B-Human

Team Report and Code Release 2008

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

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

Revision: November 5, 2008

Contents

Chapter 1

Introduction

1.1 About us

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

The senior team members have also been part of a number of 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 by the GermanTeam, 2008 by B-Smart), and winning the Four-Legged League Technical Challenge twice (2003 and 2007 by the GermanTeam). In 2007, the team was strengthened by further computer science students, who support the team on the basis of their advanced study project.

In parallel to these activities, B-Human started as a part of the joint team BreDoBrothers, which has been a cooperation of the Technische Universität Dortmund and the Universität Bremen. The team participated in the Humanoid League in RoboCup 2006. The software was based on previous works of the GermanTeam [10]. 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, and were the only team that actually won games in the preliminaries. The team members from the Technische Universität Dortmund provided an own team report [4] about their contribution for that competition.

The current team consists of the following persons:

Diploma Students. Oliver Bösche, Armin Burchardt, Erik Damrose, Katharina Gillmann, Colin Graf, Alexander Härtl, Thijs Jeffry de Haas, Mirko Kloweit, Judith Müller, Thanh Nguyen Hai, Andrik Rieskamp, André Schreck, Ingo Sieverdingbeck, Karl Trzebiatowski, Thiemo Wiedemeyer, Jan-Hendrik Worch.

Researcher. Tim Laue.

Senior Researcher and Assistant Professor. Thomas Röfer (team leader).

1.2 About the Document

In the beginning of the Standard Platform Four-Legged League, it was obligatory to release the developed source code after a competition. With this document we want to try to revive the tradition of the annual code release which has been widely neglected in the past few years.

This document gives a survey about our current system. It reflects the transition B-Human is currently undergoing. The code is basically a system designed for the Humanoid League now running on the robot *Nao* of Aldebaran Robotics, i. e., image processing and world modeling are still configured for the field of the Humanoid Kid-Size League. In addition, the behavior control implements a rather rudimentary *goto-ball-and-kick* behavior, but it already listens to the GameController of the Standard Platform League. As mentioned above, some team members of B-Human were also part of the team BreDoBrothers in the Standard Platform League. Apart from some exceptions, our code release only contains software which was originally introduced to the joint team by members of B-human or it was written after the cooperation was finished.

Chapter 2 starts with a short introduction to the required software, as well as an explanation of how to run the Nao with our software. Chapter 3 gives an introduction to the software framework. Chapter 4 deals with the cognition system and will give an overview about our perception and modeling components. In Chapter 5, we describe our walking approach and how to create special motion patterns. Chapter 6 gives an overview about the robot behavior contained in the code release, i. e. the implementation of a simple striker. Finally, Chapter 7 describes the usage of SimRobot, the program that is both used as simulator and as debugging frontend when controlling real robots.

Chapter 2

Getting Started

To use our code release, several steps are necessary: Unpacking the source code, compiling the code using Visual Studio 2008 or Linux, setting up the Nao, copying the files to the robot, and starting the software.

2.1 Unpacking

Unpack the source code to a location, the path of which must *not* contain whitespaces. The code release contains several subdirectories, which are described below.

Backup contains all files needed to set up a new Nao.

Build contains temporary files created during the compilation of the source code.

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

Doc contains a description how to create special motion patterns (cf. Sect. 5.2 as well as the compiled documentation of the behavior.

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

Src contains all the source code of the code release.

Util contains additional tools such as Doxygen [17].

2.2 Components and Configurations

The code release bundle contains a full version of the B-Human software usable on Windows as well as on Linux. The code release builds a shared library for *NaoQi* running of the real robot, as well as the same software running in our simulator SimRobot (without NaoQi). The B-Human software consists of the following components:

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

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

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

Simulator is the executable simulator (cf. Chapter 7) for running and controlling the B-Human robot code. The robot code links against the components *SimRobotCore*, *SimRobotGUI*, *Controller* and some third-party libraries. It is compilable in *Optimized*, *Debug With Release Libs*, and *Debug* configurations. All these configurations contain debug code but *Optimized* performs some optimizations and strips debug symbols (not on Windows). *Debug With Release Libs* produces debuggable robot code while linking against non-debuggable *Release* libraries.

Nao compiles the shared library for *NaoQi*. It is available in *Release*, *Optimized*, and *Debug* configurations, where *Release* produces "game code" without any support for debugging. The configuration *Optimized* produces optimized code, but still supports all debugging techniques described in Section 3.5.

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).

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

SpecialActions compiles motion patterns (*.mof* files) into an internal format (cf. Sect. 5.2).

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

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

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

2.3 Compiling using Visual Studio 2008

2.3.1 Required Software

- Visual Studio 2008 SP1
- cygwin – 1.5 with the following additional packages: make, ruby, rsync, openssh, libxml2, libxslt. Add the *...\cygwin\bin* directory to the PATH environment variable.
- gcc, glibc – Linux cross compiler for cygwin, download from http://sourceforge.net/project/showfiles.php?group_id=135860 , in order to keep symbolic links use a cygwin shell to extract.
- alcommon – copy the contents of *extern/c/aldebaran/alcommon* from the *Nao SDK release v0.1.xx linux* (*NaoQi-0.1.xx-Linux.tar.gz*) to the directory *Util/alcommon* of the unpacked

B-Human software. The package is available at the internal RoboCup download area of Aldebaran Robotics. Please note that this package is only required to compile the code for the actual Nao robot.

- Bonjour – Apple's *Bonjour* is used by the *NaoFinder* (cf. Sect. 2.7) to detect the IP addresses of all Naos of B-Human in the network. (www.apple.com)

2.3.2 Compiling

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

2.4 Compiling on Linux

2.4.1 Required Software

Additional requirements (listed by common package names) for a x686 based Linux distribution (e. g. Ubuntu Hardy):

- g++, make
- libqt4-dev – 4.3 or above (www.trolltech.com)
- ruby
- doxygen – For compiling the documentation.
- graphviz – For compiling the behavior documentation and for using the module view of the simulator. (www.graphviz.org)
- xsltproc – For compiling the behavior documentation.
- openssh-client – For deploying compiled code to the Nao.
- rsync – For deploying compiled code to the Nao.
- alcommon – copy the contents of *extern/c/aldebaran/alcommon* from the *Nao SDK release v0.1.xx linux* (*NaoQi-0.1.xx-Linux.tar.gz*) to the directory *Util/alcommon* of the unpacked B-Human software. The package is available at the internal RoboCup download area of Aldebaran Robotics. Please note that this package is only required to compile the code for the actual Nao robot.

2.4.2 Compiling

All the components (cf. Sect. 2.2) are also available for Linux. To compile one of them, simply set *Make* to the current working directory and type:

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

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

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

To clean up the whole solution use:

```
make clean
```

2.5 Configuration Files

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

odometry.cfg provides information for the selflocator while executing Special Actions. See the file or section 5.2 for more explanations.

pointsColorChange.cfg contains parameters for the ColorChangePerceptor.

pointsSegments.cfg contains parameters for the SegmentsPerceptor.

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. 3.4.4). The *teamColor* determines the color of the own goal (blue or yellow). The *playerNumber* must be different for each robot of the team. It is used to identify a robot by the GameController and in the team communication. In addition, it can be used in behavior control. *location* determines which directory in the *Location* subdirectory is used to retrieve the location-dependent settings.

walking.cfg contains parameters for the WalkingEngine (cf. Sect. 5.1.1).

Keys contains the SSH keys needed by the script *copyfiles* to connect remotely to the Nao.

Locations contains one directory for each location. These directories control settings that depend on the environment, i. e. the lighting or the field layout. It can be switched quickly between different locations by setting the according value in the *settings.cfg*. Thus different field definitions, color tables, and behaviors can be prepared and loaded.

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

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

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

Locations/<location>/field.cfg contains coordinates and field sizes.

Locations/<location>/goalLocator.cfg contains configuration parameters for the module GoalLocatorLight.

Locations/<location>/modules.cfg contains information about which representations are available and which module provides them while the code is running. Representations which should be available in both processes need to be given in the section *Shared*.

Locations/<location>/selfloc.cfg contains parameters for the module SelfLocator.

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

Robots/<robotName>/cameraCalibration.cfg contains correction values for camera roll and tilt.

Robots/<robotName>/jointCalibration.cfg contains calibration values for each joint. In this file offset, sign, minimal and maximal joint angles can be set individually.

Robots/<robotName>/robotDimensions.cfg contains values which are used by the inverse kinematics.

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

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

Scenes contains different scenes for the simulator.

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

2.6 Setting up the Nao

Setting up the Nao is only possible from a Linux OS. First of all, get the *Nao SDK release v0.1.18 linux* for Linux and the USB flash drive image *Flashdrive/usb bootable + NaoQi 0.1.18* which are both available at the internal RoboCup download area of Aldebaran Robotics. We have not yet tried more recent versions. Furthermore, *dosfstools* and *gpart* are needed and usually included in a Linux distribution. Unpack the SDK and the flashdrive image file.

Open the head of the Nao and remove the USB flash memory. Plug the USB stick into the computer. If the operating system automatically mounts the USB stick, unmount it to prevent unintended data corruption. The shell script *tools/flashusbnaokey* from the SDK has to be executed with the file of the image (*opennao-image-usb-nao-geode-0.1.18.ext3*) as its parameter. The executable-bit must be set (*chmod +x tools/flashusbnaokey*). The script renews the first partition of the flash drive, but does not touch the userdata partition.

Afterwards, mount the userdata partition (FAT) of the flash drive and copy the directory *Backup* from this code release to the stick. After starting the Nao with the USB flash device inserted, connect to the Nao via SSH and enter the directory */media/userdata/Backup* (that was just copied). Finally, start the install script *./install*. Afterwards, the Nao is ready to receive the compiled B-Human code.

2.7 Finding the Network Address

The Nao will query a *dhcp*-Server for a network address. Once a lease is acquired it will speak its network address. A *Multicast-DNS* server is used to publish the network address in the LAN. Linux users can use *avahi-browse* to find a running Nao in the network.

```
$ avahi-browse -r _bhuman-robot._tcp
+ eth0 IPv4 Nao52                                        _bhuman-robot._tcp   local
= eth0 IPv4 Nao52                                        _bhuman-robot._tcp   local
   hostname = [Nao52.local]
   address = [134.102.204.245]
   port = [0]
   txt = ["url=http://www.b-human.de/"]
```

Windows users can use the console based *NaoFinder.exe* in *Util/NaoFinder*.

2.8 Copying the Compiled Code

To copy the compiled code and the configuration files onto the Nao, the file *copyfiles.cmd* for Windows or *copyfiles.sh* for Linux has to be executed.

copyfiles requires two obligatory parameters. First, the configuration the code was compiled with (*NaoDebug*, *NaoOptimized*, or *NaoRelease*), and second, the IP address of the robot. To adjust the desired settings, it is possible to set the following optional parameters:

Option	Description
-l <location>	sets the location
-t <color>	sets the team color to blue or yellow
-p <number>	sets the player number
-s	copies the binary with debug symbols
-d	deletes the cache directory

A possible call could be:

```
copyfiles NaoOptimized 10.0.1.103 -t yellow -p 2
```

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

2.9 Working with the Nao

After pressing the chest button, it takes about 45 seconds until NaoQi is started. Currently the B-Human software is compiled as a shared library (*libbhuman.so*), and it is loaded by *NaoQi*.

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

./stop stops the instance of *NaoQi* that is automatically started during boot time. This should be done before *copyfiles* is used.

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

The Nao can but shut down in two different ways:

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

harakiri --deep && shutdown -h now will shut down the Nao. If the Nao runs on battery it will be completely switched off after a couple of seconds. In this case an external power supply is needed to start the Nao again. If the B-Human software is running, this can also be done by pressing the chest button and one of the foot buttons for more than three seconds.

2.10 Starting SimRobot

On Windows, the simulator can either be started from Microsoft Developer Studio, or by starting a scene description file in *Config/Scenes*[1]. In the first two cases, a scene description file has to be opened manually, whereas it will already be loaded in the latter case. On Linux, just run *Build/Simulator/Linux/<configuration>/Simulator*, and load a scene description file afterwards. When a simulation is started for the first time, the main window only shows the tree view in the left pane and the console window in the bottom pane. Select *Simulation/Start* to run the simulation. The scene tree will appear in the tree view. A scene view showing the soccer field can be opened by double-clicking *scene RoboCup*. The view can be adjusted by using the context menu of the window or the toolbar.

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

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

To connect to a real Nao, enter its IP address in the file *Config/Scenes/connect.con* on the PC. Afterwards, start the simulation scene *Config/Scenes/RemoteRobot.ros* (cf. Sect. 7.6.3). A remote connection to the Nao is only possible if the code running on the Nao was compiled in either the *Debug* or the *Optimized* configuration.

