsl symbols have been requested from the robot (cf. Sect. 8.4.4).

xos ? [<**pattern**>] | <**outputSymbol**> **(on** | **off** | **?** [<**pattern**>] | **unchanged** | <**value**> **).** Shows or sets a Xabsl output symbol. The command can either switch the visualization of a Xabsl output symbol in the Xabsl view on or off, or it can suppress the state of an output symbol currently set by the Xabsl engine and replace it with the value specified by this command. Type *xos ?* to list all available Xabsl output symbols. To get the available states for a certain output symbol, type *xos* <*outputSymbol*> *?*. In both cases, the resulting list can be shortened by specifying a search pattern after the question mark. Use *xos* <*outputSymbol*> *unchanged* to switch back to the state currently set by the Xabsl engine. The command *xos* only works if the Xabsl symbols have been requested from the robot (cf. Sect. 8.4.4).

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

8.8.4 Input Selection Dialog

In scripting files it is sometimes necessary to let the user choose between some values. Therefore we implemented a small input dialog. After the user has selected a value it will then be passed

as input parameter to the scripting command. To use the input dialog you just have to type the following expression:

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

Inside the brackets there has to be a list of comma-separated values. The first value will be interpreted as a label for the input box. All following values will then be selectable via the dropdown list. The following example shows how we used it to make the IP address selectable. The user's choice will be passed to the command *sc* that establishes a remote connection to the robot with the selected IP address. Figure 8.11 shows the corresponding dialog.

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

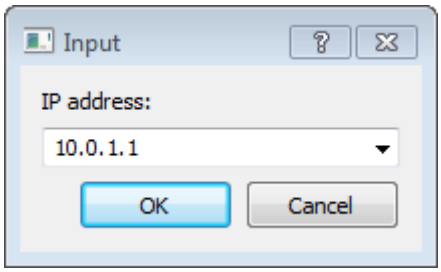


Figure 8.11: A dialog for selecting an IP address

8.9 Examples

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

8.9.1 Recording a Log File

To record a log file, the robot shall send images, joint data, sensor data, key states, odometry data, the camera matrix, and the image coordinate system. The script connects to a robot and configures it to do so. In addition, it prints several useful commands into the console window, so they can be executed by simply setting the 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 the file name behind the line *log save* is missing. Therefore, a name has to be provided to successfully execute this command.

```
# connect to a robot
sc Remote 10.1.0.101

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

# print some useful commands
echo qfr queue
echo log start
```

```
echo log stop
echo log save
echo log clear
```

8.9.2 Replaying a Log File

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

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

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

# simulation time on, otherwise log data may be skipped
st on
msg off

# all views are defined in another script
call Views

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

8.9.3 Remote Control

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

```
# connect to a robot
sc Remote 10.1.0.101

# all views are defined in another script
call ViewsJpeg

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

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

```
dr automatedRequests:xabsl:debugMessages

jc press 1 set representation:MotionRequest motion = bike;
    specialActionRequest = {specialAction = playDead; mirror = false;};
    walkRequest = { mode = targetMode; speed = { rotation = 0;
    translation = {x = 0; y = 0;}; }; target = { rotation = 0;
    translation = {x = 0; y = 0;}; }; pedantic = false; dribbling = false;
    dribbleTarget = { target = {x = 0; y = 0;}; dribbleSpeed = {x = 0; y =
       0;};
    ballPosition = { rotation = 0; translation = {x = 0; y = 0;}; };
    left = false; }; kickType = none; kickBallPosition = {x = 0; y = 0;};
    kickTarget = {x = 0; y = 0;}; }; bikeRequest = {bMotionType = kickForward;
    mirror = false; dynamical = false; ballSpecial = false; dynPoints = [];};

jc press 2 set representation:MotionRequest motion = stand;
    specialActionRequest = {specialAction = playDead; mirror = false;};
    walkRequest = { mode = targetMode; speed = { rotation = 0;
    translation = {x = 0; y = 0;}; }; target = { rotation = 0;
    translation = {x = 0; y = 0;}; }; pedantic = false; dribbling = false;
    dribbleTarget = { target = {x = 0; y = 0;}; dribbleSpeed = {x = 0; y =
       0;};
    ballPosition = { rotation = 0; translation = {x = 0; y = 0;}; };
    left = false; }; kickType = none; kickBallPosition = {x = 0; y = 0;};
    kickTarget = {x = 0; y = 0;}; }; bikeRequest = {bMotionType = kickForward;
    mirror = false; dynamical = false; ballSpecial = false; dynPoints = [];};

