B-Human

Team Report and Code Release 2012

Thomas Röfer[1], Tim Laue[1], Judith Müller[2], Michel Bartsch[2], Malte Jonas Batram [2], Arne Böckmann [2], Nico Lehmann[2], Florian Maaß[2], Thomas Münder[2], Marcel Steinbeck [2], Andreas Stolpmann[2], Simon Taddiken[2], Robin Wieschendorf [2], Danny Zitzmann[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 8, 2012

Contents

Chapter 1

Introduction

This document reports about the changes made in B-Human's system since we released our code last year [7].

1.1 About Us

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

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

In parallel to these activities, B-Human started as a part of the joint team BreDoBrothers that has been a cooperation of the Technische Universität Dortmund and the Universität Bremen. The team participated in the Humanoid League in RoboCup 2006. The software was based on previous works of the GermanTeam [4]. This team was split into two single Humanoid teams, because of difficulties in developing and maintaining a robust robot platform across two locations. The DoH!Bots from Dortmund as well as B-Human from Bremen participated in RoboCup 2007; B-Human reached the quarter finals and was undefeated during round robin. In addition to the participation in the Humanoid League at the RoboCup 2008, B-Human also attended a new cooperation with the Technische Universität Dortmund. Hence, B-Human took part in the Two-Legged Competition of the Standard Platform League as part of the team BreDoBrothers, who reached the quarter finals. After the RoboCup 2008, we concentrated our work exclusively on the Two-Legged SPL. By integrating all the students of the Humanoid League into team B-Human, the BreDoBrothers would have had more than thirty members. Therefore we decided to end the cooperation by mutual agreement to facilitate a better workflow and work-sharing.

In 2009, we participated in the RoboCup German Open Standard Platform League and won the competition. We scored 27 goals and received none in five matches against different teams. Furthermore, B-Human took part in the RoboCup World Championship and won the competition, achieving a goal ratio of 64:1. In addition, we could also win first place in the technical

Figure 1.1: The majority of the team members.

challenge, shared with NAO Team HTWK from Leipzig. We repeated our successes in 2010 and won the German Open with a goal ratio of 54:2 as well as the RoboCup with an overall goal ratio of 65:3. In 2011, we won both competitions again with the goal ratios 57:2 and 62:1. In 2012 we won the RoboCup German Open with a goal ratio of 55:1, but only became the runner-up at the RoboCup in Mexico City (with 57:11).

The current team consists of the following persons:

Students. Michel Bartsch, Malte Jonas Batram, Arne Böckmann, Martin Böschen, Nico Lehmann, Florian Maaß, Thomas Münder, Marcel Steinbeck, Andreas Stolpmann, Simon Taddiken, Robin Wieschendorf, Danny Zitzmann.

Senior Students. Alexander Fabisch, Katharina Gillmann, Colin Graf, Alexander Härtl, Tobias Kastner, Ole Jan Lars Riemann, Felix Wenk.

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

1.2 About the Document

As we wanted to revive the tradition of an annual code release several years ago, it is obligatory for us to sustain it this year. However, this year the report only describes the progress made since last year.

Chapter 2 starts with a short introduction of the software required, as well as an explanation of how to run the NAO with our software. Modeling subsystems are introduced in chapter 3.

Chapter 4 explains how we deal with two yellow goals. Chapter 5 describes our new behavior engine. The new GameController is described in chapter 6. Finally, the new build system as

well as several improvements to the simulator are described in chapter 7.

This year's walking engine is the subject of a diploma thesis and will be released separately. Thus this code release contains the walking engine from last year. Additionally we are not releasing our behavior. The code release only contains a very simple behavior that is not able to play soccer.

Chapter 2

Getting Started

The goal of this chapter is to give an overview of the code release package and to give instructions on how to enliven a NAO with our code. For the latter, several steps are necessary: unpacking the source code, compiling the code using Visual Studio 2010, Xcode 4.5, or the Linux shell, setting up the NAO, copying the files to the robot and starting the software.

2.1 Unpacking

The code release package should be unpacked to a location, whose path must *not* contain any whitespaces. After the unpacking process, the chosen location should contain several subdirectories, which are described below.

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

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

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

Make contains the Visual Studio project files, Makefiles, Xcode project files, other files needed to compile the code, the *Copyfiles* tool and two small scripts to download log files from the NAO.

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

Util contains auxiliary and third party libraries and tools (cf. [7, Chap. 9]).

2.2 Building the Code

2.2.1 Project Generation

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

2.2.2 Visual Studio 2010 on Windows

2.2.2.1 Required Software

- Windows 7
- Visual Studio 2010 SP1
- Cygwin – 1.7 with the following additional packages: make, rsync, openssh, libxml2. Add the *...\cygwin\bin* directory to the beginning of the PATH environment variable (before the Windows system directory, since there are some commands that have the same names but work differently). (`http://www.cygwin.com`)
- gcc, glibc – Linux cross compiler for Cygwin, download from the B-Human website[1], use a Cygwin shell to extract in order to keep symbolic links. The content of the file should be placed in the Cygwin root. This cross compiler package was built using crosstool-NG (`http://crosstool-ng.org/`). To use this cross compiler together with our software, we placed the needed boost and python include files into the include directory.
- alcommon – For the extraction of the required alcommon library and compatible boost headers from the *NAO SDK release v1.12.5 linux* (*naoqi-sdk-1.12.5-linux32.tar.gz*) the script *Install/installAlcommon* can be used, which is delivered with the B-Human software. The required package has to be downloaded manually and handed over to the script. It is available at `http://users.aldebaran-robotics.com` (account required). Please note that this package is only required to compile the code for the actual NAO robot.

2.2.2.2 Compiling

Generate the Visual Studio 2010 project files using the script *Make/VS2010/generate.bat* and open the solution *Make/VS2010/BHuman.sln* in Visual Studio. Visual Studio will then list all the components (cf. Sect. 2.8) of the software in the "Solution Explorer". Select the desired configuration (cf. Sect. 2.8, *Develop* would be a good choice for starters) and built the desired project: *SimRobot* compiles every project used by the simulator, *Nao* compiles every project used for working with a real NAO, and *Utils/bush* compiles the B-Human User Shell (cf. Sect. 2.7). You also may select *SimRobot* or *Utils/bush* as "StartUp Project".

2.2.3 Xcode 4.5 on OS X

2.2.3.1 Required Software

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

- OS X 10.8
- Xcode 4.5
- Qt libraries 4.8.3 or above for Mac (`http://releases.qt-project.org/qt4/source/qt-mac-opensource-4.8.3.dmg`)

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

- Cross compiler gcc 4.5.2 running on OS X and compiling for Linux (`http://crossgcc.rts-software.org/download/gcc-4.5.2-for-linux32-linux64/gcc-4.5.2-for-linux32.dmg`). Please note that the binaries produced by this cross compiler are not compatible with the runtime library on the NAO. Therefore, the runtime library is linked statically, which results in relatively large binaries.

- Xcode-Plugin for Cross Compiler gcc 4.5.2 (`http://www.b-human.de/wp-content/uploads/2012/10/GCC-Linux-4.5.2.xcplugin.zip`). It must be unpacked in */Applications/Xcode.app/Contents/PlugIns/Xcode3Core.ideplugin/Contents/SharedSupport/Developer/Library/Xcode/Plug-ins*.

- GraphViz 2.26.3 (`http://www.graphviz.org/pub/graphviz/stable/macos/snowleopard/graphviz-2.26.3.pkg`) (newer versions do not work).

- Java 6 if running OS X 10.7 or later (`http://support.apple.com/kb/DL1421`).

- alcommon – For the extraction of the required alcommon library and compatible boost headers from the *NAO SDK release v1.12.5 linux* (*naoqi-sdk-1.12.5-linux32.tar.gz*) the script *Install/installAlcommon* can be used, which is delivered with the B-Human software. The required package has to be downloaded manually and handed over to the script. It is available at `http://users.aldebaran-robotics.com` (account required). Please note that this package is only required to compile the code for the actual NAO robot. Also note that *alcommonInstall* expects the extension *.tar.gz*. If the NAOqi archive was partially unpacked after the download, e. g., by Safari, repack it again before executing the script.

2.2.3.2 Compiling

Generate the Xcode project by double-clicking *Make/MacOS/generate*. Open the Xcode project *Make/MacOS/B-Human.xcodeproj*. If Xcode suggests to upgrade the project file by replacing the *gcc* compiler with the *llvm* compiler, reject the suggestion, because the *gcc* is the cross-compiler that is still needed. A number of schemes (selectable in the toolbar) allow building SimRobot in the configurations *Debug*, *Develop*, and *Release*, as well as the code for the NAO[2] in all four configurations (cf. Sect. 2.8). For both targets, *Develop* is a good choice. In addition, the B-Human User Shell *bush* can be built as well as the documentation for the robot code in the simulator[3]. It is advisable to delete all the schemes that are automatically created by Xcode, i. e. all non-shared ones. When SimRobot is compiled on OS X 10.8, Qt currently generates a lot of compatibility warnings. They are best suppressed by deleting the line that generates them from the file *qglobal.h* in the Qt installation.