For more detailed information about SimRobot see Chapter 7.

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

Chapter 3

Architecture

The B-Human architecture is based on the framework of the GermanTeam 2007 [15]. Since most of this framework has not been documented yet, at least not in its present form, this chapter summarizes the major features of the architecture: processes, modules and representations, communication, and debugging support.

3.1 Processes

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

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

3.2 Modules and Representations

A robot control program usually consists of several modules each of which performs a certain task, e. g. image processing, self-localization, or walking. Modules require are certain input and produce a certain output (i. e. so-called *representations*). Therefore, they have to be executed

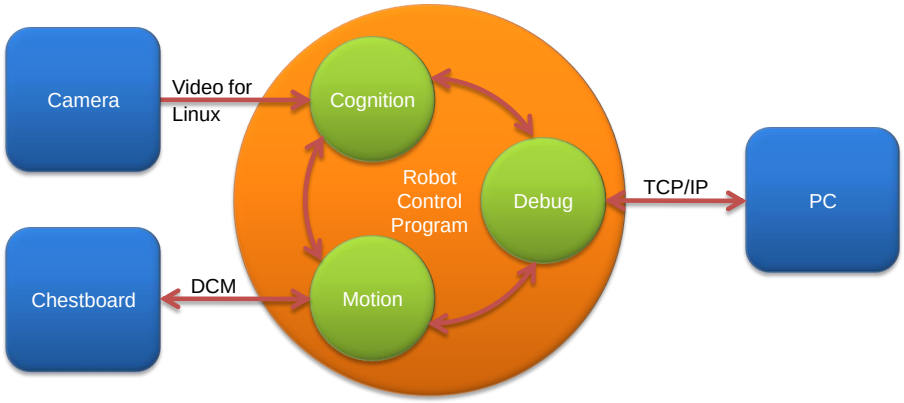

Figure 3.1: The processes used on the Nao

in a specific order to make the whole system work. The module framework introduced in [15] 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. 7.3.8).

3.2.1 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 require to instantiate an image object (approximately 300 KB in size).

3.2.2 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 3.2.3, 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. 7.3.8).

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. 7.3.8), 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. 3.5.6) and the representations provided can be sent to a host PC or even altered by it.

The latter functionality is achieved by variants of the macro *PROVIDES* that add support for *MODIFY* (cf. Sect. 3.5.5), support for streaming the representation to be recorded in a log file (*OUTPUT*, requires a *message id* with the same name as the representation, cf. Sect. 3.4), and drawing based on a parameterless method *draw* implemented by the representation itself. The maximum version of the macro is *PROVIDES_WITH_MODIFY_AND_OUTPUT_AND_DRAW*. For a reduced functionality, the sections of the name that are not required or not supported can be left out.

Besides the macro *REQUIRES*, there also is the macro *USES(<representation>)*. *USES* simply gives access to a certain representation, without defining any dependencies. Thereby, a module

can access a representation that will be updated later, accessing its state from the previous frame. Hence, *USES* can be used to model cyclic relations. The module view (cf. Sect. 7.3.8) does not display *USES* connections.

3.2.3 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 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.

3.2.4 Pseudo-Module *default*

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

3.3 Streams

In most applications, it is necessary that data can be serialized, i. e. transformed into a sequence of bytes. While this is straightforward for data structures that already consist of a single block of memory, it is a more complex task for dynamic structures, as e. g. lists, trees, or graphs. The implementation presented in this document follows the ideas introduced by the C++ iostreams library, i. e., the operators << and >> are used to implement the process

of serialization. It is also possible to derive classes from class *Streamable* and implement the mandatory method *serialize(In*, Out*)*. In addition, the basic concept of streaming data was extended by a mechanism to gather information on the structure of the data while serializing it.

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

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

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

3.3.1 Streams Available

Currently, the following classes are implemented:

PhysicalOutStream. Abstract class

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

StreamWriter. Abstract class

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

Out. Abstract class

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

OutTextSize. Determine memory size for text storage
OutTextRawSize. Determine memory size for raw text storage
OutBinaryMessage. Writing binary into a MessageQueue
OutTextMessage. Writing into a MessageQueue as text
OutTextRawMessage. Writing into a MessageQueue as raw text

PhysicalInStream. Abstract class

InFile. Reading from files

InMemory. Reading from memory

InMessageQueue. Reading from a MessageQueue

StreamReader. Abstract class

InBinary. Binary reading

InText. Reading data as text

InConfig. Reading configuration file data from streams

In. Abstract class

InStream<PhysicalInStream,StreamReader>. Abstract class template
InBinaryFile. Reading from binary files
InTextFile. Reading from text files
InConfigFile. Reading from configuration files
InBinaryMemory. Reading binary data from memory
InTextMemory. Reading text data from memory
InConfigMemory. Reading config-file-style text data from memory
InBinaryMessage. Reading binary data from a MessageQueue
InTextMessage. Reading text data from a MessageQueue
InConfigMessage. Reading config-file-style text data from a MessageQueue