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

jc press 1 jc motion 2 set representation:MotionRequest motion = walk;
    specialActionRequest = {specialAction = playDead; mirror = false;};
    walkRequest = { mode = targetMode; speed = { rotation = $5;
    translation = {x = $2; y = $1;}; }; target = { rotation = 0;
    translation = {x = 0; y = 0;}; }; pedantic = false; dribbling = false;
    dribbleTarget = { target = {x = 0; y = 0;}; dribbleSpeed = {x = 0; y =
       0;};
    ballPosition = { rotation = 0; translation = {x = 0; y = 0;}; };
    left = false; }; kickType = none; kickBallPosition = {x = 0; y = 0;};
    kickTarget = {x = 0; y = 0;}; }; bikeRequest = {bMotionType = none;
    mirror = false; dynamical = false; ballSpecial = false; dynPoints = [];};
```

Chapter 9

Acknowledgements

We gratefully acknowledge the support given by Aldebaran Robotics. We thank our sponsors IGUS and Sparkasse Bremen, as well as the Deutsche Forschungsgemeinschaft (DFG) and the Deutscher Akademischer Austauschdienst (DAAD) for funding parts of our project. Since B-Human 2011 did not start its software from scratch, we also want to thank the previous team members as well as the members of the GermanTeam and of B-Smart for developing parts of the software we use.

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

AT&T Graphviz. For compiling the behavior documentation and for the module view of the simulator.
(`http://www.graphviz.org`)

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

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

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

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

libjpeg-mmx. For the Nao we use an MMX version of the libjpeg library.
(`http://mjpeg.sourceforge.net`)

XABSL. For implementing the robot's behavior.
(`http://www.xabsl.de`)

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

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

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

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

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

zbuildgen. Creates and updates the Makefiles and Visual Studio project files.
(`http://host07.ath.cx/zbuildgen`)

Appendix A

Configuration Files

bodyContour.cfg contains parameters for the BodyContourProvider (cf. Sect. 4.1.3).

bush.cfg contains parameters for the B-Human User Shell (cf. Sect. 2.7).

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

cameraControlEngine.cfg contains parameters for the CameraControlEngine (cf. Sect. 4.1.2).

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

groundContact.cfg contains parameters for the GroundContactDetector (cf. Sect. 5.1.2). It can be configured per robot too.

Images is used to save exported images from the simulator.

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

Keys contains the SSH keys needed by *Copyfiles* and the *B-Human User Shell* to connect remotely to the Nao.

linePerceptor.cfg contains parameters for the LinePerceptor (cf. Sect. 4.1.5.3).

Locations contains one directory for each location.

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

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

Locations/<location>/coltableNew.c64 and coltableOld.c64 are the color table that is used for this location. Since we use two generations of cameras whose difference cannot be adjusted by the camera settings there are one color table for each camera generation. Which color table is used is decided in *cameraCalibration.cfg*.

Locations/<location>/combinedWorldModel.cfg contains parameters for the CombinedWorldModelProvider (cf. Sect. 4.2.10).

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

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

Locations/<location>/footPerceptor.cfg contains parameters for the FootPerceptor (cf. Sect. 4.1.5.7).

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

Locations/<location>/logging.cfg contains parameters for recording logs automatically, without a direct connection to the robot (cf. Sect. 8.7.1).

Locations/<location>/matchStatistic.cfg contains parameters for the MatchStatisticProvider (cf. Sect. 6.4).

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

Locations/<location>/obstacleCombinator.cfg contains parameters for the ObstacleCombinator (cf. Sect. 4.2.7).

Locations/<location>/pathFinder.cfg contains parameters for the PathFinder (cf. Sect. 6.3.6

Locations/<location>/Robots/<robot>/ballPerceptor.cfg contains parameters for the BallPerceptor (cf. Sect. 4.1.5.5).

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

Locations/<location>/usObstacleGrid.cfg contains parameters for the USObstacleGridProvier which provides the by the ObstacleCombinator required *USObstacleGrid* (cf. Sect. 4.2.7).

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

odometry.cfg provides information for the self-locator while executing special actions. See the file or Section 5.2.3 for more explanations.

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

regionAnalyzer.cfg contains parameters for the RegionAnalyzer (cf. Sect. 4.1.5.2).

regionizer.cfg contains parameters for the Regionizer (cf. Sect. 4.1.5.1).

robotLocatorUKF.cfg contains parameters for the RobotLocatorUKF (cf. Sect. 4.1.5.6).

robotPerceptor.cfg contains parameters for the RobotPerceptor (cf. Sect. 4.1.5.6) and the RobotLocator (cf. Sect. 4.2.4).

Robots contains one directory for each robot and the settings of the robot. The configuration files found here are used for individual calibration settings for each robot. The directory *Nao* is used by the simulator. For each robot, a subdirectory with the name of the robot must exist. There also is a directory called *Default*. Whenever a file needed is not found in the directory of the current robot, the corresponding file from the directory *Default* is used.

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

Robots/<robotName>/damageConfiguration.cfg is optional. It tracks if the sensors which are used to detect the ground contact are broken.

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

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

Robots/<robotName>/robot.cfg contains parameters for the B-Human User Shell (cf. Sect. 2.7).

Robots/<robotName>/robotDimensions.cfg contains values that are used by forward and inverse kinematics (cf. Sect. 5.2.2).

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

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

Scenes contains different scenes for the simulator.

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

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

teamMarkerPerceptor.cfg contains parameters for the TeamMarkerPerceptor (cf. Sect. 4.1.5.6).

teams.cfg contains parameters for the B-Human User Shell (cf. Sect. 2.7) and is used to as alternative to the *settings.cfg* on the robot if it is not present.

walking.cfg contains parameters for the WalkingEngine (cf. Sect. 5.2.1).

Appendix B

The Scene Description Language

B.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 used symbols.

- 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 !
  $!x, y, ...!$ 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 $Hinge$ = ...!*bodyClass*, *axisClass*!... can be written as $Hinge$ = ...*bodyClass axisClass*....

- +[...]+
  $+[x, y]+$ 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.

B.2 Grammar

```
appearanceClass      = Appearance | BoxAppearance | SphereAppearance
                     | CylinderAppearance | ComplexAppearance;
axisClass            = Axis;
bodyClass            = Body | Bumper;
compoundClass        = Compound;
deflectionClass      = Deflection;
```