When building for the NAO, a successful build will open a dialog to deploy the code to a robot (using the *copyfiles* script, cf. Sect. 2.4).[4] If the *login* script was used before to login to a NAO, the IP address used will be provided as default. In addition, the option `-r` is provided by default that will restart the software on the NAO after it was deployed. Since *copyfiles* only copies changed files by default, it is advisable to initiate a full copy with the option `-d` when starting to work with a NAO that has been used by someone else before.

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

2.2.4 Linux Shell

2.2.4.1 Required Software

Requirements (listed by common package names) for an x86-based Linux distribution (e. g. Ubuntu 10.10):

- g++-4.4, g++, make
- libqt4-dev, qt4-dev-tools, libphonon-dev – 4.3 or above (`http://qt-project.org/downloads#qt-lib`)
- doxygen, graphviz – For compiling the documentation.
- openssh-client, rsync – For deploying compiled code to the NAO.
- glibc < 2.15. The NAO does not support glibc versions above 2.14.
- alcommon – For the extraction of the required alcommon library and compatible boost headers from the *NAO SDK release v1.12.5 linux* (*naoqi-sdk-1.12.5-linux32.tar.gz*) the script *Install/installAlcommon* can be used, which is delivered with the B-Human software. The required package has to be downloaded manually and handed over to the script. It is available at `http://users.aldebaran-robotics.com` (account required). Please note that this package is only required to compile the code for the actual NAO robot.

On Ubuntu 10.10 you just need this command line to install all needed dependencies:

```
sudo apt-get install g++-4.4 g++ make libqt4-dev qt4-dev-tools libphonon-dev
doxygen graphviz openssh-client rsync libglew1.5-dev libxml2-dev  libprotobuf-
dev libjpeg62-dev
```

2.2.4.2 Compiling

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

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

To clean up the whole solution use:

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

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

2.3 Setting Up the NAO

2.3.1 Requirements

First of all, download the current version of the *NAO OS image for Atom* and the *NAO flasher* from the download area of `http://users.aldebaran-robotics.com` (login required).

At the time of writing the current version of NAO OS is 1.12.5. In order to flash the robot you furthermore need a usb stick having at least 2 GB space.

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

sed, *tar*, *mktemp*, *whoami*, and *tr*.

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

The commands in this chapter are shell commands. They should be executed inside a Unix shell. Use Cygwin on Windows to execute the commands. All shell commands should be executed from the *Install* directory.

2.3.2 Creating a Robot Configuration

Before you start setting up the NAO, you need to create configuration files for each robot you want to set up. To create a robot configuration run *createRobot*. The script expects a team id, a robot id, and a robot name. The team id is usually equal to your team number configured in *Config/settings.cfg* but you can use any number between 1 and 254. The given team id is used as third part of the IPv4 address of the robot on both interfaces. All robots playing in the same team need the same team id to be able to communicate with each other. The robot id is the last part of the IP address and must be unique for each team id. The robot name is used as host name in the NAO operating system and is saved in the chestboard of the NAO as *BodyNickname*.

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

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

./createRobot -t 3 -r 25 Penny

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

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

2.3.3 Managing Wireless Configurations

All wireless configurations are stored in *Install/Network/Profiles*. Additional configurations can be placed in *Install/Network/Profiles*. They will be installed alongside the *default* configuration. Initially the NAO will load the *default* configuration.

To switch between different configurations use the *setprofile* script on the NAO.

2.3.4 Setup

First of all plug in your USB stick and start the *NAO flasher tool*[6]. Select the *NAO OS image for Atom* and your stick. Enable "Factory reset" and click on the write button.

[5]The tools should be present on every Linux distribution. If not, the following command should help: *sudo apt-get install sed bzip2 tar coreutils*

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

After the stick has been flashed plug it into the NAO and press the chest button for about 5 seconds. Afterwards the NAO will automatically install NAO OS and reboot. Make sure that you are connected to the NAO using a network cable when it reboots. Otherwise the NAO will not be able to obtain an IP address. Once the reboot is finished the NAO will do its usual Aldebaran wake up procedure. Press the chest button to obtain the NAO's IP address. If the NAO could not obtain an IP make sure that the network cable is plugged in correctly and restart the NAO.

Make sure that you are able to ping the NAO before you proceed. If you have not done so before run the *generate* command from *Make/Linux* now. It creates temporary SSH keys that are used while installing the software on the robot. Now run *installRobot* to apply B-Human's modifications to the NAO's system by typing *./installRobot -a <ip> <Robot name>* in the *Install* directory.

For example run *./installRobot -a 169.254.220.18 Penny*.

Follow the instructions inside the shell. The shell will stop responding during the process due to a restart of the NAO's network interface. After some time you should be able to ping the NAO using the IP specified in *createRobot*. Now you can proceed to copy B-Human onto the NAO.

2.4 Copying the Compiled Code

2.4.1 Using copyfiles

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

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

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

copyfiles requires two mandatory parameters. First, the configuration the code was compiled with (*Debug*, *Optimized*, *Operate* or *Release*)[7], and second, the IP address of the robot. To adjust the desired settings, it is possible to set the following optional parameters:

Option	Description
-l <location>	Sets the location, replacing the value in the *settings.cfg*.
-t <color>	Sets the team color to *blue* or *red*, replacing the value in the *settings.cfg*.
-p <number>	Sets the player number, replacing the value in the *settings.cfg*.
-n <number>	Sets team number, replacing the value in the *settings.cfg*.
-m n <ip>	Copies to IP address <ip> and sets the player number to *n*.
-wc	Compiles also under Windows if the binaries are outdated.
-nc	Never compiles, even if the binaries are outdated.

Possible calls could be:

```
copyfiles Develop 134.102.204.229
copyfiles Release -m 1 10.0.0.1 -m 3 10.0.0.2
```

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

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

2.5 Working with the NAO

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

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

There are several scripts to start and stop NAOqi and *bhuman* via SSH.

stop stops running instances of NAOqi and *bhuman*.

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

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

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

bhumand start|stop|restart starts, stops or restarts the *bhuman* executable. After uploading files with *Copyfiles* *bhuman* and NAOqi are restarted automatically.

status shows the status of NAOqi and *bhuman*.

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

2.6 Starting SimRobot

On Windows and OS X, SimRobot can either be started from the development environment, or by starting a scene description file in *Config/Scenes*[8]. In the first case, a scene description file has to be opened manually, whereas it will already be loaded in the latter case. On Linux, just run *Build/SimRobot/Linux/<configuration>/SimRobot*, either from the shell or from your favorite file browser, and load a scene description file afterwards. When a simulation is opened for the first time, only the scene graph is displayed. The simulation is already running, which can be noted from the increasing number of simulation steps shown in the status bar. A scene

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

view showing the soccer field can be opened by double-clicking *RoboCup*. The view can be adjusted by using the context menu of the window or the toolbar. Double-clicking *Console* will open a window that shows the output of the robot code and that allows entering commands. All windows can be docked in the main window.

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

Although any object in the scene graph can be opened, only displaying certain entries in the object tree makes sense, namely the main scene *RoboCup*, the objects in the group *RoboCup/robots*, and all other views.

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

2.7 B-Human User Shell

The B-Human User Shell (bush) accelerates and simplifies the deployment of the code and the configuration of the robots. It is especially useful when controlling several robots at the same time, e. g., during the preparation for a soccer match.

2.7.1 Configuration

Since the bush can be used to communicate with the robots without much help from the user, it needs some information about the robots. Therefore, each robot has a configuration file *Config/Robots/<RobotName>/network.cfg* which defines the name of the robot and how it can be reached by the bush.[9] Additionally you have to define one (or more) teams which are arranged in tabs. The data of the teams is used to define the other properties which are required to deploy code in the correct configuration to the robots. The default configuration file of the teams is *Config/teams.cfg* which can be altered within the bush or with a text editor. Each team can have the configuration variables shown in Table 2.1.

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

2.7.2 Deploying Code to the Robots

For the simultaneous deployment of several robots the command *deploy* should be used. It accepts a single optional parameter that designates the build configuration of the code to be deployed to the selected robots. If the parameter is omitted the default build configuration of the currently selected team is used. It can be changed with the drop-down menu at the top of the bush user interface.

[9] The configuration file is created by *createRobot* described in Sect. 2.3.2.