3.3.2 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);
```

3.3.3 Making Classes Streamable

A class is made streamable by deriving it from the class *Streamable* and implementing the abstract method *serialize(In*, Out*)*. For data types derived from *Streamable* streaming operators are provided, meaning they may be used as any other data type with standard streaming operators implemented. To realize the *modify* functionality (cf. Sect. 3.5.5), the streaming method uses macros to acquire structural information about the data streamed. This includes the data types of the data streamed as well as that names of attributes. The process of acquiring names and types of members of data types is automated. The following macros can be used to specify the data to stream in the method *serialize*:

STREAM_REGISTER_BEGIN() indicates the start of a streaming operation.

STREAM_BASE(<class>) streams the base class.

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

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

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

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

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

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

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

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

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

3.4 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*.

3.4.1 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.

3.4.2 Inter-process Communication

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

3.4.3 Debug Communication

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

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

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

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

The process *Debug* manages the communication of the robot control program with the tools on the PC. For each of the other processes (*Cognition* and *Motion*) it has a sender and a receiver for their debug message queues (cf. Fig. 3.1). Messages that arrive via WLAN from the PC are stored in *debugIn*. The method *Debug::handleMessage(InMessage&)* distributes all messages in *debugIn* to the other processes. The messages received from *Cognition* and *Motion* are stored in *debugOut*. When a WLAN connection is established, they are sent to the PC via TCP/IP. To avoid communication jams, it is possible to send a *QueueFillRequest* to the process *Debug*. The command *qfr* to do so is explained in Section 7.5.3.

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

3.5 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 3.4.3. 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.

3.5.1 Debug Requests

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

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

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

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

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

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

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

This statement sends the text "Hello" if the debug request with the name *"sayHello"* is activated. Please note that only those debug requests are usable that are in the current path of execution. This means that only debug request in those modules can be activated that are currently executed. To determine which debug requests are currently available, a method called *polling* is employed. It asks all debug responses to report the name of the debug request that would activate it. This information is collected and sent to the PC (cf. command *poll* in Sect. 7.5.3).

3.5.2 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. 7.3.1) or in a color space view (cf. Sect. 7.3.2). Each debug image has an associated textual identifier that allows referring to it during image manipulation, as well as for requesting its creation from the PC. The identifier can be used in a number of macros that are defined in file *Src/Tools/Debugging/DebugImages.h*, and that facilitate the manipulation of the debug image.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

3.5.3 Debug Drawings

Debug drawings provide a virtual 2-D drawing paper and a number of drawing primitives, as well as mechanisms for requesting, sending, and drawing these primitives to the screen of the PC. In contrast to debug images, which are raster-based, debug drawings are vector-based, i. e., they store drawing instructions instead of a rasterized image. Each drawing has an identifier and an associated type that enables the application on the PC to render the drawing to the right kind of drawing paper. In addition, a description can be specified (currently, it is not used). In the B-Human system, two standard drawing papers are provided, called *drawingOnImage* and *drawingOnField*. This refers to the two standard applications of debug drawings, namely drawing in the system of coordinates of an image and drawing in the system of coordinates of the field. Hence, all debug drawings of the type *drawingOnImage* can be displayed in an image view (cf. Sect. 7.3.1) and all drawings of type *drawingOnField* can be rendered into a field view (cf. Sect. 7.3.3).

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

DECLARE_DEBUG_DRAWING(<id>, <type>, <description>) declares a debug drawing with the specified *id*, *type*, and *description*. 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", "draws a test debug drawing");
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.

3.5.4 Plots

The macro *PLOT(<id>, <number>)* allows plotting data over time. The plot view (cf. Sect. 7.3.6) 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. 7.5.3).

For example, the following statement plots the measurements of the gyro for the pitch axis. Please note that the measurements are converted to degrees, because the current implementation of the plot view can only display integer numbers. So the range of the values must be scaled up accordingly.

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

3.5.5 Modify

The macro *MODIFY(<id>, <object>)* allows reading and modifying of data on the actual robot during runtime. Every streamable data type (cf. Sect. 3.3.3) 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. 7.5.3). The first parameter of *MODIFY* specifies the identifier that is used to refer to the object from the PC, the second parameter is the object to be manipulated itself. When an object is modified using the console command *set*, it will be overridden each time the *MODIFY* macro is executed.

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

The macro *PROVIDES* of the module framework (cf. Sect. 3.2) also is available in versions that include the *MODIFY* macro for the representation provided (e. g. *PROVIDES_WITH_MODIFY*). In these cases the representation, e. g., *Foo* is modifiable under the name *representation:Foo*.

3.5.6 Stopwatches

Stopwatches allow the measurement of the execution time of parts of the code. The statements the runtime of which should be measured have to be placed into the macro *STOP_TIME_ON_REQUEST(<id>, <statements>)* (declared in *Src/Tools/Debugging/Stopwatch.h*) as second parameter. The first parameter is a string used to identify the time measurement. To activate a certain time measurement, e. g., *Foo*, a debug request *stopwatch:Foo* has to be sent. The measured time can be seen in the timing view (cf. Sect. 7.3.7). 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. 7.5.3).

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

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

Chapter 4

Cognition

In the B-Human system, the process *Cognition* (cf. Sect. 3.1) can be structured into the three functional units *perception*, *modeling*, and *behavior control*. The major task of the perception modules is to detect landmarks such as goals, beacons, and field lines, as well as obstacles and 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, and the position and speed of the ball. Only these modules are able to provide useful information for the behavior control which is described separately (cf. Chapter 6). Figure 4.1 shows all modules running in the process *Cognition* and the representations they provide.

4.1 Perception

4.1.1 Basics

B-Human uses a grid-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 the second Y channel of each *YUV422* pixel pair and also ignoring every second row. The images are scanned on vertical and horizontal scan lines (cf. Fig. 4.4). Thereby, the actual amount of scanned pixels is much smaller than the image size because the grid has fewer pixels. The output generated by the perception modules is mainly a collection of base points of the landmarks on the field and of points that lie on field lines. Each of these percepts is stored in the *PointsPercept* and contains information about the point in the image, the point's coordinates on the field relative to the robot's camera and about the point type (field line, base point of the blue/yellow goal, base point of a beacon). Another important issue handled by the perception modules is providing the relative position of the ball on the field, the *BallPercept*.

As mentioned before, the idea behind the perception modules is not to perceive goals, beacons, or field lines directly, but to provide a set of base points for each landmark. All these percepts are used by the modeling modules, e. g. for self-localization and ball tracking.

4.1.2 Definition of Coordinate Systems

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

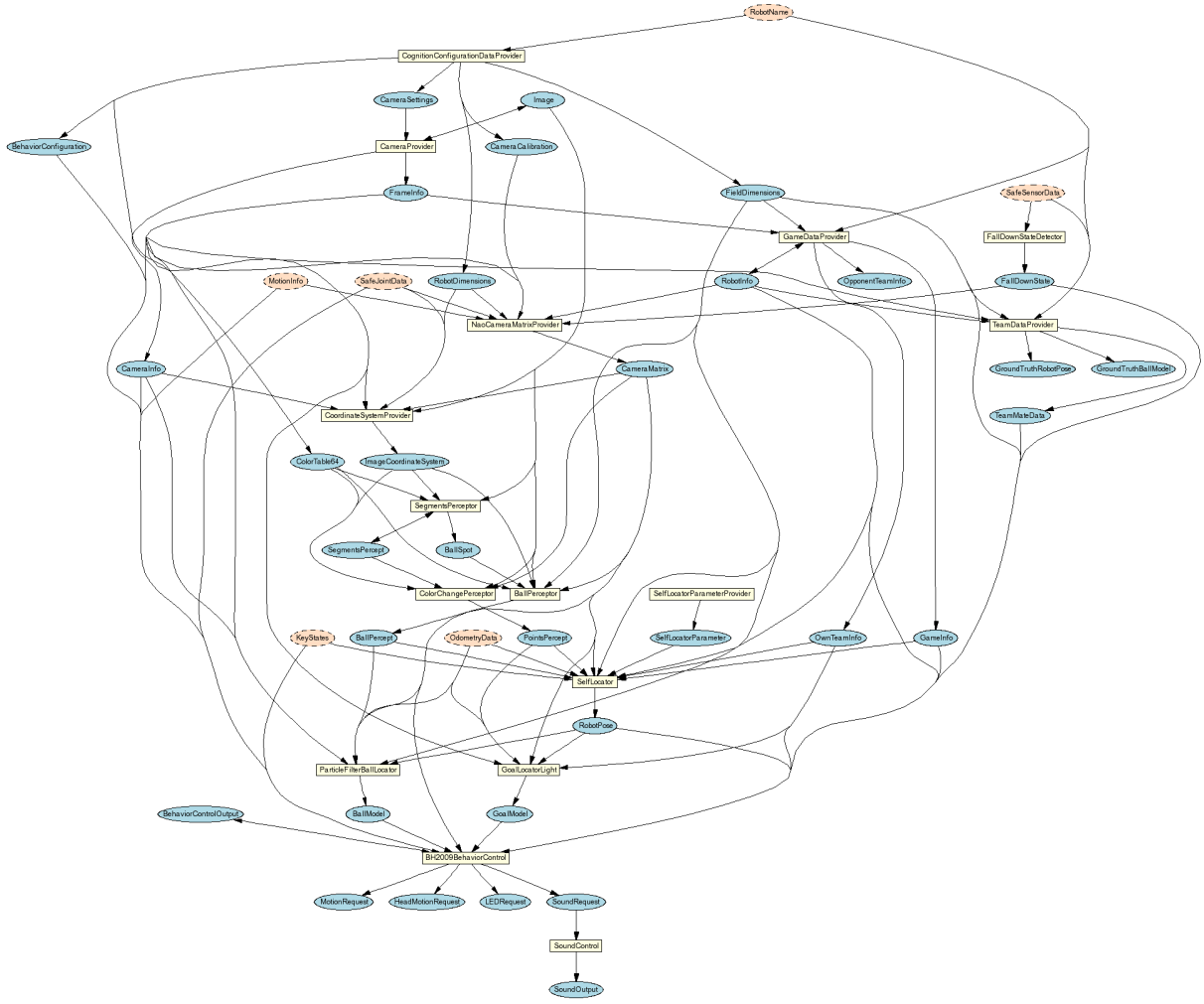

Figure 4.1: All modules and representations in the process *Cognition*

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.

4.1.3 Image Processing

The image processing is split into several different steps which are described in the following.

4.1.4 Color Segmentation

The SegmentsPerceptor works on the camera image and fills a data structure called *SegmentsPercept*. Its main task is to color-classify the image and to group neighboring pixels of similar colors into color regions. The idea behind this is to reduce noise and to simplify further image processing.

In a first step, the SegmentsPerceptor calculates the position of the horizon in the image by a 3-D camera equation using the current joint positions and camera attributes. After this, two scans are performed on every image: a horizontal one and a vertical one. The configuration file

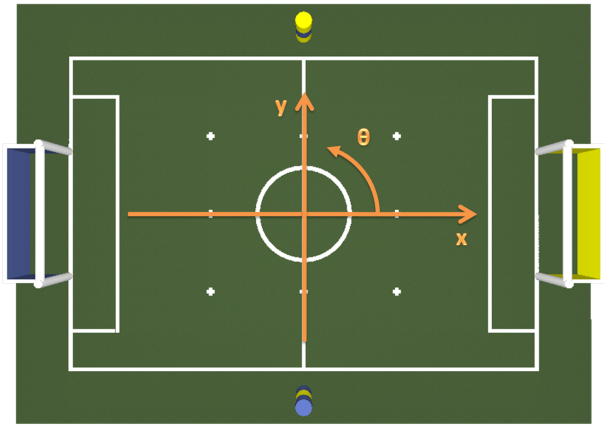

Figure 4.2: Visualization of the global coordinate system

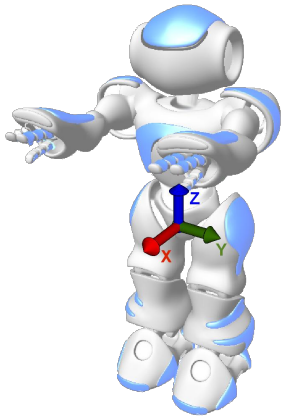

Figure 4.3: Visualization of the robot relative coordinate system

pointsSegments.cfg allows configuring some properties of this scan grid. To ensure not to ignore important information in the image, such as far balls, the grid is narrower near the horizon. On each scan line, the SegmentsPerceptor tries to determine segments of the same color with the largest possible length. To achieve this, every pixel in the image is classified by color using a manually configured color table. Only small gaps are allowed in single color sequences for being accepted as a continuous segment. For each scan, there is a list of the segments found. Each segment contains information about its color, its x or y-position in the image (depends on the orientation of the scan line) and its length.

Another task of the SegmentsPerceptor is to calculate a convex hull of the field outlines so that everything in the image that is not inside these field borders (audience, advertising boards, and so on) can be ignored.

4.1.5 Detecting Landmarks and Field Lines

The ColorChangePerceptor module uses the segments provided by the SegmentsPerceptor and tries to detect significant changes of color between two neighboring segments. Only those segments that lie within the field borders calculated by the SegmentsPerceptor are examined. For instance

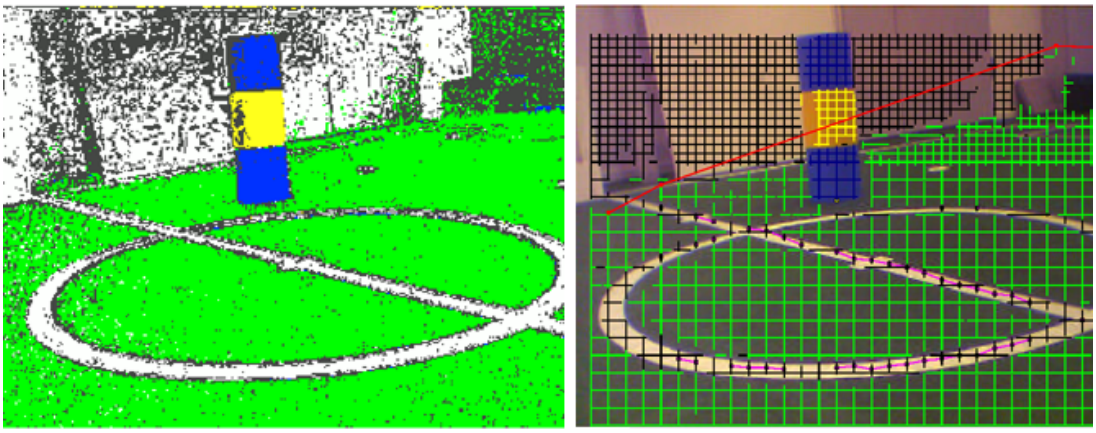

Figure 4.4: A completely segmented image and an image with its corresponding scan grid. The red lines denote the computed border of the field.

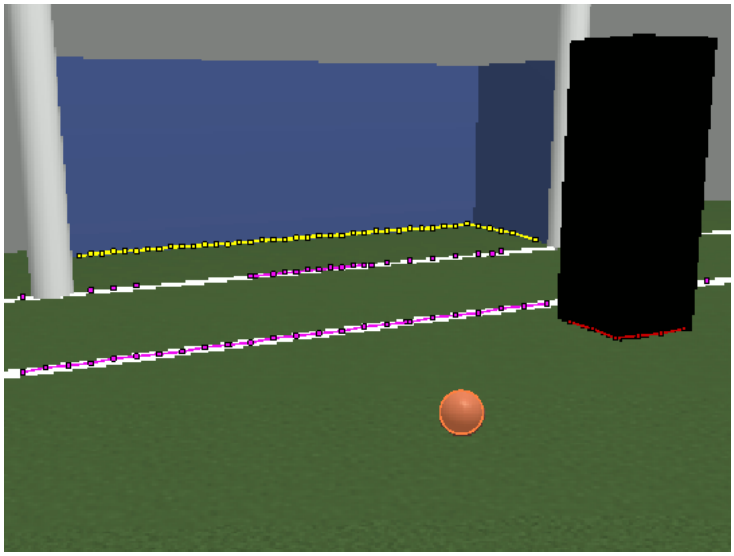

Figure 4.5: Simulator screenshot that shows the representation of landmarks and field lines as a set of 2-D points