```
extSensorClass       = Camera | DepthImageSensor | SingleDistanceSensor
                     | ApproxDistanceSensor;
frictionClass        = Friction | RollingFriction;
geometryClass        = Geometry | BoxGeometry | CylinderGeometry
                     | SphereGeometry;
infrastructureClass  = Simulation | Include;
intSensorClass       = Accelerometer | Gyroscope;
jointClass           = Hinge | Slider;
lightClass           = Light;
massClass            = Mass | BoxMass | InertiaMatrixMass | SphereMass;
materialClass        = Material;
motorClass           = ServoMotor | VelocityMotor;
primitiveGroupClass  = Quads | Triangles;
rotationClass        = Rotation;
sceneClass           = Scene;
setClass             = Set;
solverClass          = Quicksolver;
texCoordsClass       = TexCoords;
translationClass     = Translation;
verticesClass        = Vertices;


Accelerometer        = "<Accelerometer></Accelerometer>" | "<Accelerometer/>";
Gyroscope            = "<Gyroscope></Gyroscope>" | "<Gyroscope/>";

Appearance           = "<Appearance>" ?( +[translationClass, rotationClass]+
                       {setClass | appearanceClass} )? "</Appearance>";
BoxAppearance        = "<BoxAppearance>" ?( !surfaceClass! +[translationClass,
                       rotationClass]+ {setClass | appearanceClass} )?
                       "</BoxAppearance>";
ComplexAppearance    = "<ComplexAppearance>" ?( !surfaceClass, verticesClass,
                       primitiveGroupClass! +[translationClass, rotationClass
                       | texCoordsClass]+ {setClass | appearanceClass
                       | primitiveGroupClass} )? "</ComplexAppearance>";
CylinderAppearance   = "<CylinderAppearance>" ?( !surfaceClass!
                       +[translationClass, rotationClass]+
                       {setClass | appearanceClass} )? "</CylinderAppearance>";
SphereAppearance     = "<SphereAppearance>" ?( !surfaceClass!
                       +[translationClass, rotationClass]+
                       {setClass | appearanceClass} )? "</SphereAppearance>";

ApproxDistanceSensor = "<ApproxDistanceSensor>" ?( +[translationClass,
                        rotationClass]+ )? "</ApproxDistanceSensor>";
Camera               = "<Camera>" ?( +[translationClass, rotationClass]+ )?
                       "</Camera>";
DepthImageSensor     = "<DepthImageSensor>" ?( +[translationClass,
                       rotationClass]+ )? "</DepthImageSensor>";
SingleDistanceSensor = "<SingleDistanceSensor>" ?( +[translationClass,
                       rotationClass]+ )? "</SingleDistanceSensor>";

Mass                 = "<Mass>" ?( +[translationClass, rotationClass]+
                       {setClass | massClass} )? "</Mass>";
BoxMass              = "<BoxMass>" ?( +[translationClass | rotationClass]+
                       {setClass | massClass} )? "</BoxMass>";
InertiaMatrixMass    = "<InertiaMatrixMass>"
                       ?( +[translationClass, rotationClass]+
```

```
                        {setClass | massClass} )? "</InertiaMatrixMass>";
SphereMass            = "<SphereMass>" ?( +[translationClass | rotationClass]+
                        {setClass | massClass} )? "</SphereMass>"

Geometry              = "<Geometry>" ?( +[translationClass, rotationClass,
                        materialClass]+ {setClass | geometryClass} )?
                        "</Geometry>";
BoxGeometry           = "<BoxGeometry>" ?( +[translationClass, rotationClass,
                        materialClass]+ {setClass | geometryClass} )?
                        "</BoxGeometry>";
CylinderGeometry      = "<CylinderGeometry>"
                        ?( +[translationClass, rotationClass, materialClass]+
                        {setClass | geometryClass} )? "</CylinderGeometry>";
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>";
Bumper                = "<Bumper>" ?( !massClass! +[translationClass,
                        rotationClass]+ {setClass | jointClass | appearanceClass
                        | geometryClass | massClass | intSensorClass |
                        extSensorClass} )? "</Bumper>";

Material              = "<Material>" ?( {setClass | frictionClass} )?
                        "</Material>";
Friction              = "<Friction></Friction>" | "<Friction/>";
RollingFriction       = "<RollingFriction></RollingFriction>"
                      | "<RollingFriction/>";

ServoMotor            = "<ServoMotor></ServoMotor>" | "</ServoMotor>";
VelocityMotor         = "<VelocityMotor></VelocityMotor>" | "</VelocityMotor>";

Simulation            = "<Simulation>" !sceneClass! "</Simulation>";
Scene                 = "<Scene>" ?( +[solverClass]+ {setClass | bodyClass
                        | compoundClass | lightClass} )? "</Scene>";

Compound              = "<Compound>" ?( +[translationClass, rotationClass]+
                        {setClass | compoundClass | bodyClass | appearanceClass
                        | geometryClass | extSensorClass} )? "</Compound>";
Deflection            = "<Deflection></Deflection>" | "<Deflection/>";
Include               = "<Include></Include>" | "<Include/>";
Light                 = "<Light></Light>" | "<Light/>";
Set                   = "<Set></Set>" | "<Set/>";

Rotation              = "<Rotation></Rotation>" | "<Rotation>";
Translation           = "<Translation></Translation>" | "<Translation/>";
```