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

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

Before the *deploy* command copies code to the robots, it checks whether the binaries are up-to-date. If needed, they are recompiled by the *compile* command, which can also be called independently from the *deploy* command. Depending on the platform, the *compile* command uses *make*, *xcodebuild*, or *MSBuild* to compile the binaries required. On OS X, it does not keep the project files up-to-date, i. e. call one of the *generate* scripts. Instead, you have to do it manually (cf. Sect. 2.2).

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

2.7.3 Managing Multiple Wireless Configurations

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

the robot. Which configuration is used can be specified in the first argument of *changeWireless*. If the argument is omitted, the wireless configuration of the selected team is used.

2.7.4 Substituting Damaged Robots

The robots known to the bush are arranged in two rows. The entries in the upper row represent the playing robots and the entries in the lower row the robots which stand by as substitutes. To select which robots are playing and which are not you can move them by drag&drop at the appropriate position. Since this view only supports eight robots at a time, there is another view called *RobotPool*, which contains all other robots. It can be pulled out at the right side of the bush window. The robots displayed there can be exchanged with robots from the main view.

2.7.5 Creating Color Tables

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

- In order to deploy the B-Human code to the selected NAOs, to restart *bhumand* on the NAOs, and to connect to the NAOs with the simulator, you just have to type *prepareColtable*. Training will automatically be enabled in the simulator scene loaded (cf. [7, Chap. 4.14]).
- You can copy all color tables and training sets to the selected NAOs with the command *colorDeploy <location>*. Remember to load the new color table, e. g. by restarting *bhumand* with the command *restart*.
- You can push your locally modified color tables to your remote git repository by executing *commitColtable*. If you use another version control system, you will have to adapt this command.

2.7.6 Monitoring Robots

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

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

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

2.8 Components and Configurations

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

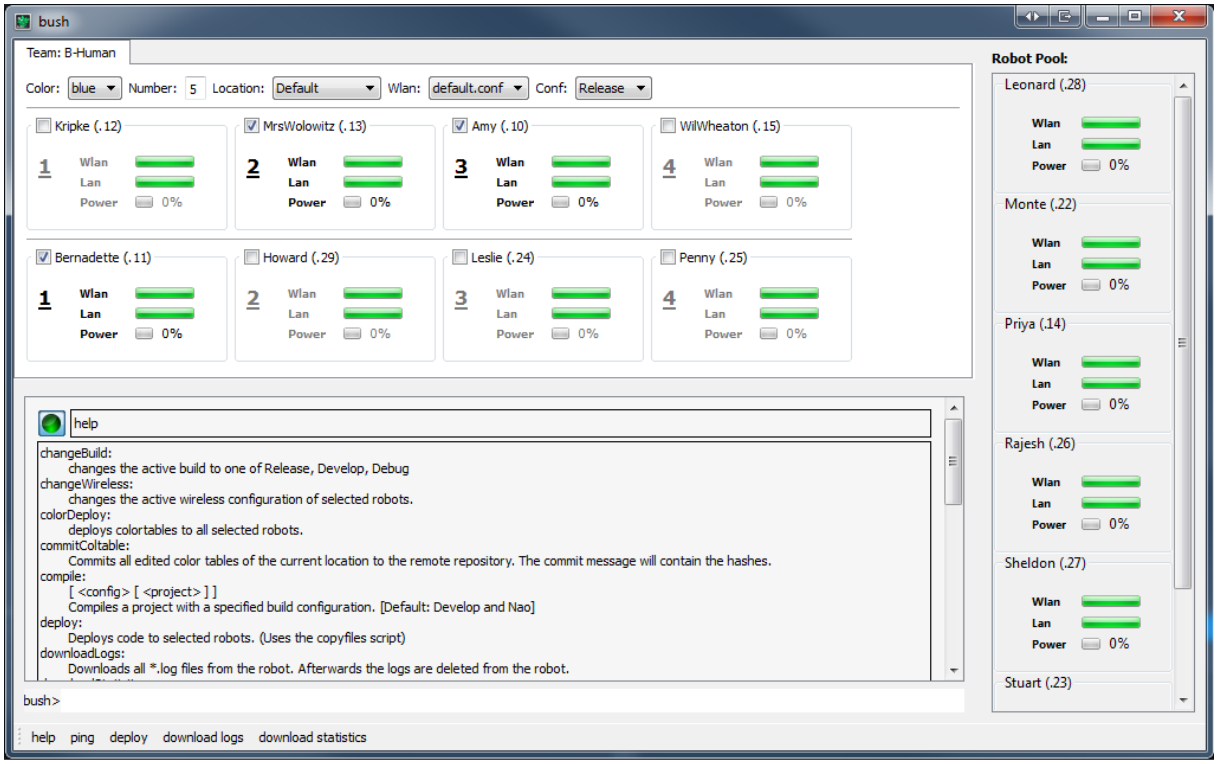

Figure 2.1: bush on Windows 7.

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

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

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

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

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

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

Nao cross-compiles the B-Human executable for the NAO. It is available in *Operate*, *Release*, *Develop*, and *Debug* configurations, where *Operate* produces "game code" without any

support for debugging. The configuration *Develop* produces optimized code, but still supports all debugging techniques. If you want to disable assertions as in *Operate* but enable debugging support, *Release* can be used.

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

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

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

[1] - on Windows - `http://msdn.microsoft.com/en-us/library/0b98s6w8(v=vs.100).aspx`

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

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

Behavior compiles the behavior engine and the behavior specified in *Src/Modules/BehaviorControl/StateMachineBehaviorEngine/Behavior*.

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

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

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

2.9 Configuration Files

Since the recompilation of the code takes a lot of time in some cases and each robot needs a different configuration, the software uses a huge amount of configuration files. All these files, which are used by the software[10] are located below the directory *Config*.

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

In addition, there are configuration files that depend on the current *location*, e. g. "RoboCup2012" or "OurLab". Locations can also be used to configure robots for a different task, e. g. for the "Open Challenge". The current location is set in the file *Config/settings.cfg*.

To handle all these different configuration files there are fall-back rules that are applied if a requested configuration file is not found. If a configuration file is robot-specific, its path is ex-

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

panded using the method `Settings::expandRobotFilename`. In that case, the search sequence for the file is:

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

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

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

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

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

Chapter 3

Modeling

Collisions with other robots, especially when not detected, have a huge influence on the correctness of the self-localization, particularly on the new symmetric field. Therefore, two new modules have been developed to detect and minimize body contact.

Additionally, the ultrasound module has been redesigned to allow more precise obstacle detection.

3.1 Arm Contact Detection

In order to improve the close-range obstacle avoidance, robots should detect whether they touch obstacles with their arms. When getting caught in an obstacle with an arm, there is a good chance that the robot gets turned around, causing its odometry data to get erroneous. This, in turn, affects the self-localization (compare with 4.1).

3.1.1 Previous Implementation

In the Code Release 2011 [7] a module to detect arm contacts already exists. It calculates the difference of the intended and actual position of an arm's shoulder joint and reports a contact whenever a certain threshold is exceeded. Due to the fact that the implementation does not handle erroneous measurements as caused by low stiffness and fast arm movement it turned out that constantly wrong arm contacts were measured.

Additionally, the old module only provides binary information about an arm contact: Whether it remained or not. That information is only partly usable when trying to track odometry changes.

3.1.2 Improvements

In order to improve the precision as well as the reactivity the ArmContactModelProvider has been moved from the Cognition to the Motion process. In doing so we enabled the module to query the required joint angles with 100 frames per second instead of 30 as in the previous release. In comparison to the previous implementation not only the difference of the intended and actual position of the shoulder joint per frame is calculated but also is buffered over several frames. Thus the average error over several frames can be calculated. Every time this error exceeds a certain threshold an arm contact is reported (cf. 3.1.7). Using this method, small errors caused by arm motions in combination with low stiffness can be smoothed. Hence we were able to

increase the accuracy wither there were a real contact caused by an obstacle. Since we also had to deal with prolonged erroneous measurements we introduced a malfunction threshold. Whenever an arm contacts continues for longer than this threshold all further arm contacts will be ignored until no contact is measured.

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

3.1.3 Detecting Contact

In order to detect arm contacts the first step of our implementation is to calculate the difference between the measured and commanded shoulder joint angles. Since we noticed that there is a small delay between receiving new measured joint positions and commanding them we do not compare the commanded and actual position of one shoulder joint from one frame. Instead we are using the commanded position from n frames[1] earlier as well as the newest measured position. Thus the result is more precise.

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

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

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

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

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

$$d_a = d \cdot f$$

3.1.4 Determining Push Direction

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

Table 3.1.4 shows how the signed components are converted into compass directions for the **right** arm. The compound directions NW, NE, SE, SW are constructed by simply combining the above rules if **both** components are greater than their thresholds, e. g. $x < 0 \wedge y < 0$ results into direction SE.