a changeover from green to yellow on a vertical scan line could mean the existence of a goal base point at this position in the image. There are some more conditions which have to be fulfilled by neighboring segments in order to be accepted in the *PointsPercept*. Some examples for conditions that are checked are a maximum distance between the segments, minimal and maximal length, or more complex segment sequences (e. g. green-white-green for a field line or blue-yellow-blue for a beacon). If all these conditions are fulfilled for a special color change, a projection for this image point to field coordinates is done and a new entry is stored in the *PointsPercept*.

Special cases are base points of landmarks that lie somewhere below the bottom of a vertical scan line and are not visible. This might happen when the robot looks too high and only the upper part of a goal is visible in the image. In this case, an entry in the *PointsPercept* is nevertheless created for the pixel at the bottom of the scan line and the flag *isCloser* is set to indicate that the point might be closer to the robot than the coordinates calculated indicate.

The ColorChangePerceptor distinguishes the following point types:

- goals (yellow and blue)
- beacons (blue-yellow-blue and yellow-blue-yellow)
- black obstacles (obstacle challenge poles)
- white field lines

Currently, the B-Human cognition system does not support the detection of opponent robots.

4.1.6 Detecting the Ball

The BallPerceptor requires the representation *BallSpot*, which is provided by the module SegmentsPerceptor. The *BallSpot* is the center of the biggest orange segment in the image.

To find out if the orange cluster is actually the ball, the image is scanned in eight different directions (horizontal, vertical, and both diagonal directions) starting at the *BallSpot*. If the

border of the image is reached or the last eight points viewed are not orange, the search is stopped and the last orange point is saved. If the ball points found are considered as valid, it is tried to find the center and the radius of the ball using the Levenberg-Marquardt method [1]. First, only hard edge points, that means points with a high contrast that are next to a green point are passed to the method. If there are not enough of those points or if the method fails, all hard edge points are taken into account. If that does not work either, all points that are not at the image border are used. If that still does not work out, the last attempt is to use all ball points.

If the center of the ball is below the horizon, the offset to the ball is calculated based on its size in the image (for far balls) or its bearing (for close balls). The result is stored in the representation *BallPercept*, The *BallPercept* is used by the ParticleFilterBallLocator to provide the *BallModel*. For more information see 4.2.2.

4.2 Modeling

4.2.1 Self-Localization

For self-localization, B-Human uses a particle filter based on the Monte Carlo method [5] as it is a proven approach to provide accurate results in such an environment [16]. Additionally, it is able to deal with the kidnapped robot problem, which often occurs in RoboCup scenarios. For a faster reestablishment of a reasonable position estimate after a kidnapping, the *Augmented MCL* approach by [8] has been implemented. Figure 4.6 shows an example of a probability distribution. A comprehensive description of our state estimation implementation is given in [11].

The module providing the *RobotPose*, which is a simple pose in 2-D, is the SelfLocator. It has already shown a reasonable performance during the last two competitions in the Humanoid League. However, the implementation within the software release has been optimized for a Humanoid League field and shows only a poor performance on SPL fields. This results from its approach to take only points on the field (as described in 4.1.1) instead of more complex perceptions into account. By using a different goal design, the SPL field provides much less points which significantly contribute to a global localization. Therefore, for RoboCup 2008, the BreDoBrothers used a modified version of this module, which made use of the more complex perceptions available within the BreDoBrothers software.

4.2.2 Ball-Modeling

Estimating the ball's velocity as well as filtering its position is also realized via a particle filter similar to the one used for self-localization. A detailed description is also given in [11]. A probability distribution of a ball in motion is shown in Fig. 4.6. By using the perception described in Sect. 4.1.6, the module ParticleFilterBallLocator computes the *BallModel*.

This module has been used by the BreDoBrothers during RoboCup 2008.

4.2.3 Tracking the Opponent's Goal

Playing soccer is about scoring goals. Since the B-Human software does not include any recognition of other robots and we refuse to completely rely on self-localization when kicking, a model of an appropriate kicking corridor towards the opponent goal is needed. This model is named

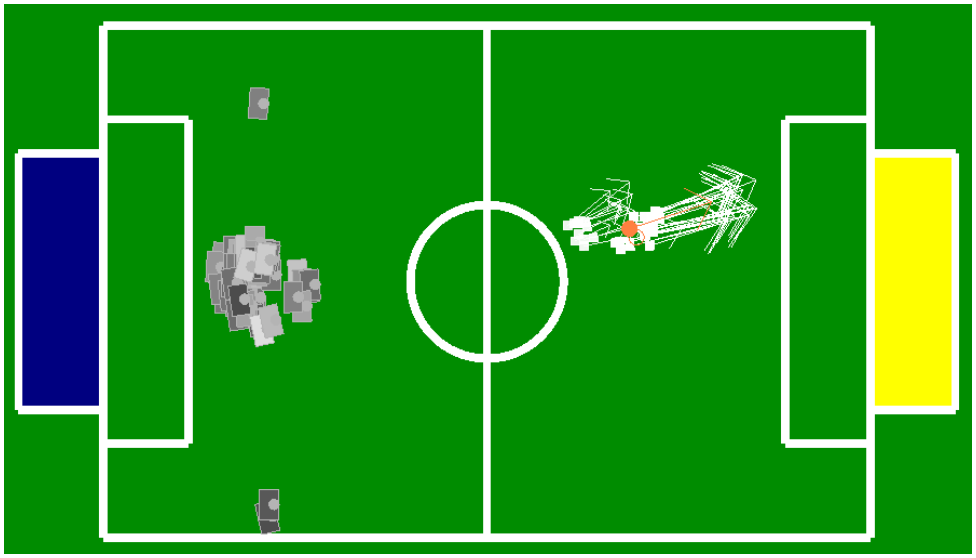

Figure 4.6: Illustration of particle-based representation of the robot pose and the ball position and velocity. Each gray box denotes one possible pose of the robot; light boxes are more likely than dark boxes. The black box shows the resulting robot pose from the filter. The dots with arrows describe the ball samples (describing position and velocity of a ball). The orange one is the resulting ball estimate.

GoalModel and is currently provided by the GoalLocatorLight. By clustering perceived segments of the goal over some seconds, the two largest free parts of the goal are estimated.

Please note that this module has been developed for the goals used in the Humanoid League – which are solid and completely colored inside – and will not provide any reasonable results for the current SPL goals.

4.2.4 Detecting a Fall

To start an appropriate get-up motion after a fall, the orientation of the robot's body needs to be computed from the measurements of the accelerometers. This task is realized by the FallDownStateDetector which provides the *FallDownState*. By averaging the noisy sensor data, the module robustly detects whether the robot is in an upright position, lying on its front, its back, its left, or its right side. The last two cases appear to be highly unlikely but have already happened to our humanoid robots during an official match.

4.2.5 Obstacle detection

In order to complete the Humaoind Kid-Size League's obstacle avoidance challenge, it is necessary to recognize black obstacles and to evade them. A representation called *FreeSpaceModel* splits up the environment in front of the robot into a fixed amount of segments. Each of these segments can either be free or blocked by an obstacle. If it is blocked, the segment contains information about the distance of the obstacle in this segment. The FreeSpaceModelProvider works on the percepts provided by the perception modules and generates this model. Furthermore, it calculates the best angle which the robot should walk to in order to avoid obstacles and approach the goal.

Chapter 5

Motion

The B-Human motion system generates all kinds of motions needed to play soccer with a robot. These are walking, kicking, standing up from the front and the back, looking in front of the feet, and standing. The walk and stand motions are generated dynamically by the WalkingEngine (cf. Sect. 5.1). All other motions are static motions, provided by the SpecialActions module (cf. Sect. 5.2). Both modules generate joint angles. The WalkingEngine provides the *WalkingEngineOutput* and the *WalkingEngineStandOutput*. The SpecialActions module provides the *SpecialActionsOutput*. According to the *MotionRequest* the MotionSelector (cf. Sect. 5.3) calculates which motions to execute and how to interpolate between different motions while switching from one to another. This information is provided in the *MotionSelection*. If necessary, both modules calculate their joint data and the MotionCombinator (cf. Sect. 5.3) combines these according to the *MotionSeletion*. The MotionCombinator provides the *JointRequest*, *OdometryData*, and *MotionInfo*. Figure 5.1 shows all modules running in the process *Motion* and the representations they provide.

5.1 Walking

The WalkingEngine is a module that creates walking motions based on a set of parameters, which are described in more detail later. In principle the resulting gait is based on fixed trajectories, which describe the foot position in dependency of the walking phase. The joint angles are calculated by an inverse kinematic model of the robot based on the desired foot positions.

The WalkingEngine requires the representation *MotionRequest* as input, which includes the *WalkRequest*. It consists of the desired speed in x- and y-direction as well as the rotation speed. Thus the WalkingEngine does not generate a static motion. Instead, it is able to produce walking motions for every incoming *WalkRequest*.

First of all the module checks whether the *WalkingParameters* have changed since the last executed frame; if this is the case the former and the new parameters are interpolated linearly. This is a nice feature when creating or optimizing a set of *WalkingParameters* manually, because otherwise the executed motion would change abruptly, which could overload the joints of the robot.

When the resulting *WalkingParameters* have been determined, the current *WalkRequest* is focused. Also in order not to produce abrupt changes of the motion, the resulting *WalkRequest* is calculated from the former and the current *WalkRequest*, where the maximal speed change for each direction (x, y, and rotation) is parameterized. Beforehand the walk speed is clipped to keep the generated trajectories within reasonable limits. The clipping is performed in a way

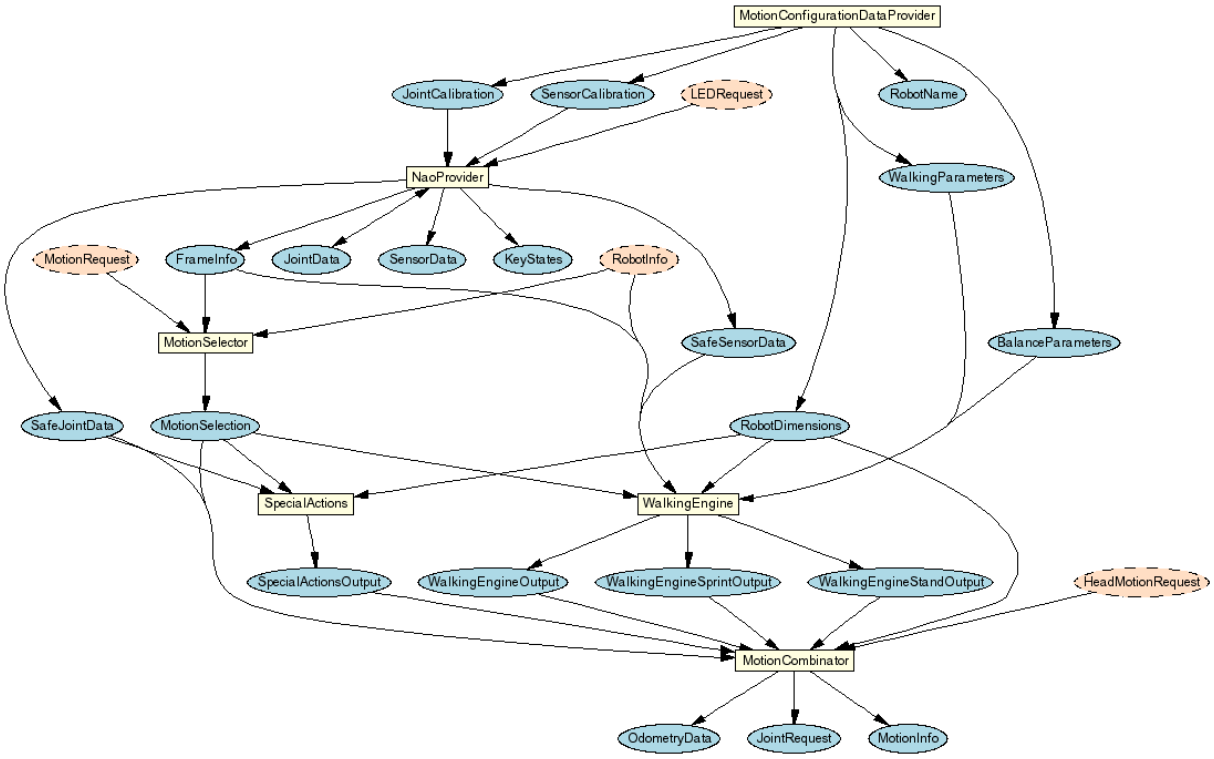

Figure 5.1: All modules and representations in the process *Motion*

that the ratio of x-, y- and rotation-speed is kept. From the resulting walk speeds the size of the trajectory is calculated (it only depends on the desired speed and the phase length).

The joints of the left and right arm are determined from the walking parameters and the arm trajectory (for the arms no inverse kinematic is used, the joints are set directly).

The body tilt and body shift are primarily determined from the according parameter. Additionally these temporary values are corrected using sensor data. The desired body shift is watched over three frames to determine the acceleration of the hip respectively the torso. If the measured sideways acceleration differs from the desired one, the body shift is corrected (if the corresponding parameters are not set to 0). From the foot trajectories and the other parameters and trajectories (body shift, foot rotation, etc.) the resulting foot position is computed. The inverse kinematic then calculates the joint angles of the legs and the feet. Furthermore it is possible to control the foot position or the resulting joint angles using sensor data.

5.1.1 Description of the WalkingParameters

stepDuration. The length of one step (consisting of two half steps of each leg) in milliseconds

maxSpeedChange. The largest possible speed change, parameterizable for each direction (x, y, and rotation), in $\frac{mm}{s^2}$ resp. $\frac{radian}{s^2}$.

odometryScale. A factor the calculated translational odometry is multiplied by. This parameter differs from 1 for the case that a specific walk request differs from the actual distance covered.

odometryRotationScale[C]CW. A factor the calculated rotational odometry is multiplied by. There are separate parameters for the clockwise and the counterclockwise direction,

because it is often the case that the robot is able to rotate faster in one of the directions.

max{Forward|Backward|Sideways|Rotation}Speed. The maximum speeds of the robots in different directions in $\frac{mm}{s}$ resp. $\frac{radian}{s}$ (higher values in a *WalkRequest* will be clipped maintaining the ratio)

elbowAngle. The fixed angle of the joints *armLeft1*, *armRight1*, *armLeft3* and *armRight3* in radian

armDeactivationRatio. The percentage of the deactivation of the arm (if a random number in [0..1] is below this value, the torque of the arm is set to zero)

bodyTiltOrigin. The fixed value for the body tilt (that means a rotation of the torso of the robot around the y-axis through the hip)

bodyTiltStandOrigin. The value for the body tilt while the robot is standing

bodyTiltSpeedFactor. The body tilt is multiplied by the current speed in x-direction and this factor; the robot leans forward the faster it is "running"