```
Quads                = "<Quads>" Quads Definition "</Quads>";
TexCoords            = "<TexCoords>" TexCoords Definition "</TexCoords>";
Triangles            = "<Triangles>" Triangles Definition "</Triangles>";
Vertices             = "<Vertices> Vertices Definition "</Vertices>";
```

B.3 Structure of a Scene Description File

B.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 name attribute (in our case all scenes set the name attribute to *RoboCup*, so that the libRoboCup is loaded by SimRobot). It is recommended to include other specifications per *Include* before the scene description starts (compare with BH2011.ros2), but it is not necessary.

B.3.2 The ref Attribute

An element with a name attribute can be referenced by the *ref*-attribute using its name, i. e. elements that are needed repeatedly in a scene need to be defined only once. For example there is only one description of the structure of a goal in the field description file (Field2010SPL.rsi2), but both a yellow and a blue goal are needed on a field. So there are two references to the goal definition. The positioning of the two goals is done by *Translation* and *Rotation* elements. The color is set by a *Set* element, which is described below.

```
⋮
<Compound ref="fieldGoal" name="yellowGoal">
  <Translation x="-3m"/>
  <Set name="color" value="fieldYellow"/>
</Compound>
<Compound ref="fieldGoal" name="blueGoal">
  <Translation x="3m"/>
  <Rotation z="180degree"/>
  <Set name="color" value="fieldBlue"/>
</Compound>
⋮
<Compound name="fieldGoal">
  <CylinderGeometry height="800mm" radius="50mm" name="post1">
    <Translation y="-700mm" z="400mm"/>
  </CylinderGeometry>
⋮
```

B.3.3 Placeholders and Set Element

A placeholder has to start with a $ followed by a arbitrary string. A placeholder is replaced by the definition specified within the corresponding *Set* element. The attribute *name* of a *Set* elements specifies the placeholder, which is replaced by the value specified by the attribute *value* of the *Set* element.

In the following code example, the color of a post of a goal is set by a *Set* element. Within the definition of the compound fieldGoal named yellowGoal, the *Set* element sets the placeholder color to the value fieldYellow. The placeholder named color of post1, which is defined in the general definition of a field goal, is replaced by fieldYellow. So the *Surface* elements reference a *Surface* named fieldYellow.