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

	x	y
is positive	N	E
is negative	S	W

Table 3.1: Converting signed error values into compass directions for the right shoulder and with $-y$ for the left

3.1.5 Usage

The push direction of each arm is used to add an obstacle to the robot's internal obstacle grid causing the robot to perform an evasion movement when it hits an obstacle with one of his arms. In addition, arm contacts might lower the robot's side confidence value (cf. 4.1), as arm contact with an obstacle might rotate the robot.

3.1.6 Debugging

For debugging and configuration purposes, a debug drawing (cf. Figure 3.2) was added to visualize the error values gathered by the ArmContactModelProvider. It depicts the error vectors for each arm and can be activated by using the following commands in SimRobot:

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

Figure 3.1: The left arm of the left robot is blocked and thus pushed back.

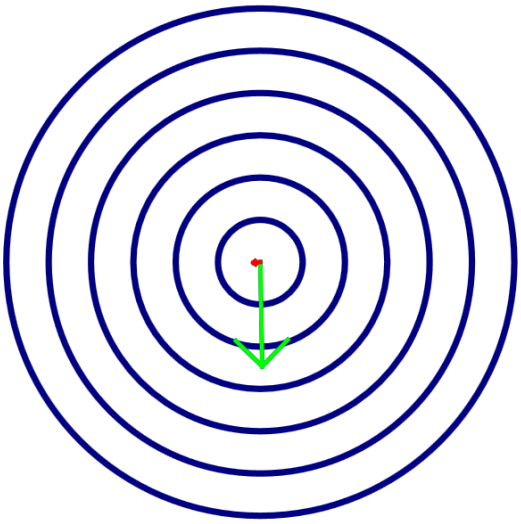

Figure 3.2: The green arrow denotes push direction and strength for the left arm.

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

```
call ArmContactDebug
```

These plots are useful when parameterizing the ArmContactModelProvider.

3.1.7 Configuration

The ArmContactModelProvider does not have a configuration file. However, it can be configured by modifying the default values in the ArmContactModelProvider::Parameters class. The following values can be configured:

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

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

frameDelay Frame offset for calculating the arm divergence (cf. Sect. 3.1.3).

debugMode Enables debug sounds.

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

3.2 Foot Contact Detection

If all mechanisms of avoiding an obstacle fail, the robot might just run into it without even noticing it. The new FootContactModelProvider makes use of the NAO's foot bumpers to detect obstacles that the robot ran into.

Each foot has a bumper that consists of two contact sensors that provide binary contact detection. The state of each sensor pair is aggregated into a single contact information which is then buffered over a certain amount of frames. If the sum of all values in the contact buffer exceeds a given threshold, a foot contact for the corresponding foot is reported.

On older robots, the contact bumpers tend to get jammed, causing constant contact to be reported. Therefore, the module ignores contacts that remain for longer than a certain amount of frames.

3.2.1 Configuration

It is not possible to configure the FootContactModelProvider using a configuration file. However one can modify the values in FootContactModelProvider::Parameters directly. The following values can be configured:

contactThreshold Threshold (in contacts/s) before a foot contact is detected.

malfunctionThreshold Threshold in frames of continuous contact before the contact is ignored.

debug Enables the debug sound.

soundDelay Delay between debug sounds.

3.3 Ultrasound

Detecting several obstacles in middle range by only using the vision system is very tedious. Vision needs a lot of specific algorithms and strongly depends on lighting conditions. Additionally various obstacle structures need to be known beforehand. Ultrasound is more dynamic in detecting unknown and overlapping obstacles and is therefore very suitable to provide an additional, albeit imprecise, model of the current surroundings. This chapter describes our approach to improve the precision of ultrasonic measurements.

3.3.1 The Sonar Hardware

NAO is using two ultrasonic sensors, each composed of one transmitter and one receiver inside NAO's chest. The sensors can be operated in many different modes (cf. [6]). These modes are used to control the interaction between the transmitters and receivers.

Previously following modes were used:

0	Left transmitter & left receiver
1	Left transmitter & right receiver
2	Right transmitter & right receiver
3	Right transmitter & left receiver

Table 3.2: Previously used sensor modes

The modes 4 and 6 are used now. Mode 4 means: use the left emitter and both receivers (usually called left-to-both). Mode 6 means right-to-both. This enhancement was done to double the sample rate of the sonar module. The modes can be changed in the configuration file *usControl.cfg* in the *Locations* directories. However this should be done with caution because the SensorFilter (cf. [7, Chap. 5.1] and MultiUSObstacleGridProvider depend on the modes 4 and 6.

Each receiver delivers ten measurements. The detection range reaches from 25 cm to 255 cm, whereas values below 25 cm are clipped. Thus, 25 cm actually means 25 cm or closer. Due to inaccuracies or mistakes in NAOqi the sensors sometimes falsely report an obstacle below 25 cm as 180 cm instead of 25cm. Therefore, very close robots are not detected very well.

3.3.2 Filtering Measurements

Even though the receiver provides ten measurements, only the first three are used to create a sound world state. With respect to the unstable and noisy measurements of the sonar module these echoes are arithmetically averaged by using a median filter over 6 frames. The filtering is performed by the SensorFilter.

3.3.3 Providing an Obstacle Model

The MultiUSObstacleGridProvider uses the filtered measurements to create an obstacle grid. The *USObstacleGrid* is composed of cells. Each cell covers a 36 cm^2 area of the field around the robot. Additionally it has a weight from 0 to a configurable value (see 3.3.6, cellMaxOccupancy). The MultiUSObstacleGridProvider increases the weight of all cells corresponding to a measurement.

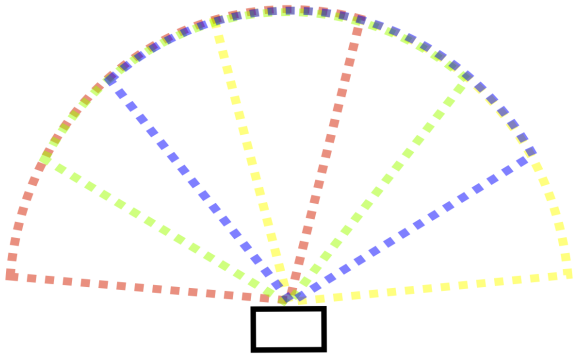

Figure 3.3: Us sensor cones outgoing from the robot. The square represents the robot. The colored cones are the regions, covered by the different ultrasound modes.

3.3.3.1 Measurement Entering Algorithm

The different sensor modes define four regions represented as cones based on the calibration (cf. 3.3.7). For best obstacle detection the cones should be uniformly distributed (cf. Fig. 3.3).

The four overlapping cones (red, green, blue and yellow) form up to seven sub-cones depending on the calibration, which are the highest resolution for an obstacle detection. These sub-cones are represented as a list of angle intervals.

Every frame a new measurement is provided by the SensorFilter. This measurements are grouped by their distance based on the parameter groupingDistance declared in the file *usObstacleGrid.cfg*. Those groups are supposed to be a single obstacle, assuming that two obstacles do not have the same distance. Unfortunately, in some cases two different obstacles are merged into a single one. Thanks to the movement the two obstacles will be separated very fast, with respect to the groupingDistance, so that the cellOccupiedThreshold used by the ObstacleCombinator (cf. Sect. 3.3.4) will not be reached and the merged group will not be marked as an obstacle.

The obstacle detection algorithm is separated into three parts:

1. In figure 3.4 an obstacle, represented by a square, is placed in front of the robot. The colored solid lines represents the measurements of the sensor cones, together the three measurements will form a group. Each measurement in one group increases a counter for every sub-cone interval, represented by the array beneath.

2. Afterwards all sub-cones that are overlapped by a cone without a measurement at the current distance will be reset to zero. (cf. the yellow cone resets all his sub-cones 3.5)

3. Finally the sub-cones with the highest counter have to be found. In this example this is only the third sub-cone. Afterwards an obstacle will be entered into the USObstacleGrid for each found sub-cone.

3.3.4 Combining Obstacles

After recording the measured obstacles in the *USObstacleGrid*, the ObstacleCombinator (cf. [7, Chap. 4.2.7]) considers the given information to combine cells into clusters by using the floodfill-algorithm. Cells are only combined if their weight exceeds a specified threshold (see 3.3.6, cellOccupiedThreshold).