bodyTiltCorrection. The integrative part of a PID-controller for the body tilt (the average offset of the measured body tilt within the last 50 frames is multiplied by this factor and added to the body tilt)

bodyShiftCorrection. The factor for a P controller that controls sideways motion of the body to prevent the body from start rocking more and more violently

bodyShiftToAccel. The factor to transform the sideways body motion from the theoretical accelerations based on the commanded trajectory to the actual sideways acceleration

footPanCorrection. A value that is added constantly to the rotation trajectory in radian

footRollCorrection. The roll angle of the feet (rotation around the x-axis of the feet)

stepShape. The shape of the foot trajectories (for possible values and further description see *Src/Tools/Math/Trajectories.cpp*)

stepOrigin. Three-dimensional coordinates that represent the center of the foot trajectory (in millimeters) of the left foot (for the right foot the y-coordinate is negated)

stepTilt{Forward|Backward|Sideways}. To be ignored[1]

stepHeight. The height in millimeters each foot is lifted from the ground

stepPhases. The length of the ground phase, the lifting phase and the air phase of the 3-D foot-trajectory (the lowering phase is one minus the three other phases). This parameter is not needed by all trajectory shapes, see Section 5.1.2.2.

bodyShiftShape. The shape of the 1-D trajectory of the body shift. The so-called body shift is a value that is added to the y-position of both feet (hence the body is shifted relative to the feet)

bodyShiftOrigin. The origin of the body shift (if this parameter does not equal zero the robot walks and stands asymmetrically)

[1] These parameters are a relic from the walking engine for the AIBO.

bodyShiftScaleMode. The way the amplitude of the trajectory is scaled by the size (should always be *linear* as the size is hard-coded 1.0)

bodyShiftScale. The amplitude of the body shift in millimeters

bodyShiftPhaseShift. This value is added to the phase as input of the body shift trajectory relative to the foot trajectory

bodyShiftPhases. The length of the phases "going to the left", "staying left", "going to the right" of the 1-D body-shift-trajectory (the phase "staying right" results of the three other phases). This parameter is not needed by all trajectory shapes, see Section 5.1.2.1.

bodyShiftSidewaysOriginFactor{Left|Right}. The current y-speed is multiplied by this parameter and added to the body shift origin. This parameter causes that the robot leans to the side if he walks to the side.

rotationShape. The shape of the foot rotation trajectory

rotationOrigin. The origin of the foot rotation (if this parameter does not equal zero the feet are not oriented straight forward)

rotationScaleMode. The way the amplitude of the rotation trajectory is scaled by the rotation speed (should be linear, otherwise the rotation is not scaled correctly and does not react correctly on the rotation request)

rotationScale. The factor the rotation speed is multiplied by to get the amplitude of the rotation trajectory (this parameter should be near 1.0, otherwise the odometry probably would not fit to the resulting gait and has to be adapted)

rotationPhaseShift. The phase of the rotation trajectory is shifted by this value relative to the foot trajectory

rotationPhases. The length of the phases "rotating left", "staying left", "rotating right" of the 1-D rotation-trajectory (the phase "staying right" results of the three other phases). This parameter is not needed by all trajectory shapes, see Section 5.1.2.1.

armShape. The shape of the trajectory for the joints *armLeft0* and *armRight0*

armOrigin. The origin of the arm trajectory

armScaleMode. The way the amplitude of the arm trajectory is scaled by. This can be useful to let the arm swing to one direction only.

armScale. The factor the amplitude of the arm trajectory is multiplied by (in radian)

armPhaseShift. The phase of the arm trajectory is shifted by this value relative to the foot trajectory

armPhases. The length of the phases "swing forward", "staying forward", "swing backward" of the 1-D rotation-trajectory (the phase "stay backward" results of the three other phases) This parameter is not needed by all trajectory shapes, see Section 5.1.2.1.

5.1.2 Trajectories

5.1.2.1 1-D Trajectories

The 1-D trajectories are used in different parts (*bodyShift*, rotation, etc.) of the walking engine. All of them are strictly periodical, which means $amplitude(0) = amplitude(1.0)$. Additionally the function is differentiable at $\varphi = 0$ and $\varphi = 1$ (with exception of *halfSine*). Most of the trajectories are not parameterizable, those trajectories which *are* parameterizable are marked with vertical lines showing the variable sections of the trajectory. In the trajectories *rectangle*, *sineRectangle*, and *airSine* the section is linearly stretched. The trajectory *constAccel* computes the shape of the parabolas and the linear part between them, keeping the gradient at the transition constant. Hence the function within the variable ranges is not just stretched.

It is recommended to use the trajectories *constAccel* and *sineRectangle* for two reasons: they can be parameterized very exactly in contrast to most other shapes and the absolute value of the second derivation (acceleration) of them is quite small to reduce mechanical stress.

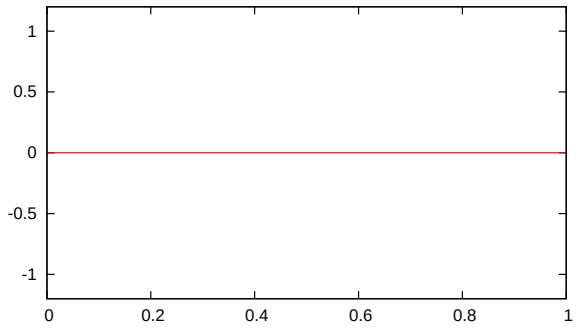

Figure 5.2: 1-D trajectory *constant*

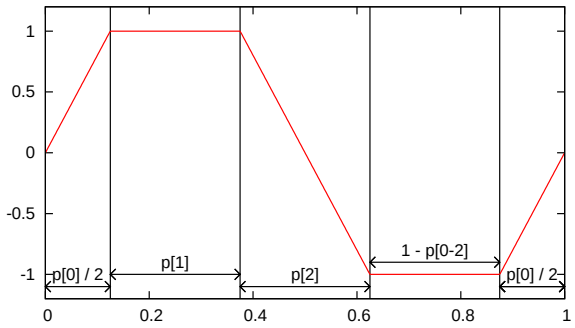

Figure 5.3: 1-D trajectory *rectangle*

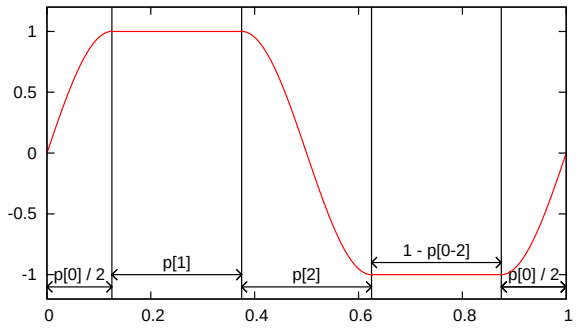

Figure 5.4: 1-D trajectory *sineRectangle*

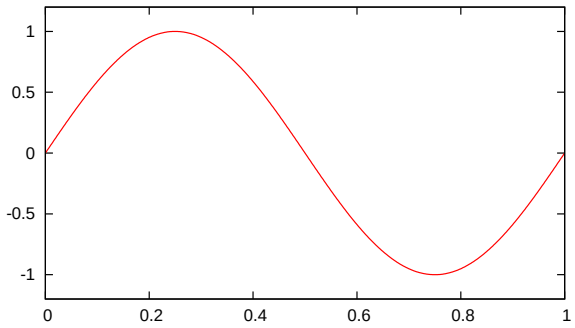

Figure 5.5: 1-D trajectory *sine*

5.1.2.2 3-D Trajectories

All 3-D trajectories produce a *Vector3* as output representing three-dimensional coordinates. The x- and y-coordinates depend on each other, that means internally only two amplitudes are computed from the phase, one of them is used for the x- and y-coordinates (determined from *xSize* and *ySize*) and the other is only used for the z-coordinate. Therefore the shape of the different trajectories is visualized by a 2-D plot, where the x-axis shows the amplitude used for the x- and y-coordinates and the y-axis shows the amplitude used for the z-coordinate.

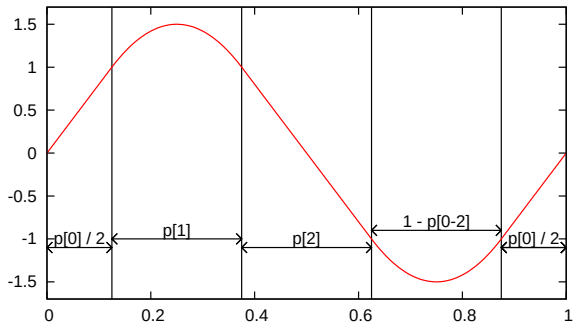

Figure 5.6: 1-D trajectory *constAccel*

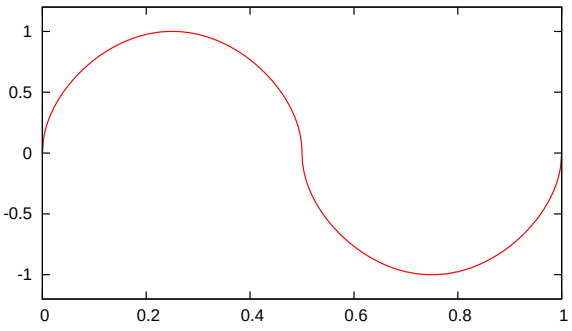

Figure 5.7: 1-D trajectory *sineSqrt*

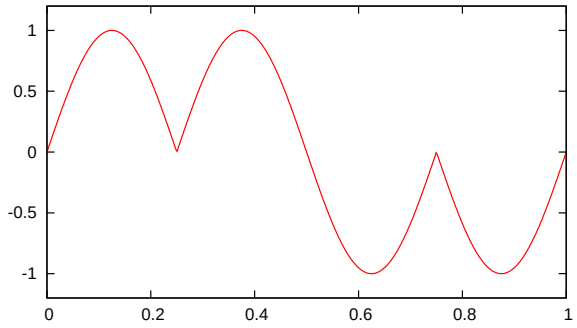

Figure 5.8: 1-D trajectory *doubleSine*

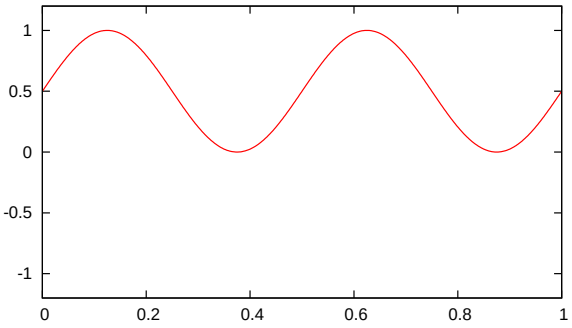

Figure 5.9: 1-D trajectory *doubleSinePositive*

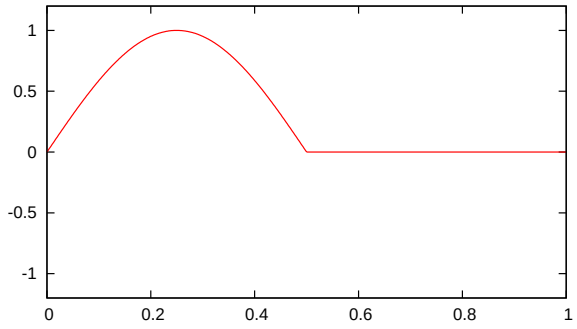

Figure 5.10: 1-D trajectory *halfSine*

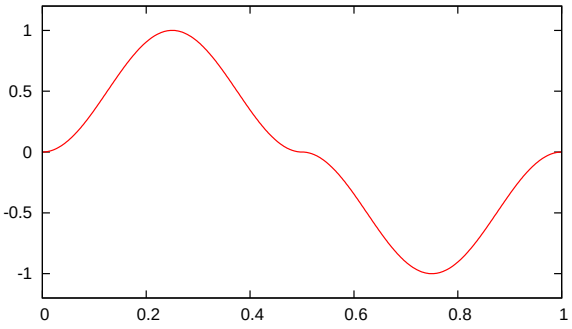

Figure 5.11: 1-D trajectory *flatSine*

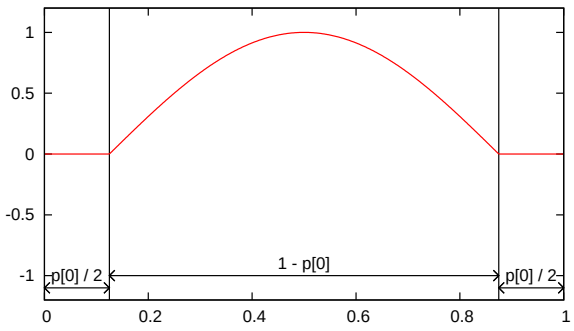

Figure 5.12: 1-D trajectory *airSine*

As already stated above not all trajectory shapes use the parameter phases, respectively not all of them. The trajectory *ellipse* (cf. Fig. 5.16) does not use any of the phase lengths, it just produces a harmonical ellipse. Another characteristic of the ellipse is that the minimum of the amplitude is not 0 but -1 and that there is no real ground phase. The trajectories *halfEllipse* (cf. Fig. 5.17) and *inverseHalfEllipse* (cf. Fig. 5.18) only take into account the length of the ground phase, the arc in case of *halfEllipse* and the straight line in case of the *inverseHalfEllipse* are computed as the difference between 1 and the ground phase. The other three shapes take all phase lengths into account. The fourth phase is computed as difference between the sum of the three given phases and 1. For the shape *rectangle* (cf. Fig. 5.13) the phase lengths just parameterize the time needed for each of the edges of the rectangle. The trajectory *trapezoid* (cf. Fig. 5.14) takes the first phase length for the ground time, the times for rising, and holding up (the remainder is for lowering). By setting the third phase to zero also a triangle can be formed. The trajectory *constAccel* (cf. Fig. 5.15) is just a combination of two 1-D trajectories: *constAccel* for the x- and y-coordinates and *sineRectangle* for the z-coordinate. The trajectory *sineRectangle* is phase-shifted, so that the center of the ground phase is reached at $\varphi = 0$. In contrast to the other trajectories the absolute value of the x/y-amplitude of *constAccel* exceeds 1.0, only the ground phase is guaranteed to be in the range $[-1.0 \ldots 1.0]$.

The point at $\varphi = 0$ of each trajectory shape is the lowest point with x-coordinate $= 0$.

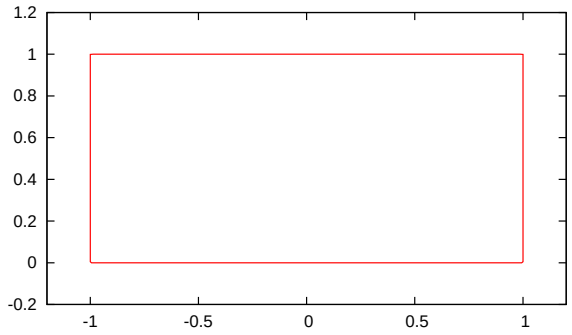

Figure 5.13: 3-D trajectory *rectangle*

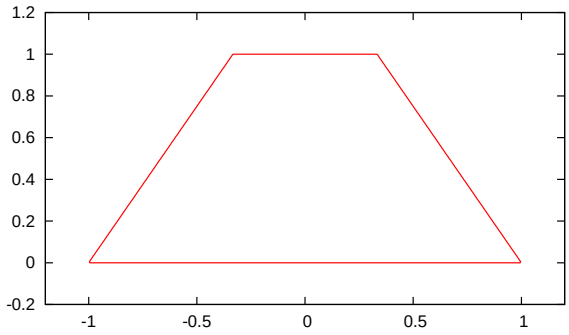

Figure 5.14: 3-D trajectory *trapezoid*

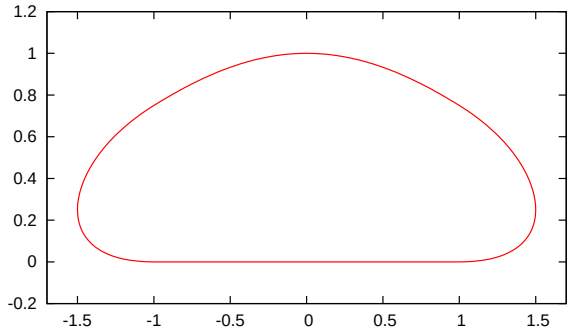

Figure 5.15: 3-D trajectory *constAccel*

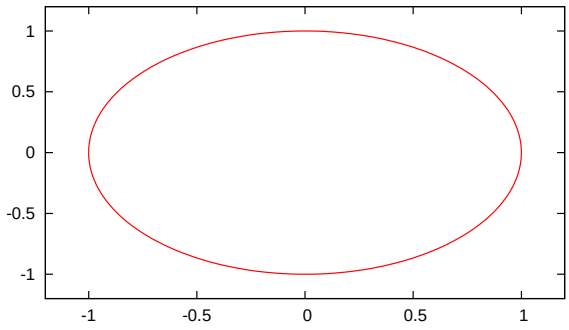

Figure 5.16: 3-D trajectory *ellipse*

5.2 Special Actions

Special actions are hardcoded motions that are provided by the module SpecialActions. By executing a special action, different target joint values are sent consecutively, allowing the robot

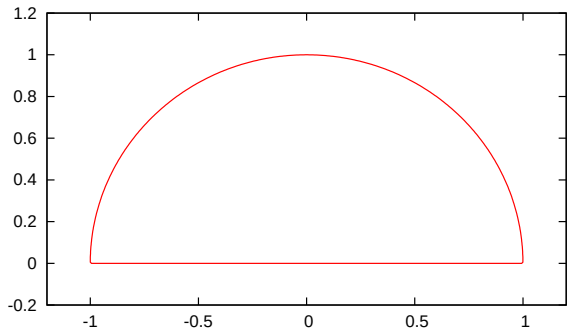

Figure 5.17: 3-D trajectory *halfEllipse*

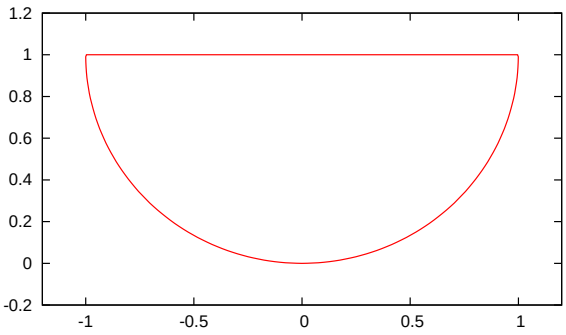

Figure 5.18: 3-D trajectory *inverseHalfEllipse*

to perform actions such as kicking, standing up, or searching for the ball near its feet. 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*. Then, each line represents a set 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[2]/roll/pitch, knee pitch, ankle pitch/roll). A * does not change the angle of the joint (keeping, e. g., the joint angles set by the head motion control), a – deactivates the joint. Each line ends with two more values. The first decides if the target angles will be set immediately (the value is 0), forcing the robot to move its joints as fast as possible, or if 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.

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

An example of a special action:

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

To receive proper odometry data for special actions, they have to be manually set in the file *Config/odometry.cfg*. It can be specified whether the robot moves at all during the execution of the special action, and if yes, how it has moved after completing the special action, or whether

[2]Ignored

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.

5.3 Motion Combination

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

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

If the *MotionRequest* requires switching from one motion to another, the MotionSelector calculates influence ratios for the motion currently executed and the target motion. These ratios are required by the MotionCombinator to merge the joint angles generated by the WalkingEngine and the SpecialActions module together. The ratios are interpolated linearly. The interpolation time between different motions depends on the requested target motion.

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

Chapter 6

Behavior Control

The part of the B-Human system that makes the decisions is called *Behavior Control*. The behavior was modeled using the Extensible Agent Behavior Specification Language (XABSL) [14]. The module provides the representations *MotionRequest*, *HeadMotionRequest*, *LEDRequest* and *SoundRequest*.

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

6.1 XABSL

XABSL is a programming language that is designed to model an agent behavior. To work with it, it is important to know its general structure. In XABSL following base elements are used: *options*, *states*, *decisions*, *input symbols*, and *output symbols*. A behavior consists of options that are arranged in an option graph (cf. Fig. 6.1 and Fig. 6.2). There is a single option to start the whole behavior from which all other options are called; this is the root of the option graph. Each option describes a specific part of the behavior such as a skill or a head motion of the robot, or it combines such basic features. For this description each option consists of several states. Each option starts with its *initial state*. Inside a state, an action can be executed and optionally a decision can be made. An action can consist either of the modification of output symbols (for example head motion requests or walk requests), or a call of another option. A decision comprises conditions and transitions the latter of which are changes of the current state within the same option.

This structure is clarified with an example:

```
option example_option
{
  initial state first_state
  {
    decision
    {
      if(boolean_expression)
        goto second_state;
      else if(boolean_expression)
        goto third_state;
      else
        stay;
```

```
    }
    action
    {
      output_symbol = input_symbol * 3
    }
  }

  state second_state
  {
    action
    {
      secondOption();
    }
  }

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

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

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

```
      output_symbol = input_symbol * 3
    }
  }

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

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

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

option example_suboption
{

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

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

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

```
option example_superior_option
```

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

option example_suboption
{

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

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

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

6.2 Setting up a new Behavior

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

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

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

6.3 Playing Soccer

The root option of the soccer behavior is *start_soccer*. In there the options *head_control* and *body_control* are invoked. *head_control* is responsible for the head motion of the robot. Depending on the chosen head motion by the role, the corresponding option is executed. The task of the option *body_control* is to start the whole behavior. Within this option, the following states are available, following the rules of the Standard Platform League 2008 [3]:

- state_initial
- state_ready
- state_set
- state_playing
- state_penalized
- state_finished

These states can be set either by the GameController or by pressing the chest button[1]. At the beginning the *initial_state* is executed. Within this state the robot executes the special action

[1]Currently the penalty state can only be set by the game controller and not by pressing the chest button.

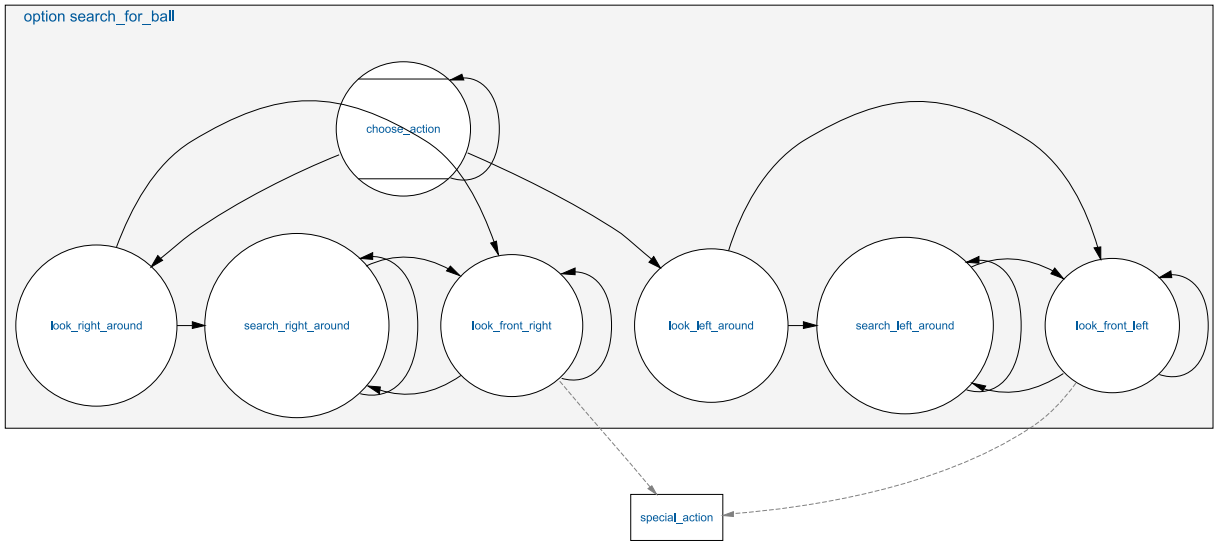

Figure 6.1: The option *search_for_ball*

stand and looks straight forward. By switching to the *ready_state* the robot begins walking to a its kick-off position. This point diversifies according to the roles of the robots and whether the own team has kickoff or not. During walking to the kick-off position the robot looks around to ensure a good localization. After it has reached its goal position the state is changed to the *set_state*. Within this state the robot executes the special action *stand* again and searches for the ball, and if it has found it, it will continuously look at it.

After switching to the *playing_state*, the robot starts playing according to its defined role. It is possible to stop the game by switching to the state *finished*. Within this state the robot stands looking straight ahead.

It is also possible to stop a certain robot by sending it to the penalized state in which the robot stands and looks down. The only role that is already implemented for the Nao is the striker. The task of the striker is to find the ball, approach it, aim at the opponent goal, and kick. Corresponding to this situation also the body control is only implemented for the striker role.

6.3.1 Searching for the Ball

If the robot has not seen the ball for more than two seconds, it starts searching for the ball. This is done by the option *search_for_ball* (cf. Fig. 6.1). At first, the robot decides whether to search on its right or left side, depending on the last known ball position. If the robot was close to the ball when it was seen the last time, it first looks down to check if the ball is in front of its feet. If the ball is not there the robot starts rotating to the side where the ball was seen last. Every 7500 ms the robot checks, whether the ball is in front of its feet and, if that was not the case, goes on with the search.

6.3.2 Approaching the Ball

As soon as the ball is seen, the option *go_to_ball* is executed (cf. Fig. 6.2). At first is decided whether the robot can approach the ball from behind or needs to go around the ball to aim at the opponent goal. In the first case the robot directly starts approaching the ball. In the second case the robot determines on which side to pass the ball, depending on its angle to the ball and

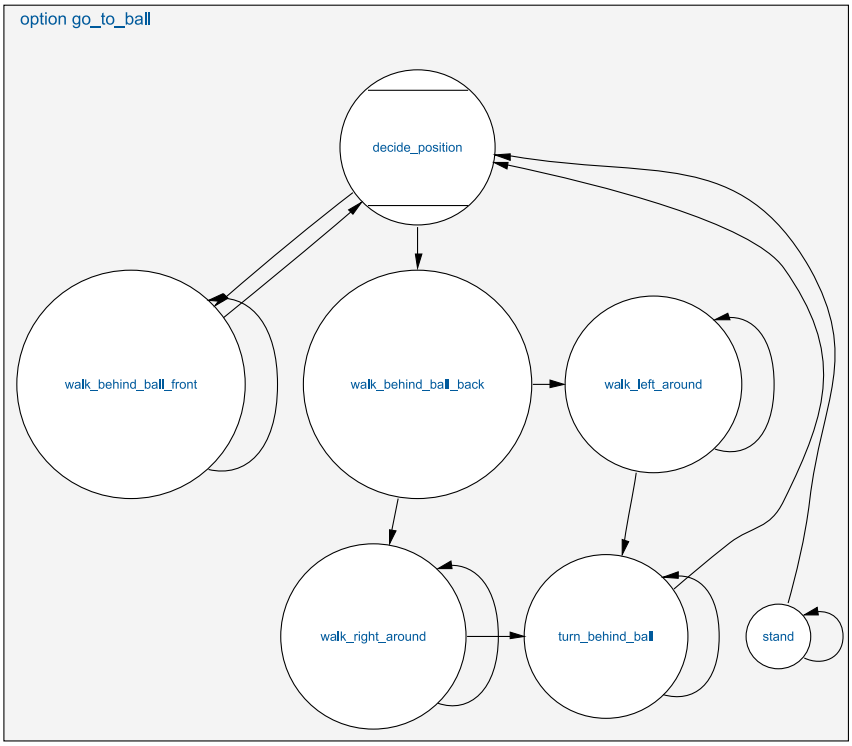

Figure 6.2: The option *go_to_ball*

on the angle to the goal. The last step is turning around after the ball was passed and getting into a good position to kick.

6.3.3 Kicking

After the robot reached a good position, meaning the ball is close enough to kick it and the robot is aiming at the opponent goal, the option *kick* is executed (cf. Fig. 6.3). In this option it is checked whether to kick with the right or with the left foot, depending on the position of the ball. If necessary the robot carries out some corrections on its position. As soon as it is ready to kick the special action *kick_left_nao* is executed. To find out more about special actions see Section 5.2.

6.4 Humanoid League's Obstacle Challenge

The obstacle avoidance behavior is based on a very simple idea: It just realizes a walk into the walking direction given by the FreeSpaceModelProvider and executes a head motion that tries to look at the base points of obstacles in front. Some special handling is only required when the direction to the goal is lost (e. g. when the view at the goal is completely blocked by obstacles), or when the robot stands directly in front of an obstacle. In the first case the robot will try to use self-localization to turn to the goal, in the other case it will explicitly turn away from the obstacle.

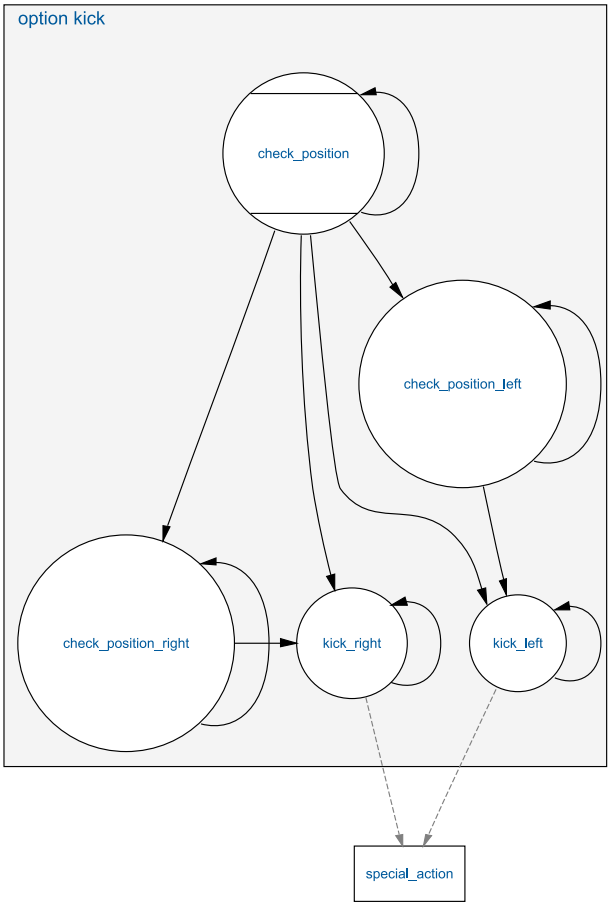

Figure 6.3: The option *kick*

Chapter 7

SimRobot

7.1 Introduction

The B-Human software package uses the physical robotics simulator SimRobot [13, 12] 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.

7.2 Scene View

The scene view (cf. Fig. 7.1 right) appears if the *scene* is opened from the tree view. The view can be rotated around two axes, and it supports several mouse operations:

- Left-clicking an object allows dragging it to another position. Robots and the ball can be moved in that way.
- Left-clicking while pressing the *Shift* key allows rotating objects around their body centers.
- Select an *active* robot by double-clicking it. Robots are active if they are defined in the compound *robots* in the scene description file (cf. Sect. 7.4).

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

7.3 Information Views

In the simulator, *information views* are used to display debugging output such as debug drawings. Such output is generated by the robot control program, and it is sent to the simulator via *message queues*. The views are interactively created using the console window, or they are defined in a script file. Since SimRobot is able to simulate more than a single robot, the views are instantiated separately for each robot. There are eight kinds of views related to information received from robots: *image views*, *color space views*, *field views*, the *Xabsl view*, the *sensor data view*, *plot views*, the *timing view*, and *module views*. 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. All information views can be selected from the tree view (cf. Fig. 7.1 left).

Figure 7.1: SimRobot with the tree view on the left and two scene views on the right. The console window is shown at the bottom.

7.3.1 Image Views

An image view (cf. left of Fig. 7.2) displays information in the system of coordinates of the camera image. It is defined by giving it a name and a background image using the console command *vi* and by adding debug drawings to the view using the command *vid* (cf. Sect. 7.5.3).

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