```
⋮
<CylinderAppearance height="800mm" radius="50mm" name="post1">
  <Translation y="-700mm" z="400mm"/>
  <Surface ref="$color"/>
</CylinderAppearance>
⋮
<Compound ref="fieldGoal" name="yellowGoal">
  <Translation x="-3m"/>
  <Set name="color" value="fieldYellow"/>
</Compound>
⋮
<Surface name="fieldYellow" diffuseColor="rgb(60, 60, 0)" ambientColor="rgb(45, 45,
    0)" specularColor="rgb(40, 40, 40)" shininess="30"/>$\vdots$
```

B.4 Attributes

B.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.

B.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

B.4.3 sceneClass

- **Scene** Describes a scene and specifies the controller of the simulation.
  - name The name of the controller library (without prefix lib. In our case it is RoboCup).

    - **Use:** optional
    - **Range:** String
  - color The background color of the scene, see B.4.22.
  - stepLength
    - **Units:** s
    - **Default:** 0.01s
    - **Use:** optional
    - **Range:** $(0, MAXFLOAT]$
  - gravity Sets the gravity in this scene.
    - **Units:** $\frac{mm}{s^2}$, $\frac{m}{s^2}$
    - **Default:** $-9.80665\frac{m}{s^2}$
    - **Use:** optional
  - CFM Sets ODE cfm (constraint force mixing) value.
    - **Default:** -1
    - **Use:** optional
    - **Range:** $[0, 1]$
  - ERP Set ODE erp (error reducing parameter) value.
    - **Default:** -1
    - **Use:** optional
    - **Range:** $[0, 1]$
  - contactSoftERP Sets another erp value for colliding surfaces.
    - **Default:** -1
    - **Use:** optional
    - **Range:** $[0, 1]$
  - contactSoftCFM Sets another cfm value for colliding surfaces.
    - **Default:** -1
    - **Use:** optional
    - **Range:** $[0, 1]$

B.4.4 solverClass

- **Quicksolver**
  - iterations
    - **Default:** -1
    - **Use:** optional
    - **Range:** $(0, MAXINTEGER]$
  - skip
    - **Default:** 1
    - **Use:** optional
    - **Range:** $(0, MAXINTEGER]$

B.4.5 bodyClass

- **Body** Specifies an object that has a mass and can move.
  - name The name of the body.
    - **Use:** optional
    - **Range:** String
- **Bumper**
  - name The name of the bumper.
    - **Use:** optional
    - **Range:** String

B.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

B.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

B.4.8 massClass

- **Mass** All this mass classes define the mass of an object.
  - name The name of the mass declaration.
    - **Use:** optional
    - **Range:** String
- **BoxMass**

  - name The name of the boxMass declaration.
    - **Use:** optional
    - **Range:** String
  - value The mass of the box.
    - **Units:** g, kg
    - **Default:** 0
    - **Use:** required
    - **Range:** $[0, MAXFLOAT]$
  - width The width of the box.
    - **Units:** mm, cm, dm, m, km
    - **Default:** 0
    - **Use:** required
    - **Range:** $[-MAXFLOAT, MAXFLOAT]$
  - height The height of the box.
    - **Units:** mm, cm, dm, m, km
    - **Default:** 0
    - **Use:** required
    - **Range:** $[-MAXFLOAT, MAXFLOAT]$
  - depth The depth of the box.
    - **Units:** mm, cm, dm, m, km
    - **Default:** 0
    - **Use:** required
    - **Range:** $[-MAXFLOAT, MAXFLOAT]$

- **SphereMass**
  - name The name of the sphereMass declaration.
    - **Use:** optional
    - **Range:** String
  - value The mass of the sphere.
    - **Units:** g, kg
    - **Default:** 0
    - **Use:** required
    - **Range:** $[0, MAXFLOAT]$
  - radius The radius of the sphere.
    - **Units:** mm, cm, dm, m, km
    - **Default:** 0
    - **Use:** required
    - **Range:** $[0, MAXFLOAT]$

- **InertiaMatrixMass** The matrix of the mass moment of inertia. Note that this matrix is a symmetric matrix.
  - name The name of the InertiaMatrixMass declaration.
    - **Use:** optional

    - **Range:** String
- value The total mass.
    - **Units:** g, kg
    - **Default:** 0
    - **Use:** required
    - **Range:** $[0, MAXFLOAT]$
- x The center of mass in x direction.
    - **Units:** mm,cm,dm,m,km
    - **Default:** 0
    - **Use:** optional
    - **Range:** $[-MAXFLOAT, MAXFLOAT]$
- y The center of mass in y direction.
    - **Units:** mm,cm,dm,m,km
    - **Default:** 0
    - **Use:** optional
    - **Range:** $[-MAXFLOAT, MAXFLOAT]$
- z The center of mass in z direction.
    - **Units:** mm,cm,dm,m,km
    - **Default:** 0
    - **Use:** optional
    - **Range:** $[-MAXFLOAT, MAXFLOAT]$
- ixx Moment of inertia around the x-axis when the object is rotated around the x-axis.
    - **Units:** $g * mm^2$, $kg * m^2$
    - **Default:** 0
    - **Use:** required
    - **Range:** $[-MAXFLOAT, MAXFLOAT]$
- ixy Moment of inertia around the y-axis when the object is rotated around the x-axis.
    - **Units:** $g * mm^2$, $kg * m^2$
    - **Default:** 0
    - **Use:** optional
    - **Range:** $[-MAXFLOAT, MAXFLOAT]$
- ixz Moment of inertia around the z-axis when the object is rotated around the x-axis.
    - **Units:** $g * mm^2$, $kg * m^2$
    - **Default:** 0
    - **Use:** optional
    - **Range:** $[-MAXFLOAT, MAXFLOAT]$
- iyy Moment of inertia around the y-axis when the object is rotated around the y-axis.
    - **Units:** $g * mm^2$, $kg * m^2$
    - **Default:** 0
    - **Use:** required
    - **Range:** $[-MAXFLOAT, MAXFLOAT]$
- iyz Moment of inertia around the z-axis when the object is rotated around the y-axis

    - **Units:** $g * mm^2$, $kg * m^2$
    - **Default:** 0
    - **Use:** optional
    - **Range:** $[-MAXFLOAT, MAXFLOAT]$
  - izz Moment of inertia around the z-axis when the object is rotated around the z-axis.
    - **Units:** $g * mm^2$, $kg * m^2$
    - **Default:** 0
    - **Use:** required
    - **Range:** $[-MAXFLOAT, MAXFLOAT]$