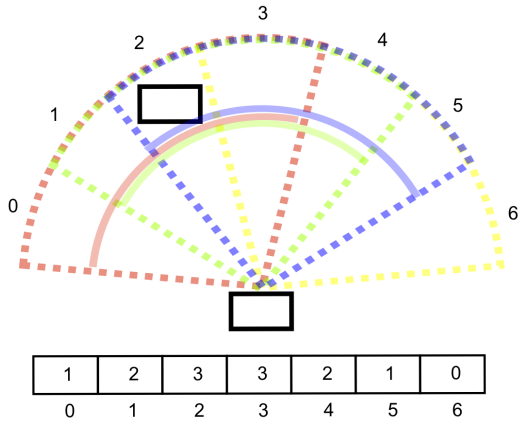

Figure 3.4: Us sensor cones with an obstacle

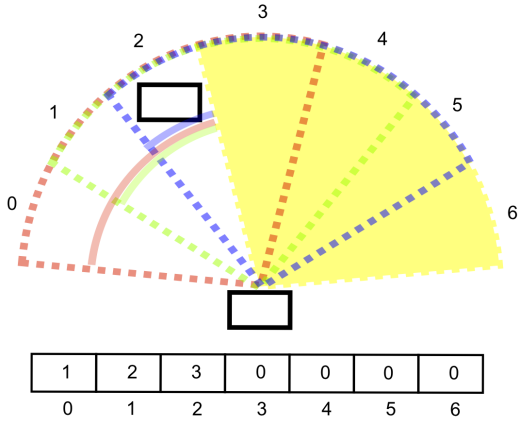

Figure 3.5: Resetting the hit counters of not measured cone

3.3.5 Aging Cells

Playing soccer means moving around and therefore most of the robots will not stay on the same position for longer periods. This leads toward an aging of the cells triggered by the MultiUSObstacleGridProvider every frame. Aging means that the weight of a cell will be decreased if no measurement is present. The aging-speed can be configured (see 3.3.6, cellFreeInterval).

Overlapping Measurements and Very Fast Aging

If the sensors measured an obstacle behind another obstacle, which is already present in the *USObstacleGrid*, the weight of the closer obstacle will be aged much faster. This makes sense, since the sensors cannot measure an obstacle with distance x, if another obstacle with distance $y < x$ is in between. This mechanism is called *Freedrawing*.

3.3.6 Configuration

The following values can be configured in *usObstacleGrid.cfg* in the *Locations* directories.

cellFreeInterval The time it takes to decrease a cells weight by one.

groupingDistance If the distance between two measurements is less then this value, those measurements will be grouped together.

maxValidUSDist The maximum distance to accept a measurement as valid.

minValidUSDist The minimum distance to accept a measurement as valid.

cellOccupiedThreshold Cells with a weight greater than this value are recognized as obstacles.

cellMaxOccupancy The maximum weight of a cell.

usRightPose Position and orientation of right US sensor.

usLeftPose Position and orientation of left US sensor.

usCenterPose Position and orientation of (virtual) center US sensor.

agingFactorOnFreeDrawing In case of Freedrawing (cf. 3.3.5) a cells value will be multiplied with this value.

3.3.7 Calibration

As a result of imprecise manufacturing, the ultrasound cones are usually not in the place where they are supposed to be according to the documentation. Therefore, the opening and closing angles of each cone have to be calibrated.

The calibration has to be set in *Config/Robots/RobotName/usCalibration.cfg*. It contains the following values:

llLeftOpeningAngle Left angle of the left-to-left cone.

llRightOpeningAngle Right angle of the left-to-left cone.

lrLeftOpeningAngle Left angle of the left-to-right cone.

lrRightOpeningAngle Right angle of the left-to-right cone.

rrLeftOpeningAngle Left angle of the right-to-right cone.

rrRightOpeningAngle Right angle of the right-to-right cone.

rlLeftOpeningAngle Left angle of the right-to-left cone.

rlRightOpeningAngle Right angle of the right-to-left cone.

medianBufferSize Size of the buffer used for the median filter (cf. 3.3.2).

maxScanRange Maximum scan range for this robot. Measurements further away than this value will be ignored.

The left angle is the one closer to the robots left hand.

In assistance to the calibration process, the SensorFilter provides plots for raw and filtered sensor data. These plots can be activated using the following command in SimRobot:

```
call USDebug
```

3.3.7.1 Calibration process

The following process describes how to calibrate the left-to-left cone. All other cones are calibrated analoge to this description.

1. Place NAO on even ground.
2. Make sure there are at least 2 meters of free space in front of NAO.
3. Draw a semi-circle around NAO. The radius should be somewhere between 500 and 1000 mm. (See figure 3.6 for an example setup)
4. Add degree markings to the circle. NAO's left hand should point to 90°.
5. Make sure that all LeftToLeft plots show a constant measurement at 2550.

Figure 3.6: Ultrasound calibration device

6. Move an obstacle around the circle starting from 90° (left hand of the robot).
7. Monitor the LeftToLeft plots in SimRobot while moving an obstacle around the circle.
8. As you move the obstacle to the border of the cone the plot will start to jitter and stabilize.
9. As soon as it stabilizes you have found the value for llLeftOpeningAngle.
10. Continue moving the obstacle around the circle while monitoring the value.
11. As you move the obstacle closer to the right border of the cone the plot will start to jitter again.
12. As soon as the jitter gets too strong (above 50 %) you have found the value for llRightOpeningAngle.
13. Once all angles are recorded, move the robot onto an empty field and let him walk around.
14. Monitor all plots while the robot is walking. Check if any sensor measures the ground.
15. If the robot measures the ground while walking, adjust *maxScanRange* to a value below the lowest ground measurement.

Please note that the precision of the calibration process is average at best.

3.3.8 Notes on the Quality of Ultrasonic Measurements and our Approach

The precision of ultrasonic measurements is influenced by several factors:

- Manufacturing impreciseness causes different opening angles on every robot which is leading to a long and error-prone calibration process.

- Reflections from obstacles far away can be registered as very close obstacles because the reflected signal arrives in later measurement cycles.
- All NAOs use the same frequency. The more NAOs are on the field the more interference and false measurements are registered.
- The opening angles of the sensors are very large making the precise localization of obstacles impossible.
- The calibration highly depends on the kind of obstacle used.

Due to these factors ultrasonic measurements remain very imprecise even after careful calibration. The introduced algorithm does not handle that imprecision very well. Additionally it assumes that overlapping measurements always belong to one obstacle, this is obviously not the case.

Chapter 4

Localization

This year's rule changes introduced two yellow goals instead of a yellow and a blue one. This chapter describes how our self-localization deals with that problem. Additionally, it introduces a way to detect very close goal posts.

4.1 Orientation on a Symmetric Field

For the SPL Open Challenge 2011, we already presented a preliminary solution for playing on a field with two yellow goals, as described in [7]. For the 2012 competitions, we used the same approach but implemented several refined heuristics. The approach does not require any additional detections of certain features in the field's surrounding.

4.1.1 Changes to Existing Modules

For self-localization, B-Human uses a combination of a particle filter and an Unscented Kalman filter [3] (implemented in the modules SelfLocator and RobotPoseUnscentedFilter respectively). The former computes a global position estimate; the latter performs local tracking for refining the global estimate (cf. [7]). Both components needed to be modified for the new field layout. In a first step, the measurement models of both filters have been adapted by adding two more yellow goal posts and removing the blue posts accordingly. In addition, the particle filter's sensor resetting mechanism needed to be adapted in a similar manner. These changes could already allow a proper position tracking in many situations.

Nevertheless, the particle filter's sensor resetting mechanism might generate samples that are at a totally wrong position due to the ambiguity of the goal posts that are used for computing the new samples. In the worst case, which occasionally occurs, these samples outstandingly match the perceptions made in the following execution cycles. Consequently, a majority of all samples will be replaced by samples at the resetting position. Fortunately, a Standard Platform League field with uniform goal colors is twofold point symmetric with reference to the field's center. This means that even if the sensor resetting causes a wrong position, this position is not arbitrary but symmetric to the previous position. By assuming that no teleportations to symmetric positions occur in reality, we can deal with this kind of symmetry in a straightforward manner by computing a mirrored robot pose whenever the sample set represents the symmetric counterpart.

However, this approach is not failsafe as events such as persistent collisions with other robots close to the field center might confuse a robot to such an extent that it looses track of its orien-

tation. Furthermore, a correct initialization is required when a robot starts to play. Solutions to these two problems are described in the following subsections.

4.1.2 The Own Side Model – A Reliable Base for Position Tracking

Whenever a robot starts to play, the field half, in which it is located, can be inferred from the rules:

- Before the game, after the halftime break, and after a timeout, the teams position their robots next to their own half for automatic placement. When the game state changes to *READY*, a robot must be in its own half.
- All robots must be in their own half when the game starts, i. e. the game state changes from *SET* to *PLAYING*.
- After a penalty, a robot is always placed in its own half.

In some of these situations, even more detailed position information is available, e. g. the goalkeeper is inside its penalty area when the game state changes from *SET* to *PLAYING*, the possible positions for reentering the field after a penalty are specified in detail, and if the opponent team has kickoff, a minimum distance from the opponent half is guaranteed. By considering a robot's walked distance through integrating odometry offsets over time, it is possible to infer the correct field side over a longer time period. In many matches, for instance, the goalkeeper does not move far enough to have any opportunity to enter the opponent half and thereby always knows its current field half.