```
vi image raw
vid raw representation:PointsPercept:Image
vid raw representation:BallPercept:Image
```

If color table editing is activated, image views will react to the following mouse commands (cf. commands *ct on* and *ct off* in Sect. 7.5.3):

Left mouse button. The color of the pixel or region selected is added to the currently selected color class. Depending on the current configuration, the neighboring pixels may also be taken into account and a larger cube may be changed in the color table (cf. commands *ct imageRadius* and *ct colorSpaceRadius* in Sect. 7.5.3). However if a region is selected, the *imageRadius* is ignored.

Left mouse button + Shift. If only a single pixel is selected, the color class of that pixel is chosen as the current color class. It is a shortcut for *ct* <*color*> (cf. Sect. 7.5.3). If a region is selected, all colors of pixels in that region that are not already assigned to a color class are assigned to the selected color class. Thus all colors in a certain region can be assigned to a color class without destroying any previous assignments.

Left mouse button + Ctrl. Undoes the previous action. Currently, up to ten steps can be undone. All commands that modify the color table can be undone, including, e. g., *ct clear* and *ct load* (cf. Sect. 7.5.3).

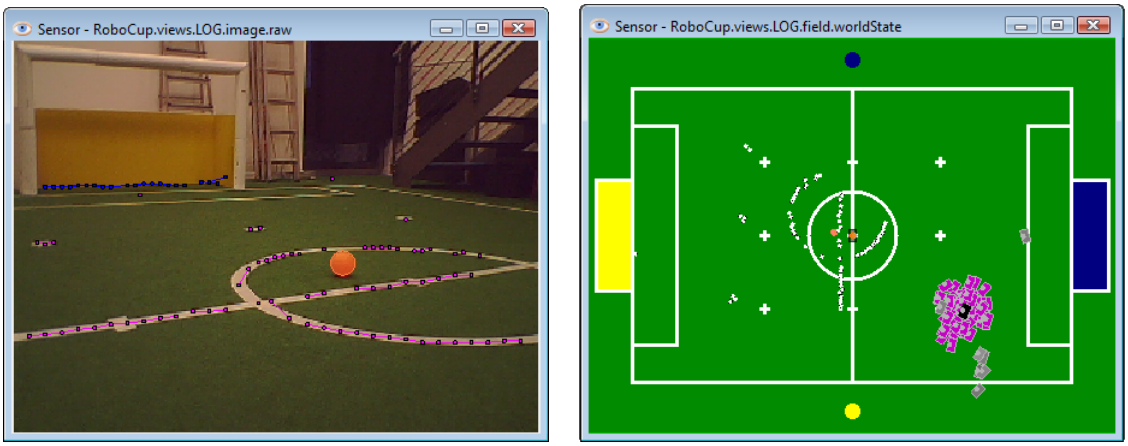

Figure 7.2: Image view and field view with several debug drawings

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. 7.5.3). However if a region is selected, the *imageRadius* is ignored.

7.3.2 Color Space Views

Color space views visualize image information in 3-D (cf. Fig. 7.3). They can be rotated by clicking into them with the left mouse button and dragging the mouse afterwards. There are three kinds of color space views:

Color Table. This view displays the current color table in YCbCr space. Each entry that is assigned to a certain color class is displayed in a prototypical color. The view is useful while editing color tables (cf. Fig. 7.3 down right).

Image Color Space. This view displays the distribution of all pixels of an image in a certain color space (*HSI*, *RGB*, *TSL*, or *YCbCr*). It can be displayed by selecting the entry *all* for a certain color space in the tree view (cf. Fig. 7.3 top right).

Image Color Channel. This view displays an image while using a certain color channel as height information (cf. Fig. 7.3 left).

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

```
v3 image raw
```

7.3.3 Field Views

A field view (cf. right of Fig. 7.2) displays information in the system of coordinates of the soccer field. It is defined similar to image views. For instance, the view *worldState* is defined as:

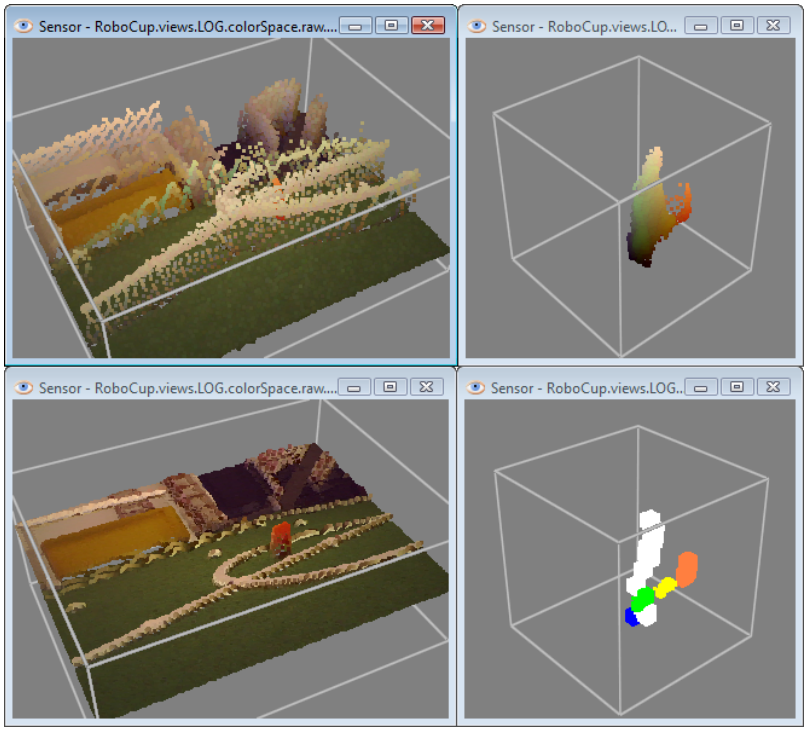

Figure 7.3: Color channel views, image color space view, and color table view

```
vf worldState
vfd worldState fieldPolygons
vfd worldState fieldLines
vfd worldState module:ParticleFilterSelfLocator: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:PointsPercept:Field
vfd worldState representation:GoalModel:largestFreePart
vfd worldState representation:MotionRequest

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

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

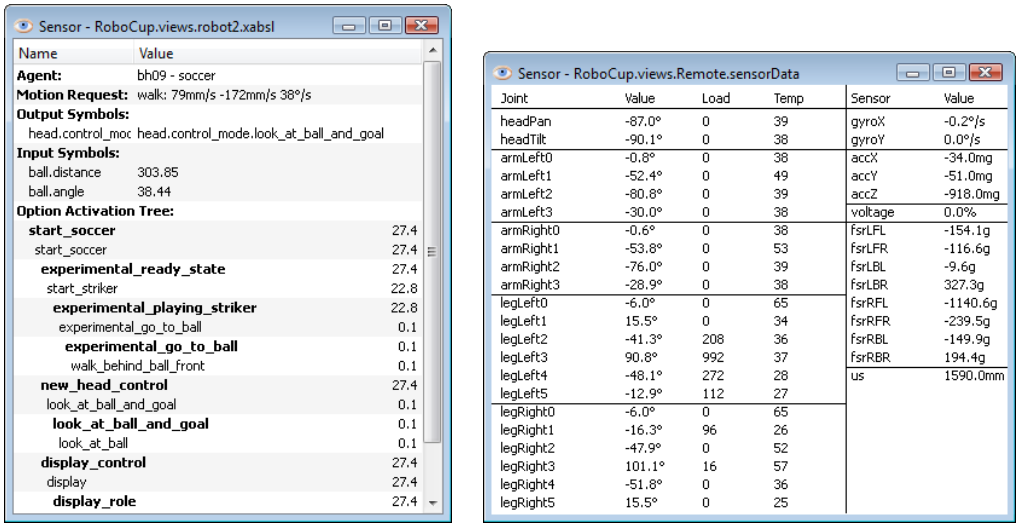

Figure 7.4: Xabsl view and sensor data view

7.3.4 Xabsl View

The Xabsl view is part of the set of views of each robot. It displays information about the robot behavior currently executed. In addition, two debug requests have to be sent (cf. Sect. 7.5.3):

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

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

The view can also display the current values of input symbols and output symbols. The symbols to display are selected using the console commands *xis* and *xos* (cf. Sect. 7.5.3).