B.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 B.4.22
  - name
    - **Use:** optional
    - **Range:** String
  - width The width of the box.
    - **Units:** mm, cm, dm, m, km
    - **Default:** 0
    - **Use:** required
    - **Range:** $[-MAXFLOAT, MAXFLOAT]$
  - height The height of the box.
    - **Units:** mm, cm, dm, m, km
    - **Default:** 0
    - **Use:** required
    - **Range:** $[-MAXFLOAT, MAXFLOAT]$
  - depth The depths of the box.
    - **Units:** mm, cm, dm, m, km
    - **Default:** 0
    - **Use:** required
    - **Range:** $[-MAXFLOAT, MAXFLOAT]$
- **SphereGeometry**
  - color A color definition, see B.4.22
  - name
    - **Use:** optional

    - **Range:** String
  - radius The radius of the sphere.
    - **Units:** mm, cm, dm, m, km
    - **Default:** 0
    - **Use:** required
    - **Range:** $[-MAXFLOAT, MAXFLOAT]$

- **CylinderGeometry**
  - color A color definition, see B.4.22
  - name
    - **Use:** optional
    - **Range:** String
  - radius The radius of the cylinder.
    - **Units:** mm, cm, dm, m, km
    - **Default:** 0
    - **Use:** required
    - **Range:** $[-MAXFLOAT, MAXFLOAT]$
  - height The height of the cylinder.
    - **Units:** mm, cm, dm, m, km
    - **Default:** 0
    - **Use:** required
    - **Range:** $[-MAXFLOAT, MAXFLOAT]$

B.4.10 materialClass

- **Material** Specifies a material.
  - **Use:** required
  - **Range:** String
  - name The name of the material.
    - **Use:** optional
    - **Range:** String

B.4.11 frictionClass

- **Friction** Specifies the friction of a material.
  - material The material the friction belongs to.
    - **Use:** required
    - **Range:** String
  - value The value of the friction.
    - **Default:** 1
    - **Use:** required

    * **Range:** $[0, MAXFLOAT]$

- **RollingFriction** Specifies the rolling friction of an material.
  - material The material the rolling friction belongs to.
    * **Use:** required
    * **Range:** String
  - value The value of the rolling friction.
    * **Default:** 1
    * **Use:** required
    * **Range:** $[0, MAXFLOAT]$

B.4.12 appearanceClass

- **Appearance** The appearance elements specify only shapes for the surfaces, so all appearance elements require a *Surface* specification. Appearance elements do not have a physical structure. Therefore a geometry has to be defined.
  - name The name of this appearance.
    * **Use:** optional
    * **Range:** String

- **BoxAppearance**
  - name The name of this appearance. To specify how it should look like an element of the type surfaceClass is needed.
    * **Use:** optional
    * **Range:** String
  - width The width of the box.
    * **Units:** mm, cm, dm, m, km
    * **Default:** 0
    * **Use:** required
    * **Range:** $[-MAXFLOAT, MAXFLOAT]$
  - height The height of the box.
    * **Units:** mm, cm, dm, m, km
    * **Default:** 0
    * **Use:** required
    * **Range:** $[-MAXFLOAT, MAXFLOAT]$
  - depth The depth of the box.
    * **Units:** mm, cm, dm, m, km
    * **Default:** 0
    * **Use:** required
    * **Range:** $[-MAXFLOAT, MAXFLOAT]$

- **SphereAppearance**
  - name

    - **Use:** optional
    - **Range:** String
  - radius The radius of the sphere.
    - **Units:** mm, cm, dm, m, km
    - **Default:** 0
    - **Use:** required
    - **Range:** $[-MAXFLOAT, MAXFLOAT]$

- **CylinderAppearance**
  - name
    - **Use:** optional
    - **Range:** String
  - height The height of the cylinder.
    - **Units:** mm, cm, dm, m, km
    - **Default:** 0
    - **Use:** required
    - **Range:** $[-MAXFLOAT, MAXFLOAT]$
  - radius The radius of the cylinder.
    - **Units:** mm, cm, dm, m, km
    - **Default:** 0
    - **Use:** required
    - **Range:** $[-MAXFLOAT, MAXFLOAT]$

- **ComplexAppearance**
  - name
    - **Use:** optional
    - **Range:** String

B.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:** $[-MAXFLOAT, MAXFLOAT]$

B.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

B.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:** $[0, 1]$

B.4.16 deflectionClass

- **Deflection** Specifies the maximum and minimum deflection of a joint.
  - min The minimal deflection.
    - **Default:** 0
    - **Use:** required
    - **Range:** $[-MAXFLOAT, MAXFLOAT]$
  - max The maximal deflection..
    - **Default:** 0
    - **Use:** required
    - **Range:** $[-MAXFLOAT, MAXFLOAT]$
  - stopCFM
    - **Default:** -1
    - **Use:** optional
    - **Range:** $[0, 1]$
  - stopERP
    - **Default:** -1
    - **Use:** optional
    - **Range:** $[0, 1]$

B.4.17 motorClass

- **ServoMotor**
  - maxVelocity The maximum velocity of this motor.
    - **Units:** radian/s, degree/s
    - **Default:** 0
    - **Use:** required
    - **Range:** $[-MAXFLOAT, MAXFLOAT]$
  - maxForce The maximum force of this motor.
    - **Units:** N
    - **Default:** 0
    - **Use:** required
    - **Range:** $[-MAXFLOAT, MAXFLOAT]$
  - p The p value of the motor's pid interface.
    - **Default:** 0
    - **Use:** required
    - **Range:** $[-MAXFLOAT, MAXFLOAT]$
  - i The i value of the motor's pid interface.
    - **Default:** 0
    - **Use:** optional
    - **Range:** $[-MAXFLOAT, MAXFLOAT]$
  - d The d value of the motor's pid interface.

    * **Default:** 0
    * **Use:** optional
    * **Range:** $[-MAXFLOAT, MAXFLOAT]$

- **VelocityMotor**
  - maxVelocity The maximum velocity of this motor.
    * **Units:** radian/s, degree/s
    * **Default:** 0
    * **Use:** required
    * **Range:** $[-MAXFLOAT, MAXFLOAT]$
  - maxForce The maximum force of this motor.
    * **Units:** N
    * **Default:** 0
    * **Use:** required
    * **Range:** $[-MAXFLOAT, MAXFLOAT]$