The information contained in this model, called the *OwnSideModel*, allows the particle filter as well as the SideConfidenceProvider that handles the recovery from kidnapping (cf. 4.1.3) to resolve the field's twofold point symmetry in a number of game situations. As this approach relies on the rules, we have to expect the referees to apply them correctly.

4.1.3 Computation of Side Confidence for Kidnapping Recovery

The approach described previously allows a correct pose estimation in situations following certain game events. However, in the course of play, the certainties derived from these heuristics vanish. As aforementioned, certain game events might interfere with the localization process and cause uncertainty about the correct position alternative. To recover from such situations, we introduced the concept of side confidence (stored in the representation *SideConfidence*).

As long as the *OwnSideModel* is confident about the current field half, the side confidence remains high. Hence, the particle filter remains in its normal tracking mode. After having walked over a certain distance and thereby lost the absolute confidence about the own side, events such as collisions with other robots (cf. 3.1) might decrease the side confidence over time. Falling down close to the field center even causes a total loss of confidence.

Having reached a state of uncertainty, a robot needs to refer to its teammates to regain confidence as we do not use any additional features outside the field. Similar to our Open Challenge 2011 demonstration, the ball is used as an additional point of reference. Whenever a robot perceives the ball, this observation is set in relation to the fused ball model of its teammates as well as to the mirrored alternative. If a robot's perceptions are consistent with those of its teammates over time, the robot's confidence increases. In contrast, if the perceptions are consistent with the mirrored alternative, the self-localization modules are requested to switch to the alternative

Figure 4.1: Side confidence indicated by colored squares above the robots' heads. The rectangles on the field denote the robots' current pose estimates.

position estimate. For these comparisons, only teammates that are confident about their own positions are considered.

In general, these mutual agreements about the current ball position sustain the team's self-localization (cf. Fig. 4.1) throughout a whole match.

4.2 Alone and Confused Behavior

When using the previously described self-localization approach, situations might occur, in which a robot has a very low side confidence and no contact to other robots that might tell it in which direction it has to play. In this state, the robot is called "alone and confused". During RoboCup 2012, this happened quite often as our robots had many collisions with opponent robots, fell over quite often, and were not able to reliably communicate with their teammates.

Our robots try to resolve such situations by kicking the ball beyond the sideline and interpreting the following referee action. According to the rules [1, Section 3.10], the ball will be replaced

> *a) one meter back from the point it went out or b) one meter back from the location of the kicking robot. We define 'back' as being towards the goal of the team that last touched the ball.*

If the robot detects the ball within a certain time, the SideConfidenceProvider decides in which direction the robot has to play by comparing the actual drop-in position of the ball to the expected one. Thereby, it is crucial that the ball is kicked out close to the middle of the field since it is possible that the ball will be put back to the same side of the robot regardless of the field side if it stands too close to a ground line. To make sure that this is the case, the robot might dribble the ball closer to the middle line before kicking it out.

This solution highly depends on a quick and correct decision of the referees. If the ball is put back to field on the wrong side of the robot, it will continue to play towards the wrong goal.

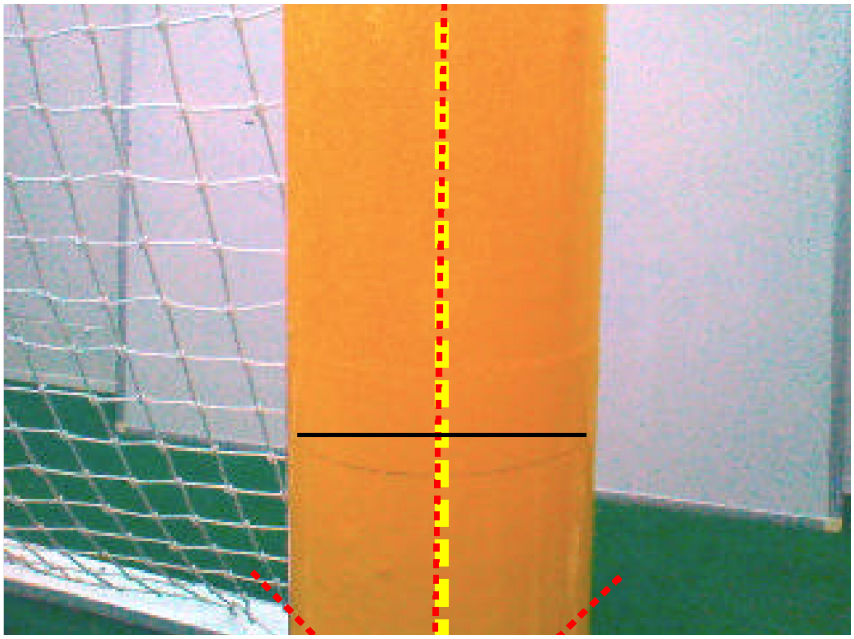

Figure 4.2: Perception of a close goal.

4.3 Perception of Close Goals

Goal posts are important landmarks for the robot's self localization. Thus, recognizing them on the field is a crucial task to determine the robot's current position.

In general, something in the robot's sight is recognized as a goal post, if it is yellow and has the right shape. Additionally, its top must be above the robot's horizon and most important, its bottom must be surrounded by the green field. Where yellow meets green is where the post is located on the field. This is the point the robot is interested in.

This way of recognizing goal posts becomes problematic if the robot is too close to a goal post. In this case, the robot typically cannot see the green field below the post (cf. Fig. 4.2) and thus has to ignore it. To solve this issue, we needed a new algorithm to identify goal posts and their locations even if the robot is too close to see the green below it.

A near post appears in the robot's sight as a big yellow region in the image. As our image preprocessing modules only detect regions below the horizon, we have to check whether the yellow region grows out of the image at the top. The width here must be wider due to perspective distortion. If the robot recognizes such a very close goal post, it must be able to determine the goal post's position even though it cannot see the field. This is done by utilizing the difference in width between the real-world post and the yellow block to determine the distance between the post and the robot.

Chapter 5

State Machine Behavior Engine (SMBE)

Since some aspects of XABSL[5] were very annoying to use and did not fit our desire for a maximum of freedom while programming, we decided to implement our own behavior engine that is based on the concepts of XABSL but solves some of its problems.

Just like XABSL the new behavior engine contains options and states but no longer requires symbols to communicate with the representations provided by other modules.. Instead, C++ code can be used inside the options and states. It can directly access the representations provided by other modules, mainly by simply defining them as a requirement of the behavior module (see [7, Chap. 3.3]). SMBE is based on C++ and uses its own compiler to generate a single class containing all options, states and additional C++ code. This generated class is then compiled as any other source file.

Besides adding all needed representations as requirements of the module StateMachineBehaviorEngine, they also have to be added to the *input* and/or *output* sections of the first option file. In case of the behavior provided in the code release this is the file *Behavior.h*. Also, every behavior file has to include the header file *StateMachineBehavior*.

```
#include <StateMachineBehavior>

input
{
  FallDownState theFallDownState;
  FrameInfo theFrameInfo;
  GameInfo theGameInfo;
  KeyStates theKeyStates;
  MotionInfo theMotionInfo;
};

output
{
  HeadMotionRequest theHeadMotionRequest;
  MotionRequest theMotionRequest;
};
```

5.1 States

States consist of a *decision* part and an *action* part. The *decision* part is used to decide whether the state machine has to switch to another state or whether it has to stay in the current state for one more cycle. A state change is initiated by returning the name of the target state. The *action* part is used to change values of representations or to call other options.

```
state someState()
{
  decision
  {
    if(something)
      return otherState;
  }
  action
  {
    someVariable = someValue;
    OtherOption();
  }
}

state otherState()
{
  ...
}
```

The keywords *stateTime* and *optionTime* return the time that the state machine has spent inside this state or option in milliseconds. It can also be checked whether another option that has been called in the *action* part has reached a certain state.

```
decision
{
  state s = OtherOption();
  if(s == OtherOption.finalState)
    return otherState;
  if(stateTime > 5000 || optionTime > 10000)
    return otherState;
}
action
{
  OtherOption();
}
```

5.2 Options

As stated above, any C++ code can be used inside of options. This includes variables and methods. These can also be used and modified by other options. When an option is executed, but was not executed in the previous cycle, it always starts with its first state.

```
option SomeOption()
{
  state someState()
  {
    decision
    {
    }
    action
    {
```

```
      OtherOption.i = 42;
      OtherOption();
    }
  }
};

option OtherOption()
{
public:
  int i;
  state someState()
  {
    decision
    {
      if(i == someMethod())
        return otherState;
    }
    action
    {
    }
  }
private:
  int someMethod() {return 21;}
};
```

5.2.1 Common State

The *common state* is not a real state, but it defines decisions and actions that are shared by all states of an option. They are executed before the decisions and actions defined in the individual states.

```
option SomeOption()
{
  common()
  {
    decision
    {
      if(something)
        return someState;
    }
    action
    {
      i = 42;
    }
  }
  state someState()
  {
    decision
    {
      if(somethingElse)
        return otherState;
    }
    action
    {
      i *= 2;
    }
  }
  state otherState()
  {
    decision
```

```
    {
    }
    action
    {
    }
  }
};
```

This is basically the same as:

```
option SomeOption()
{
  state someState()
  {
    decision
    {
      if(something)
        return someState;
      if(somethingElse)
        return otherState;
    }
    action
    {
      i = 42;
      i *= 2;
    }
  }
  state otherState()
  {
    decision
    {
      if(something)
        return someState;
    }
    action
    {
      i = 42;
    }
  }
};
```

Chapter 6

Open Challenge

6.1 The new GameController

A RoboCup game has a human referee. Unfortunately the robots cannot understand him directly. Instead the referee's assistant relays his decisions to the robots using a software called GameController.