7.3.5 Sensor Data View

The sensor data view displays all the sensor data taken by the robot, e. g. all joint angles, accelerations, gyro measurements, pressure readings, and sonar readings (cf. right view in Fig. 7.4). To display this information, the following debug requests must be sent:

```
dr representation:JointData
dr representation:SensorData
```

7.3.6 Plot Views

Plot views allow plotting data sent from the robot control program through the macro *PLOT* (cf. Fig. 7.5 left). They keep a history of the values sent of a defined size. Several plots can be displayed in the same plot view in different colors. Please note that currently plot views can only display integer values. Although the macro *PLOT* also accepts floating point values, they will always be rounded to the nearest integer. Hence, small floating values (e. g. radians) have to be scaled to a bigger range of values (e. g. degrees) to be plotted. A plot view is defined

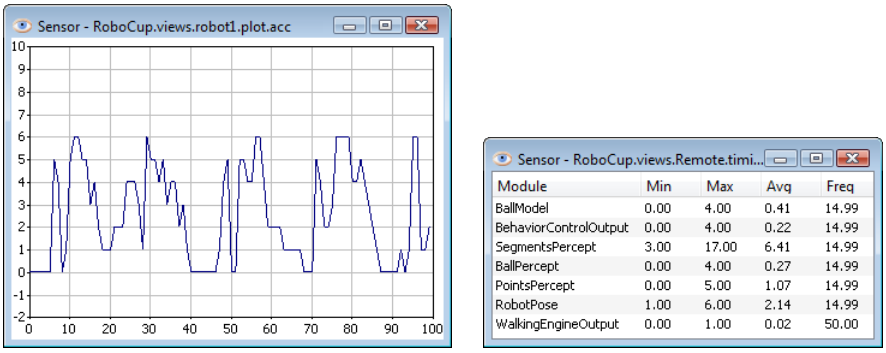

Figure 7.5: The plot view and the timing view

by giving it a name using the console command *vp* and by adding plots to the view using the command *vpd* (cf. Sect. 7.5.3).

For instance, the view on the left side of Figure 7.5 was defined as:

```
vp acc 100 -2 10
vpd acc accX blue
```

7.3.7 Timing View

The timing view displays statistics about the speed of certain modules (cf. Fig. 7.5 right). It shows the minimum, maximum, and average runtime of the execution of a module in milliseconds. In addition, the frequency is displayed with which the module was executed. 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.

7.3.8 Module Views

Since all the information about the current module configuration can be requested from the robot control program, it is possible to automatically generate a visual representation. The graphs such as the one that is shown in Figure 7.6 are generated by the program *dot* from the *Graphviz* package [6] 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 can either be displayed ungrouped or grouped by 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 one for the process *Motion*, both grouped and ungrouped.

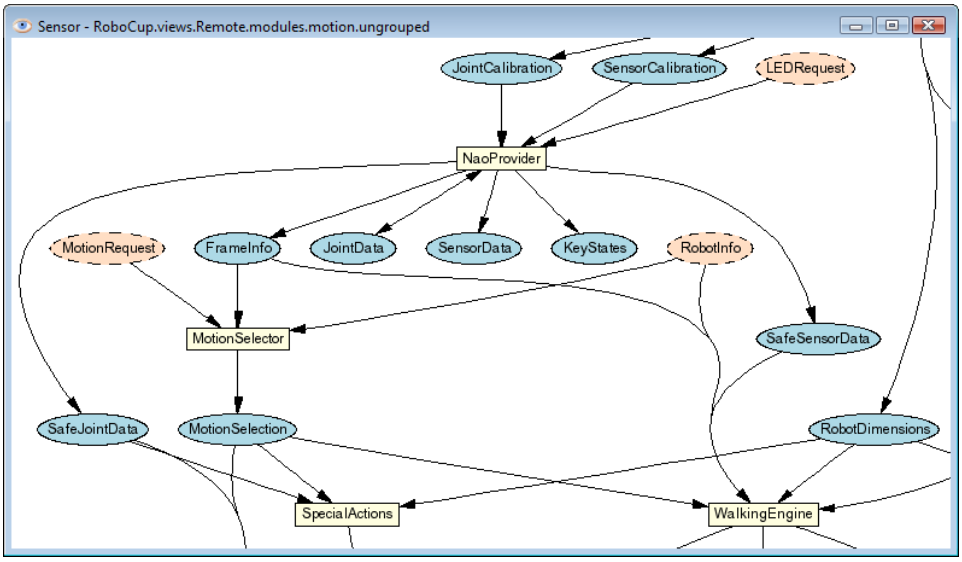

Figure 7.6: The module view showing part of the modules in the process *Motion*

7.4 Scene Description Files

The language of scene description files is RoSiML [7]. In the main *.ros* files, such as *BH2009.ros*, three compounds can be edited:

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

<Compound name="extras">. Below the compound *robots*, there is the compound *extras*. It contains *passive* robots, i. e. robots that just stand around, but that are not controlled by a program. Passive robots can be activated by moving their definition to the compound *robots*.

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

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

7.5 Console Commands

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

7.5.1 Initialization Commands

gt <name> <id>. Starts the ground truth module. The module uses a ceiling camera connected via FireWire (IEEE 1394) together with special markers mounted on top of the robots to determine the global position of the robots and the ball. It is based on the vision software of the B-Smart Small Size League team [2]. This information is distributed in the UDP team communication, so each robot knows its global position and the position of the ball. This information can be used for development and documentation. The parameter *name* defines the name used to identify the module in the section *views* of the tree view and in the command *robot* (see below). The *id* specifies the number of the FireWire camera used. An arbitrary number of cameras can be attached by instantiating a ground truth module for each camera. It is required that the command *tc* (see below) is executed before this one.

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 tree view. When the simulation is reset or the simulator is exited, the connection will be terminated.

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

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

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

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

7.5.2 Global Commands

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

cls. Clears the console window.

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

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

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

ro (stopwatches | sensorData | robotHealth | motionRequest) (off | on). Set a release option sent by team communication. The *release options* allow sending commands to a robot running the actual game code. They are used to toggle switches that decide which additional information is broadcasted by the robot in the team communication. *stopwatches* activates all time measurements in the process *Cognition*. 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. It 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.

7.5.3 Robot Commands

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

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

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

on | **off.** Activates or deactivates mouse handling in image views. If activated, clicking into an image view modifies the color table (cf. Sect. 7.3.1). Otherwise, mouse clicks are ignored.

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

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

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

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

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

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

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

smart [**off**]**.** Activates the smart color selection mechanism or deactivates it. The smart mode only affects the behavior when selecting a region of the image by mouse. If it 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 smart mode is on by default.

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

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

log ? | start | stop | pause | forward | backward | repeat | goto <number> | clear | (keep | remove) <message> {<message>} | (load | save | saveImages [raw]) <file> | cycle | once. The command supports both recording and replaying log files. The latter is only possible if the current set of robot processes was created using the initialization command *sl* (cf. Sect. 7.5.1). The different parameters have the following meaning:

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

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

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

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

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

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

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

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

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

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

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

poll. Poll 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**>**.** Send queue fill request. This request defines the mode how the message queue from the debug process to the PC is handled.

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

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

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

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

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

set ? [<**pattern**>] | <**key**> **(?** | **unchanged** | <**data**> **).** Change debug data or show 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**>]**.** Add a set of 3-D color space views for a certain image (cf. Sect. 7.3.2). The image can either be the camera image (simply specify *image*) or a debug image. It will be JPEG compressed if the option *jpeg* is specified. The last parameter is the name that will be given to the set of views. If the name is not given, it will be the same as the name of the image. A question mark followed by an optional filter pattern will list all available images.

vf <name>. Add field view (cf. Sect. 7.3.3). A field view is the means for displaying debug drawings in field coordinates. The parameter defines the *name* of the view.

vfd ? [<**pattern**>] | <**name**> **(?** [<**pattern**>] | <**drawing**> **(on** | **off)).** (De)activate 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**>]**.** Add an image view (cf. Sect. 7.3.1). An image view is the means for displaying debug drawings in image coordinates. The image can either be the camera image (simply specify *image*), a debug image, or no image at all (*none*). It will be JPEG-compressed if the option *jpeg* is specified. If *segmented* is given, the image will be segmented using the current color table. The last parameter is the name that will be given to the set of views. If the name is not given, it will be the same as the name of the image plus the word *Segmented* if it should be segmented. A question mark followed by an optional filter pattern will list all available images.

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

vp <**name**> <**numOfValues**> <**minValue**> <**maxValue**>**.** Add a plot view (cf. Sect. 7.3.6). 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.

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

xbb ? [<**pattern**>] | **unchanged** | <**behavior**> {<**num**>}**.** Selects a Xabsl basic behavior. The command suppresses the basic behavior currently selected by the Xabsl engine and replaces it with the behavior specified by this command. Type *xbb ?* to list all available Xabsl basic behaviors. The resulting list can be shortened by specifying a search pattern after the question mark. Basic behaviors can be parameterized by a list of decimal numbers. Use *xbb unchanged* to switch back to the basic behavior currently selected by the Xabsl engine. The command *xbb* only works if the Xabsl symbols have been requested from the robot (cf. Sect. 7.3.4). Note that basic behaviors are not used anymore in the B-Human code.

xis ? [<**pattern**>] | <**inputSymbol**> **(on** | **off).** Switches the visualization of a Xabsl input symbol in the Xabsl view on or off. Type *xis ?* to list all available Xabsl input symbols. The resulting list can be shortened by specifying a search pattern after the question mark. The command *xis* only works if the Xabsl symbols have been requested from the robot (cf. Sect. 7.3.4).

xo ? [<**pattern**>] | **unchanged** | <**option**> {<**num**>}**.** Selects a Xabsl option. The command suppresses the option currently selected by the Xabsl engine and replaces it with the option specified by this command. Options can be parameterized by a list of decimal numbers. Type *xo ?* to list all available Xabsl options. The resulting list can be shortened by specifying a search pattern after the question mark. Use *xo unchanged* to switch back to the option currently selected by the Xabsl engine. The command *xo* only works if the Xabsl symbols have been requested from the robot (cf. Sect. 7.3.4).

xos ? [<**pattern**>] | <**outputSymbol**> **(on** | **off** | **?** [<**pattern**>] | **unchanged** | <**value**> **).** Show or set 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. 7.3.4).

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

7.6 Examples

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

7.6.1 Recording a Log File

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

```
# connect to a robot
sc Remote 10.1.0.101

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

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

7.6.2 Replaying a Log File

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

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

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

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

# 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
```

7.6.3 Remote Control

This script demonstrates joystick remote control of the robot. The last command has 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

# joystick: motion request
js 2 100 0.01 # x axis
js 1 0.5 0.01 # rotation axis
jc motion set representation:MotionRequest { motion = specialAction;
    specialActionRequest = { specialAction = stand; mirror = false; };
    walkRequest = { speed = { rotation = $1; translation = { x = $2; y = 0; };
    }; }; kickRequest = { kickType = forward; }; }
```

Chapter 8

Acknowledgements

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

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

Avahi. The Avahi mDNS/DNS-SD daemon (http://www.avahi.org/).

CMU 1394 Digital Camera Driver. Firewire driver and the API for the cameras used in the ground truth environment.

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

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

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

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

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

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

OpenCV. Used in the simulator's ground truth environment (http://sourceforge.net/projects/opencvlibrary).

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

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

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

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

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

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

QT. The GUI framework of the simulator (http://trolltech.com/products).

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

Bibliography

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

[2] A. Burchardt, K. Cierpka, S. Fritsch, N. Göde, K. Huhn, T. Kirilov, B. Lassen, T. Laue, M. Miezal, E. Lyatif, M. Schwarting, A. Seekircher, and R. Stein. B-Smart - Team Description for RoboCup 2007. In U. Visser, F. Ribeiro, T. Ohashi, and F. Dellaert, editors, *RoboCup 2007: Robot Soccer World Cup XI Preproceedings*. RoboCup Federation, 2007.

[3] RoboCup Technical Committee. RoboCup Standard Platform League (Nao) Rule Book. http://www.tzi.de/spl/pub/Website/Downloads/NaoRules2008.pdf as of August 13, 2008.

[4] Stefan Czarnetzki, Daniel Hauschildt, Sören Kerner, and Oliver Urbann. BreDo-Brothers Team Report 2008, 2008. Only available online: http://www.it.irf.uni-dortmund.de/IT/Publikationen/pdf/CHKU08.pdf.

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

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

[7] 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.

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

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

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

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

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

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

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

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

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

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