B.4.18 surfaceClass

- **Surface** Defines the appearance of a surface.
  - diffuseColor The diffuse color, see B.4.22.
  - ambientColor The ambient color of the light, see B.4.22.
  - specularColor The specular color, see B.4.22.
  - emissionColor The emitted color of the light, see B.4.22.
  - shininess The shininess value.
    * **Use:** optional
    * **Range:** o
  - diffuseTexture A texture.
    * **Use:** optional
    * **Range:** o

B.4.19 intSensorClass

- **Gyroscope** Mounts a gyroscope on a body.
  - name The name of the gyroscope.
    * **Use:** optional
    * **Range:** String
- **Accelerometer** Mounts a accelerometer on a body.
  - name The name of the accelerometer.
    * **Use:** optional
    * **Range:** String

B.4.20 extSensorClass

- **Camera** Mounts a camera on a body.
  - name Name of the camera.
    - **Use:** optional
    - **Range:** String
  - imageWidth The width of the camera image.
    - **Default:** 0
    - **Use:** required
    - **Range:** Integer > 0
  - imageHeight The height of the camera image.
    - **Default:** 0
    - **Use:** required
    - **Range:** Integer > 0
  - angleX Opening angle in x.
    - **Units:** degree, radian
    - **Default:** 0
    - **Use:** required
    - **Range:** Float > 0
  - angleY Opening angle in y.
    - **Units:** degree, radian
    - **Default:** 0
    - **Use:** required
    - **Range:** Float > 0
- **SingleDistanceSensor**
  - name The name of the sensor.
    - **Use:** optional
    - **Range:** String
  - min The minimum distance this sensor can measure.
    - **Default:** 0
    - **Use:** optional
    - **Range:** $[-MAXFLOAT, MAXFLOAT]$
  - max The maximum distance this sensor can measure.
    - **Default:** 999999.f
    - **Use:** optional
    - **Range:** $[-MAXFLOAT, MAXFLOAT]$
- **ApproxDistanceSensor**
  - name The name of the sensor.
    - **Use:** optional
    - **Range:** String

  - min The minimum distance this sensor can measure.
    - **Default:** 0
    - **Use:** optional
    - **Range:** $[-MAXFLOAT, MAXFLOAT]$
  - max The maximum distance this sensor can measure.
    - **Default:** 999999.f
    - **Use:** optional
    - **Range:** $[-MAXFLOAT, MAXFLOAT]$
  - angleX The maximum angle in x-direction the ray of the sensor can spread.
    - **Units:** degree, radian
    - **Default:** 0
    - **Use:** required
    - **Range:** Float > 0
  - angleY The maximum angle in y-direction the ray of the sensor can spread.
    - **Units:** degree, radian
    - **Default:** 0
    - **Use:** required
    - **Range:** Float > 0

- **DepthImageSensor**
  - name
    - **Use:** optional
    - **Range:** String
  - imageWidth The width of the image.
    - **Default:** 0
    - **Use:** required
    - **Range:** Integer > 0
  - imageHeight The height of the image.
    - **Default:** 1
    - **Use:** optional
    - **Range:** Integer > 0
  - angleX
    - **Units:** degree, radian
    - **Default:** 0
    - **Use:** required
    - **Range:** Float > 0
  - angleY
    - **Units:** degree, radian
    - **Default:** 0
    - **Use:** required
    - **Range:** Float > 0
  - min The minimum distance this sensor can measure.

    - **Default:** 0
    - **Use:** optional
    - **Range:** $[-MAXFLOAT, MAXFLOAT]$
  - max The maximum distance this sensor can measure.
    - **Default:** 999999.f
    - **Use:** optional
    - **Range:** $[-MAXFLOAT, MAXFLOAT]$
  - projection The kind of projection.
    - **Default:** 0
    - **Use:** optional
    - **Range:** perspective, spheric

B.4.21 lightClass

- **Light** Definition of a light source.
  - diffuseColor Diffuse color definition, see B.4.22
  - ambientColor Ambient color definition, see B.4.22
  - specularColor Specular color definition, see B.4.22
  - x The x position of the light source.
    - **Units:** mm,cm,dm,m,km
    - **Use:** optional
    - **Range:** o
  - y The y position of the light source.
    - **Units:** mm,cm,dm,m,km
    - **Use:** optional
    - **Range:** o
  - z The z position of the light source.
    - **Units:** mm,cm,dm,m,km
    - **Use:** optional
    - **Range:** o
  - constantAttenuation The constant attenuation of the light.
    - **Use:** optional
    - **Range:** $[0.f, MAXFLOAT]$
  - linearAttenuation The linear attenuation of the light.
    - **Use:** optional
    - **Range:** $[0.f, MAXFLOAT]$
  - quadraticAttenuation The quadratic attenuation of the light.
    - **Use:** optional
    - **Range:** $[0.f, MAXFLOAT]$
  - spotCutoff
    - **Units:** mm,cm,dm,m,km

    - **Use:** optional
    - **Range:** $[-MAXFLOAT, MAXFLOAT]$
  - spotDirectionX The x direction of the light spot.
    - **Units:** mm,cm,dm,m,km
    - **Use:** optional
    - **Range:** $[-MAXFLOAT, MAXFLOAT]$
  - spotDirectionY The y direction of the light spot.
    - **Units:** mm,cm,dm,m,km
    - **Use:** optional
    - **Range:** $[-MAXFLOAT, MAXFLOAT]$
  - spotDirectionZ The z direction of the light spot.
    - **Units:** mm,cm,dm,m,km
    - **Use:** optional
    - **Range:** $[-MAXFLOAT, MAXFLOAT]$
  - spotExponent
    - **Use:** optional
    - **Range:** $[0.f, 128.f]$

B.4.22 Color Specification

There two ways of specifying a color for a color-attribute.

- **HTML-Style** To specify a color in html-style the first sign of the color value has to be a # followed by hexadecimal values for red, blue, green (and maybe a fourth value for the alpha-channel. These values can be one-digit or two-digits, but not mixed.
  - #rgb e.g. #f00
  - #rgba e.g. #0f0a
  - #rrggbb e.g. #f80011
  - #rrggbbaa e.g. #1038bc
- **CSS-Style** A css color starts with rgb (or rgba) followed by the values for red, green, blue put into brackets and separated by commas. The values for r,g,b has to be between 0 and 255 or between 0% and 100%, the a-value has to be between 0 and 1.