The current GameController has three major disadvantages:

1. It cannot measures time precisely. For example, this year in Mexico matches where around two minutes too long.
2. The graphical user interface is hard to use. Some areas are packed with buttons while others have lots of free space. Several important buttons are too small while nearly useless buttons are much bigger.
3. The software architecture is not structured very well. This makes it very hard to solve the other issues.

With the regular changes to the RoboCup rules in mind we decided to create a new GameController.

The new GameController2 is written in Java 1.6.

6.1.1 Architecture

The new architecture (cf. Fig. 6.1) is based on a combination of the model-view-controller (MVC) and the command pattern.

The GameController communicates with the robots using a C struct called GameControlData as defined in the RoboCup SPL rules. It contains information about the current game and penalty state of each robot. It is broadcasted via UDP several times per seconds.

Robots may answer using the GameControlReturnData C struct. However this feature is optional and most teams do not implement it.

Both C-structures are transferred to Java for the new GameController2 and they do nothing but hold the information and provide conversion methods from/to a byte stream. Unfortunately the GameControlData does not contain all the information needed to fully describe the current state of the game. For example, it lacks information about the number of timeouts or penalties

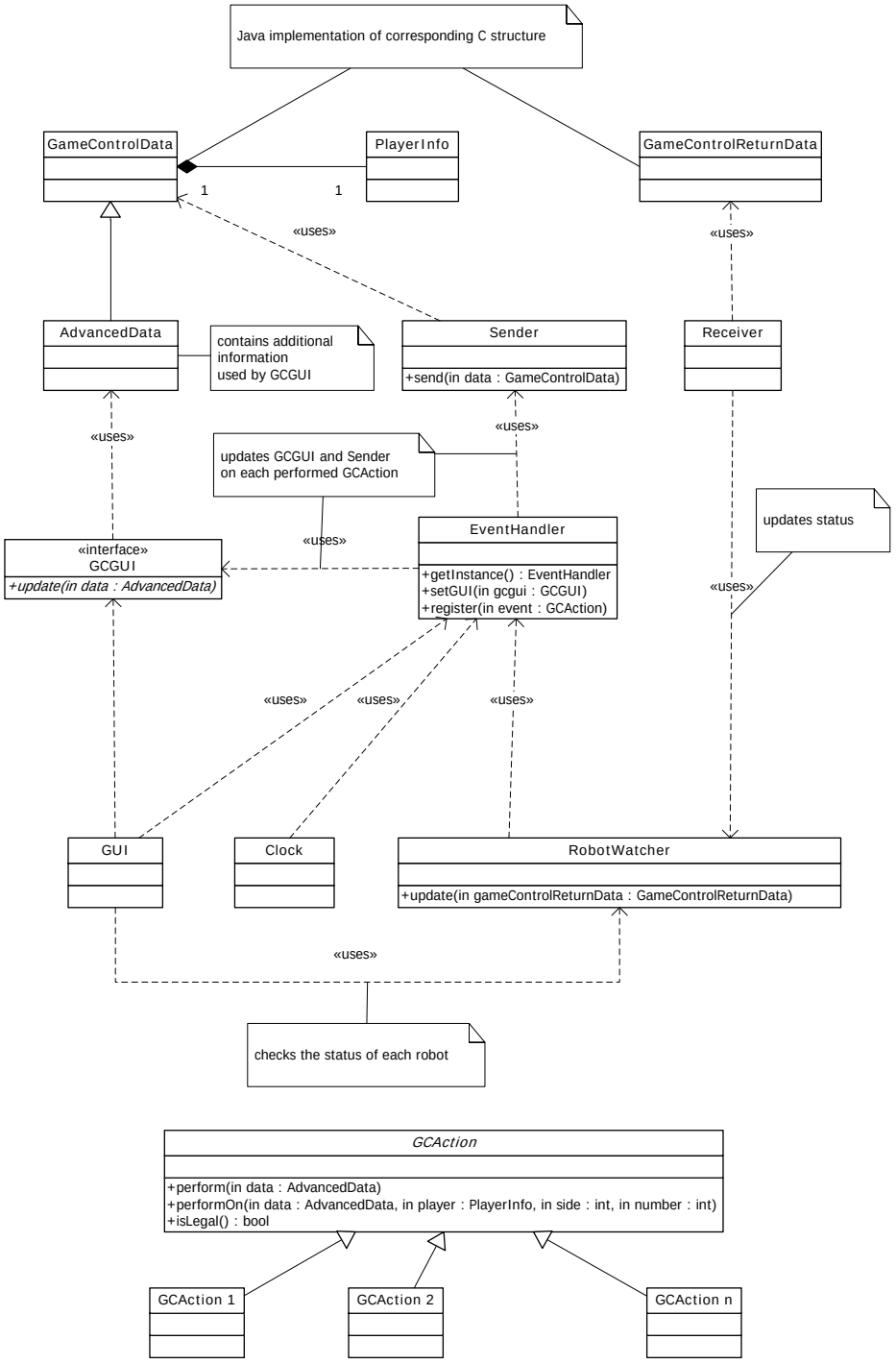

Figure 6.1: The architecture of the new GameController.

that a team has taken and the game time is only precise up to one second. Therefore the GameControlData class is extended by a new class called AdvancedData. This new class holds the complete game state. From the classical MVC point of view the AdvancedData would be the model.

The view component is represented by the GUI-Package. All GUI functionality is controlled via the GCGUI interface. GCGUI only provides an update method with the AdvancedData as parameter. This update method is called frequently. Therefore the GUI can only display the current game state and nothing more.

The controller part of the model-view-controller architecture is kind of tricky because it has to deal with parallel inputs from the user, a ticking clock and robots via network. The old GameController has many get- and set-methods in the model, which could be called from everywhere in the code. Hence, they are synchronized, but since each of them only modifies single entries in the model, referee decisions are not really applied in an atomic way. Therefore, there is a lot of additional lines of code for all the getters and setters without a real benefit.

To simplify the access to the model we only allow access to it from one thread. Everything that modifies the model is within action classes which are defined by extending the abstract class GCAction. All threads (GUI, timing, network) register actions at the EventHandler. The EventHandler executes the actions on the GUI thread (cf. 6.2).

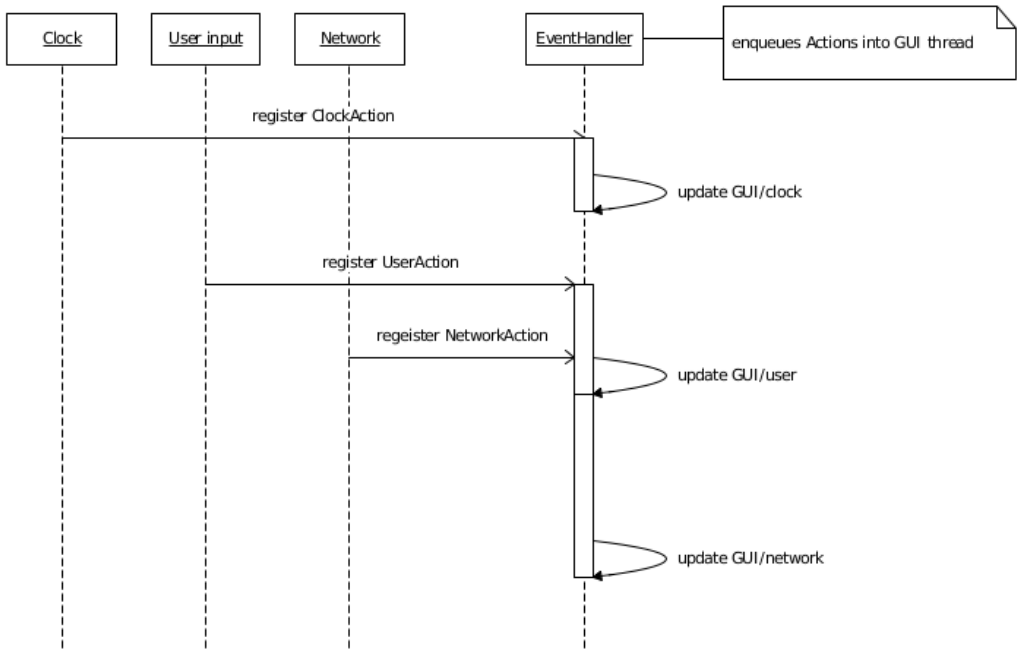

Figure 6.2: The sequences between some threads

Each action knows, based on the current game state, if it can be legally executed according to the rules. For example, switching directly from the initial to the playing state, penalizing a robot for holding the ball in ready state or decreasing the goal count is illegal.

6.1.2 UI Design

The look of the new GUI (cf. Fig. 6.3) is very different from the old one. It is completely symmetric and all buttons are as big as possible. In addition keyboard shortcuts are provided for each button. Thus making it possible to operate the GameController in a more efficient way.

Buttons are only enabled if the corresponding actions are legal. This should decrease the frequency of failures made by the referees and more important it prevents them from doing strange illegal actions to undo a failure they made.

Figure 6.3: The look of the new GameController

However if illegal actions are disabled the assistant referee needs another way to correct mistakes. The GUI provides an undo functionality to do just that.

All undoable actions are displayed in a time line at the bottom of the GameController. By double clicking on one of the actions in the time line the game state will be reverted to the state right before that action has been executed. However the game time won't be reverted.

When testing their robots most teams want to be able to do illegal actions on the GameController. Therefore the GameController has a functionality to switch in and out of a test mode. While being in test mode all actions are allowed at any time.

Chapter 7

Tools

B-Human has a long standing tradition in developing their own tools to aid the software development and debugging process. This chapter describes the changes and additions to our tool collection.

7.1 Build System

In 2011 the tool *zbuildgen* was used to generate project files for Visual Studio and the Makefile used in Linux. This tool has been replaced with the tool *mare* (*Ma*-ke *Re*-placement) which does essentially the same thing. The advantage of mare is that both platforms share a single build target specification file (*Make/Linux/Marefile*) to reduce the time and effort needed to customize or add build targets. To specify build targets, mare uses its own – not yet documented – build target specification language that is based on nested hash tables.

7.2 Representation Views

SimRobot offers two console commands (get & set) to view or edit anything that the robot exposes using the MODIFY macro. While those commands are enough to occasionally change some variables they can become quit annoying during heavy debugging sessions.

For this reason we introduced a new dynamic representation view. It displays modifiable content using a property browser. Property browsers are well suited for displaying hierarchical data and should be well known from various editors e.g. Microsoft Visual Studio or Eclipse (cf. 7.2).

In contradiction to the name the view can not only display representations but anything that is made available using the MODIFY macro.

7.2.1 Construction

A new representation view is constructed using the *vr* command in SimRobot. For example `vr representation:ArmContactModel` will create a new view displaying the ArmContactModel. Representation views can be found in the representation category of the scene graph (cf. 7.1).

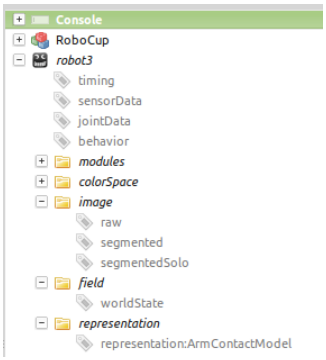

Figure 7.1: Representation views can be found in the representation category of the scene graph.

Property	Value
contactLeft	False
contactRight	False
pushDirectionLeft	NONE
pushDirectionRight	NONE
lastPushDirectionLeft	NONE
lastPushDirectionRight	NONE
durationLeft	0
durationRight	0
timeOfLastContactLeft	0
timeOfLastContactRight	0

Figure 7.2: Representation view of the ArmContactModel

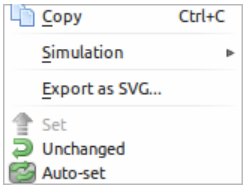

Figure 7.3: Representation view context menu

7.2.2 Usage

The representation view automatically updates itself three times per second. Higher update rates are possible but result in a much higher CPU usage.

To modify data just click on the desired field and start editing. The view will stop updating itself as soon as you start editing a field. The editing process is finished either by pressing enter or by deselecting the field. By default modifications will be send to the robot immediately. This feature is called the auto-set mode. It can be turned off using the context menu (cf. 7.3). If the auto-set mode is disabled, data can be transmitted to the robot using the *send* button from the context menu.

Once the modifications are finished the view will resume updating itself. However you may not notice this since modification freezes the data on the robot side.

To reset the data use the *unchanged* button from the context menu. After pressing the unchanged button the data will be unfrozen on the robot side and you should see the data change again.

7.3 Logging

In [7, Chap. 8.7] and [8] a logging mechanism that could record and store log files without the need of a permanent network connection has been introduced. However it required the debug process to be active. Therefore we were unable to acquire log files during real matches.

Therefore a new logging mechanism (called CognitionLogger) has been implemented. It does no longer depend on the presence of the debug process. Instead the whole logger has been realized as a single module running inside the cognition process. Just like every other module the CognitionLogger gains access to the representations that should be logged through the blackboard [2]. Therefore the logger has to require all representations that should be logged.

Each frame the CognitionLogger copies the representations that should be logged into an internal buffer. The buffer is organized into blocks. Each block stores 60 frames (roughly one second). If the logger runs out of buffer-space it discards the oldest block.

The buffer content is written to disk whenever the robot is in an idle state.

Depending on the situation the buffer can be rather big. Writing it to disk would halt the cognition process for several seconds. Therefore the writing process is spread over several cognition frames (see writeBlockSize in 7.3.1 for details).

7.3.1 Configuration

The behavior of the CognitionLogger can be customized by editing the *logger.cfg* file in the *Config* folder:

logFilePath Defines the path and name of the log file without file extension. E.g. */home/nao/logs/testlog*

maxBufferSize Size of the internal buffer (in seconds).

blockSize Size of one block (in byte). Note that maxBufferSize∗blockSize is always allocated on the NAO. This value strongly depends on the number and size of the representations that should be logged. If this value is too small it is not possible to log everything. If it is too large the memory might not be enough. A good value can be determined by observing the statistics for several minutes.

writeBlockSize Number of blocks that are written to disk per cognition frame.

cognition_representations A list of all representations that should be logged. The CognitionLogger has to require each representation mentioned here. In addition the CognitionLogger needs to know how to map the name of each representation to the actual object. This is done using a simple look-up table. This table contains all current Representations. Newly created representations need to be added to this table manually by modifying the constructor of the CognitionLogger.

enabled Enables or disables the logger.

outputStatistics If this flag is set to true the logger will print the maximum used block size to the console. These statistics can be used to optimize the *blockSize*.

7.3.2 Replaying Logs

Three new commands have been added to make replaying log files more comfortable:

log mr Enables all logged representations at once.

log fast_forward Jumps 100 frames forward.

log fast_rewind Jumps 100 frames backwards.

Bibliography

[1] RoboCup Technical Committee. RoboCup Standard Platform League (Nao) rule book, 2012-05-08. available online: `http://www.tzi.de/spl/pub/Website/Downloads/Rules2012.pdf`.

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

[3] Simon J. Julier and Jeffrey K. Uhlmann. A new extension of the kalman filter to nonlinear systems. In *Proceedings of AeroSense: The 11th International Symposium on Aerospace/Defense Sensing, Simulation and Controls*, pages 182–193, Orlando, FL, USA, 1997.

[4] Tim Laue and Thomas Röfer. Getting upright: Migrating concepts and software from four-legged to humanoid soccer robots. In Enrico Pagello, Changjiu Zhou, and Emanuele Menegatti, editors, *Proceedings of the Workshop on Humanoid Soccer Robots in conjunction with the 2006 IEEE International Conference on Humanoid Robots*, Genoa, Italy, 2006.

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

[6] Aldebaran Robotics. Architecture of the pref file device.xml / of key/values in almemory, 2012. Only available online: `http://www.aldebaran-robotics.com/documentation/naoqi/sensors/dcm/pref_file_architecture.html#us-actuator-value`.

[7] Thomas Röfer, Tim Laue, Judith Müller, Alexander Fabisch, Fynn Feldpausch, Katharina Gillmann, Colin Graf, Thijs Jeffry de Haas, Alexander Härtl, Arne Humann, Daniel Honsel, Philipp Kastner, Tobias Kastner, Carsten Könemann, Benjamin Markowsky, Ole Jan Lars Riemann, and Felix Wenk. B-Human team report and code release 2011, 2011. Only available online: `http://www.b-human.de/downloads/bhuman11_coderelease.pdf`.

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