  - rgb(r,g,b) e.g. rgb(255, 128, 0)
  - rgba(r,g,b,a) e.g. rgba(0%, 50%, 75%, 0.75)

Bibliography

[1] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18:509–517, September 1975.

[2] Alexander Fabisch, Tim Laue, and Thomas Röfer. Robot Recognition and Modeling in the RoboCup Standard Platform League. In Enrico Pagello, Changjiu Zhou, Sven Behnke, Emanuele Menegatti, Thomas Röfer, and Peter Stone, editors, *Proceedings of the Fifth Workshop on Humanoid Soccer Robots in conjunction with the 2010 IEEE-RAS International Conference on Humanoid Robots*, Nashville, TN, USA, 2010.

[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] 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.

[5] Keyan Ghazi-Zahedi, Tim Laue, Thomas Röfer, Peter Schöll, Kai Spiess, Arndt Twickel, and Steffen Wischmann. RoSiML - Robot Simulation Markup Language, 2005. `http://www.informatik.uni-bremen.de/spprobocup/RoSiML.html`.

[6] Colin Graf and Thomas Röfer. A center of mass observing 3D-LIPM gait for the RoboCup Standard Platform League humanoid. In Thomas Röfer, Norbert Michael Mayer, Jesus Savage, and Uluc Saranli, editors, *RoboCup 2011: Robot Soccer World Cup XV*, Lecture Notes in Artificial Intelligence. Springer, 2011.

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

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

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

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

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

[12] James J. Kuffner and Steven M. LaValle. RRT-connect: An efficient approach to single-query path planning. In *Proceedings of the 2000 IEEE International Conference on Robotics and Automation (ICRA 2000)*, volume 2, pages 995–1001, San Francisco, CA, USA, 2000.

[13] Tim Laue, Thijs Jeffry de Haas, Armin Burchardt, Colin Graf, Thomas Röfer, Alexander Härtl, and Andrik Rieskamp. Efficient and reliable sensor models for humanoid soccer robot self-localization. In Changjiu Zhou, Enrico Pagello, Emanuele Menegatti, Sven Behnke, and Thomas Röfer, editors, *Proceedings of the Fourth Workshop on Humanoid Soccer Robots in conjunction with the 2009 IEEE-RAS International Conference on Humanoid Robots*, pages 22 – 29, Paris, France, 2009.

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

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

[16] 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.

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

[18] 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.

[19] 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.

[20] Steven M. LaValle and James J. Kuffner. Randomized kinodynamic planning. In *Proceedings of the 1999 IEEE International Conference on Robotics and Automation (ICRA 1999)*, volume 1, pages 473–479, Detroit, MI, USA, 1999.

[21] 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.

[22] Martin Loetzsch, Max Risler, and Matthias Jüngel. XABSL - A Pragmatic Approach to Behavior Engineering . In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2006)*, pages 5124–5129, Beijing, China, 2006.

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

[24] 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.

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

[26] Max Risler. *Behavior Control for Single and Multiple Autonomous Agents Based on Hierarchical Finite State Machines*. Fortschritt-berichte vdi, Technische Universität Darmstadt, May 15 2009.

[27] Max Risler and Oskar von Stryk. Formal behavior specification of multi-robot systems using hierarchical state machines in XABSL. In *Proceedings of the AAMAS08-Workshop on Formal Models and Methods for Multi-Robot Systems*, Estoril, Portugal, 2008.

[28] 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.

[29] 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.

[30] Thomas Röfer, Tim Laue, Armin Burchardt, Erik Damrose, Katharina Gillmann, Colin Graf, Thijs Jeffry de Haas, Alexander Härtl, Andrik Rieskamp, André Schreck, and Jan-Hendrik Worch. B-Human Team Report and Code Release 2008, 2008. Only available online: `http://www.b-human.de/file_download/16/bhuman08_coderelease.pdf`.

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

[32] Thomas Röfer, Tim Laue, Judith Müller, Armin Burchardt, Eric Damrose, Alexander Fabisch, Fynn Feldpausch, Katharina Gillmann, Colin Graf, Thijs Jeffry de Haas, Alexander Härtl, Daniel Honsel, Philipp Kastner, Tobias Kastner, Benjamin Markowsky, Michael Mester, Jonas Peter, Ole Jan Lars Riemann, Martin Ring, Wiebke Sauerland, André Schreck, Ingo Sieverdingbeck, Felix Wenk, and Jan-Hendrik Worch. B-Human Team Report and Code Release 2010, 2010. Only available online: `http://www.b-human.de/file_download/33/bhuman10_coderelease.pdf`.

[33] Thomas Röfer, Tim Laue, Judith Müller, Alexander Fabisch, Katharina Gillmann, Colin Graf, Alexander Härtl, Arne Humann, and Felix Wenk. B-Human Team Description for RoboCup 2011, 2011.

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

[35] 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`.

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

[37] Stefan Zickler, Tim Laue, Oliver Birbach, Mahisorn Wongphati, and Manuela Veloso. SSL-Vision: The Shared Vision System for the RoboCup Small Size League. In Jacky Baltes, Michail G. Lagoudakis, Tadashi Naruse, and Saeed Shiry, editors, *RoboCup 2009: Robot Soccer World Cup XIII*, volume 5949 of *Lecture Notes in Artificial Intelligence*, pages 425–436. Springer, 2010.